

**A COMPUTER-AIDED SOFTWARE ENGINEERING TOOLKIT
FOR THE INTEGRATION OF CAD/CAM
APPLICATION SOFTWARE IN A NETWORK ENVIRONMENT**

by

Michele M. Grieshaber

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Mechanical Engineering

Approved:



Dr. A. Myklebust, Chairman



Dr. J. R. Mahan



Dr. M. Deisenroth



Dr. R. West



Dr. S. Jayaram

November 21, 1991

Blacksburg, Virginia

A Computer-Aided Software Engineering Toolkit for the Integration of CAD/CAM Application Software in a Network Environment

by

Michele Marie Grieshaber

(Abstract)

Much progress has been made in recent years in the development of Computer-Aided Design and Computer-Aided Manufacturing (CAD/CAM) tools for engineering design, analysis, and manufacturing. Unfortunately, most of these CAD/CAM applications were constructed independently and without standardization. In essence, they automate a single aspect of design, analysis, or manufacturing and cannot be combined to form a cohesive environment, since integration among applications was not addressed during the design phase of CAD/CAM application software creation.

In view of this problem, a novel approach is suggested for software integration of applications in a network environment. The distributed integration solution described in this dissertation employs a new "integration client/server" relationship, where the integration server is the core of the system, providing functions to translate or transform data between applications. The integration client consists of an interface with the server, a CAD/CAM application, and a user interface with the integrated system called the GRIM (GRaphical Interface Manager). There is only one integration server in the system, but there may be an unlimited number of clients.

The solution created for distributed integration is implemented in a Computer-Aided Software Engineering (CASE) workbench, geared specifically toward the generation of integration systems. This workbench is known as the CAD/CAM CASE Workbench, and includes an integration solution as well as standard CASE tools. The integration

solution contains several tools which will aid a system designer in generating integration systems for CAD/CAM applications. Included is the distributed integration solution described in this dissertation. The distributed integration solution is designed to facilitate the semi-automatic generation of an integration system. It consists of an integration server at the center of the integration system which manages the exchange of data among the integration clients. The integration clients are the CAD/CAM applications in the context of the integration system. To use the distributed integration solution, the integration system designer will customize portions of the structure charts, data dictionary, and module specifications contained in the workbench according to the needs of the applications programs and generate C-source code defining the integration system.

Using the distributed integration solution, the user will be able to effect data requests for applications, using the GRIM to interact with the system. All data exchanges are request driven. In addition to the distributed integration solution, this research includes a prototype integrated system which allows data to be requested from one application, and translated to a second for display and manipulation. The prototype was tested in a distributed environment and the results are described.

ACKNOWLEDGEMENTS

Funding for this research was provided by the IBM Corporation. I would like to thank Al Bracco, Alan Levit, and Tony Fiore for their help and support during the course of this project. I would also like to express my sincere thanks to Paul Clarke for helping me in more ways than he knows.

It is impossible for me to express the gratitude I have for the guidance I have received from my two advisors, without whom, my education and my outlook would not have been the same. Dr. J. R. Mahan served as my master's thesis advisor and has been instrumental in my development as an engineer and as a citizen of the world. His foresight and encouragement made it possible for me to spend two separate sejours in France. If he had not handed me an application for the Fulbright scholarship I may never have applied. My current advisor, Dr. Arvid Myklebust, has shown me the true meaning of the term doctor in philosophy. The constant encouragement he offered me over the past four years has helped me to become confident and capable in more than just my field of study. In addition, I would like to thank the members of my committee for their willingness to share knowledge and advice with me both before and during the course of this research.

I would further like to thank two individuals for getting me through long nights in the lab. Ludwig von Beethoven for composing his sixth and ninth symphonies and Martin Grunau for cheering me up and calming me down.

And finally, to my family for the love and support they have given to me during my time at VPI and throughout my life.

TABLE OF CONTENTS

1.0 INTRODUCTION	1
2.0 LITERATURE REVIEW	6
2.1 Integration of CAD/CAM Applications	6
2.2 CASE and Its Role in Integration.....	10
2.3 Summary	12
3.0 THE CAD/CAM CASE WORKBENCH.....	13
3.1 CAD/CAM CASE Workbench Background	13
3.2 Purpose of the CAD/CAM CASE Workbench	14
3.2.1 Categories of Integration.....	16
4.0 THE DISTRIBUTED INTEGRATION SOLUTION	19
4.1 The Integration Client	30
4.1.1 AP/SOCK Interface.....	33
4.1.2 The CAD Application	34
4.1.3 The GRIM Widget.....	36
4.2 The Integration Server	43
4.3 Communications	47
5.0 STRUCTURED ANALYSIS AND STRUCTURED DESIGN	49
5.1 Structured Analysis and Requirements Specification	49
5.2 Structured Design.....	53
6.0 INTEGRATION TOOLKIT	58
6.1 Data Flow Perspective of the Distributed Integration Solution.....	63
6.2 The Distributed integration solution in Structure Charts	72
6.2.1 GRIM Structure Charts.....	73
6.2.2 Client Application Structure Charts	79

- 6.2.3 Integration Server Structure Charts..... 87
- 7.0 DISTRIBUTED INTEGRATION SOLUTION PROTOTYPE..... 95
 - 7.1 The ACSYNT Client Application 98
 - 7.2 B-Spline Client Application 104
 - 7.3 The Prototype Integration Server 109
 - 7.4 Data Exchange in the Prototypical Integration System 111
- 8.0 CONCLUSIONS 119
- REFERENCES 123
- APPENDIX A: DATA DICTIONARY 126
- APPENDIX B: DATA FLOW DIAGRAMS / P-SPECS 136
- APPENDIX C: GRIM STRUCTURE CHARTS / M-SPECS..... 201
- APPENDIX D: CLIENT APPLICATION STRUCTURE CHARTS / M-
SPECS 251
- APPENDIX E: INTEGRATION SERVER STRUCTURE CHARTS / M-
SPECS 306
- APPENDIX F: UTILITIES,ETC..... 352
- VITA 359

LIST OF FIGURES

Figure 1: Paradigm for the generation of CAD/CAM integration systems [Penn91].....	2
Figure 2: CAD/CAM CASE Workbench.	15
Figure 3: Integration client/server relationship.	24
Figure 4: Motif toolkit above XtIntrinsics above X.....	26
Figure 5: Client AP/SOCK connected to application by proprietary interface.....	29
Figure 6: Client data routed through AP/SOCK Interface.	32
Figure 7: Basic GRIM widget.....	39
Figure 8: Sample transfer function.	46
Figure 9: Software development life-cycle.....	50
Figure 10: Components of a data flow diagram.	52
Figure 11: The Yourdon structured design process [Page88].	54
Figure 12: An example showing structure chart components.....	57
Figure 13: Integration toolkit in CASE environment.	60
Figure 14: Context Diagram of the integration system.	65
Figure 15: Data flow diagram of the integration server and clients - DFD 0.	67
Figure 16: Data flow diagram of the client interface - DFD 1.	69
Figure 17: Data flow diagram of the integration server - DFD 2.1.	71
Figure 18: GRIM widget displaying attribute list.	80
Figure 19: Sample relation file entries.....	89
Figure 20: Request/response sequence.....	102
Figure 21: B-Spline MODEL data structure.	105
Figure 22: Prototype client applications and integration server.	113
Figure 23: The ACSYNT client application.	114

Figure 24: The B-Spline client application prototype..... 116
Figure 25: B-Spline and ACSYNT clients running on the same workstation. 118

LIST OF TABLES

Table 1: GRIM opcode table	78
Table 2: Client application opcode table.....	84
Table 3: Integration server opcode table.....	92
Table 4: The ACSYNT client application opcode table.	103
Table 5: The B-Spline client application opcode table.....	108
Table 6: Integration server prototype opcode table.	112

1.0 INTRODUCTION

Much progress has been made in recent years in the development of Computer-Aided Design and Computer-Aided Manufacturing (CAD/CAM) tools for engineering design, analysis, and manufacturing. Unfortunately, most of these CAD/CAM applications were constructed independently and without standardization. In essence, they automate a single aspect of design, analysis, or manufacturing and cannot be combined to form a cohesive environment, since integration among applications was not addressed during the design phase of CAD/CAM application software creation. Additional refinement of individual tools will only provide diminishing returns until the sharing of data, and possibly functions, among them is also automated within the framework of an integrated environment.

The problem of software integration is difficult enough to solve on a single platform; the existence of software applications residing on different workstations in a network configuration significantly complicates the task. The first question addressed in this research is "What is an effective integration solution for dissimilar CAD software applications in a network environment?" The answer to this question involves CASE technology and the second question which arises is "How can successive implementations of this distributed integration solution be enabled in a semi-automatic fashion using CASE tools?" Although this is the order in which the questions are addressed in this dissertation, the overall problem which was resolved is the one which deals with the paradigm for generation of integration systems. Figure 1 depicts this paradigm and shows a CASE workbench. This workbench contains a toolkit designed to facilitate the task of an integration system designer by generating source code which describes an integration system. The ultimate goal of this research is to create a CASE

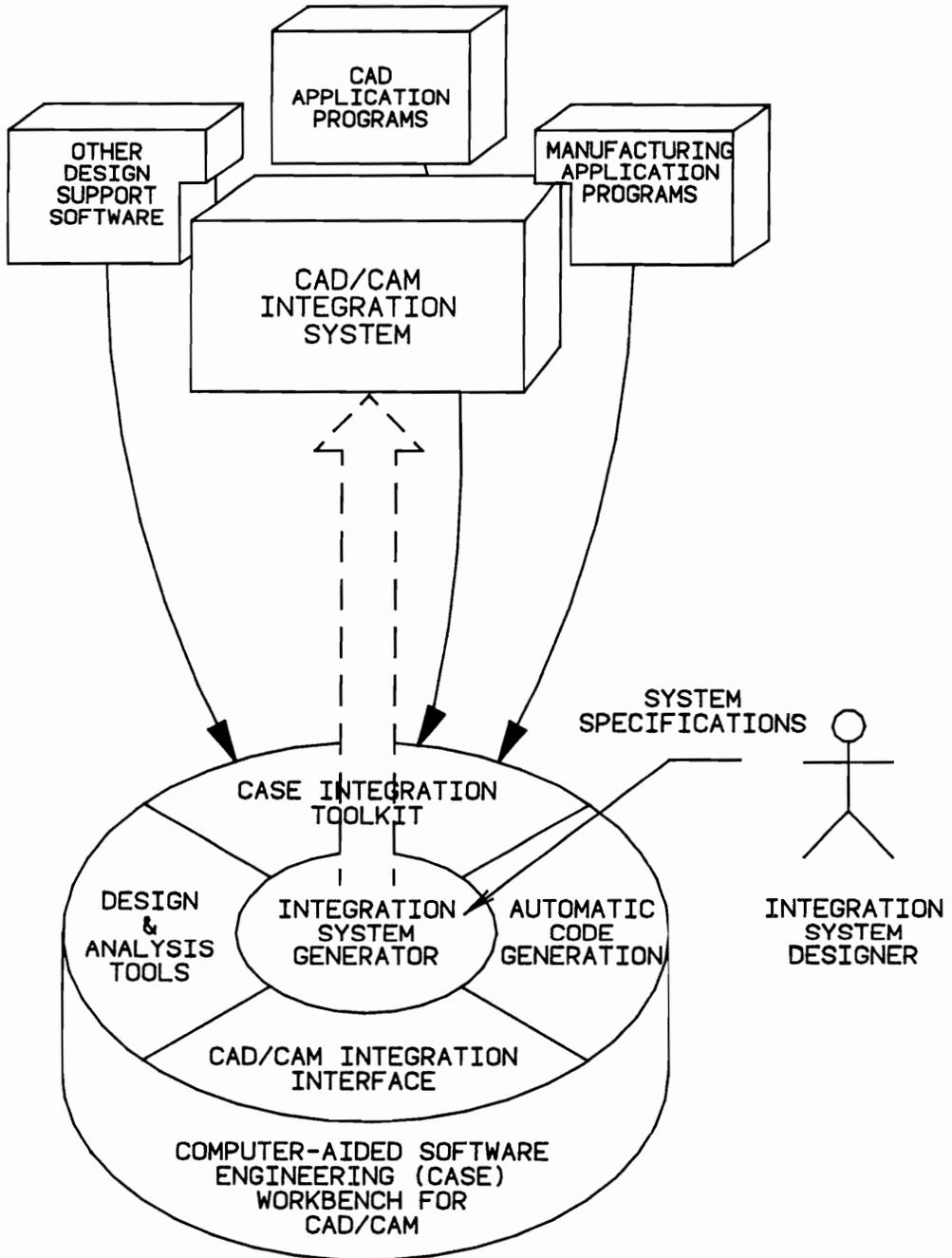


Figure 1: Paradigm for the generation of CAD/CAM integration systems [Penn91].

workbench capable of generating integration systems. In order to achieve this goal, it is necessary to develop a scheme for integration in a network environment which will lend itself to development into a set of CASE tools.

While the ultimate idea of CASE is to be able to specify design requirements and generate source code from those specifications, the state of the art has not quite reached that point. In reality, there may be some necessary degree of manual interaction in the development process, be it writing pseudo-code or actual code, even though the overall code structure may be generated. The function of a CASE tool is to leverage the development process, ideally automating it, but not necessarily. This is a critical point because integration may require, on the part of the integration system designers, in-depth knowledge of all applications targeted for incorporation in the system. This includes knowledge of the various data structures used in the applications. This condition is also imposed by the creators of an earlier integration enabler called the Environment for Application Software Integration and Execution (EASIE) [Rowe88]. They state "[data specific information] can only be provided by the program experts or application programmers who are intimately familiar with the codes being integrated." They also state that "no software tools can substitute for this knowledge."

Though development of the initial system relies heavily on knowledgeable individuals, maintenance of the resulting system will not require the same expertise, since the system will have been analyzed, designed, and implemented with the aid of a CAD/CAM CASE Workbench [Penn91]. The CAD/CAM CASE Workbench is a CASE workbench which has been modified to include a toolkit used primarily for the generation of integration systems for CAD/CAM applications. Important information on the parameters of the integrated system will therefore be available from the database

contained in the workbench. In addition, the workbench will enable programmers to reverse engineer, at both the design and the analysis levels, the codes to be integrated and extract necessary data from them to assist in producing the integrated environment.

The CAD/CAM CASE Workbench, originally specified by Pennington [Penn91], was designed to address the integration of engineering CAD/CAM applications. This is a novel approach to integration, considering that there are relatively few commercially available CASE systems geared toward the requirements of engineering; the bulk of the systems are business related. The CASE Integration Toolkit, contained in this workbench, will contain tools to implement the distributed integration solution described in this document. These tools will be in the form of generic data flow diagrams and structure charts which can be tailored to fit the integrated system through use of analysis and design tools and source code generators present in the CASE workbench. The resulting integration system will be described by these generic data flow diagrams (DFD's) and structure charts, along with other DFD's and structure charts created by the integration system designer. The final integration system will be generated from these structure charts, module specifications (which describe each module of a structure chart), and the data dictionary by using the C-source code generator included in the CASE workbench.

The following discussions explore the question of integration in terms of the integration mechanism, the feasibility of a system utilizing this mechanism in a network environment, and the ease with which an integration system designer can employ specific CASE tools, including the CAD/CAM Integration Toolkit to analyze, design, and realize the final integrated system.

The research conducted for this dissertation has the following goals:

- 1) identify mechanisms for interclient communication for integration
- 2) design CASE tools which will facilitate the generation of integration systems for CAD/CAM applications in a network environment
- 3) create a distributed integration solution which is effective in managing the exchange of data among applications in the integration system and which lends itself to developments into a set of CASE tools
- 4) demonstrate feasibility of the integration solution by using two CAD applications running in a distributed environment

The structure of the dissertation is as follows: a survey of pertinent literature which explores past work on the integration of CAD/CAM applications and CASE tools as a means to achieve integration, an overview of how the integration solution created in this research will fit into the CAD/CAM CASE workbench philosophy, an in-depth description of the integration solution, a discussion of the development process for the integration process explained using data flow diagrams, a discussion of the format of the tools created to effect the integration solution, and concluding remarks about the research presented in this dissertation. This is followed by details on the implementation of the distributed integration solution and a prototype integration system using this workbench for two aircraft design CAD programs.

2.0 LITERATURE REVIEW

The literature related to the integration of CAD/CAM applications is extensive and varies with respect to the levels of integration achieved. A few relevant papers are reviewed here.

2.1 Integration of CAD/CAM Applications

In July 1990 Pennington [Mykl90] conducted an industry-wide survey on integration. The survey results indicated that there does not seem to be a consensus on the exact interpretation of the term "integration". To some of the respondents, integration meant the sharing of information (data) among applications, even if that required the manual reentry of data. Pennington points out that of the respondents to the CAD/CAM integration survey, those who reported the most successful and flexible systems were companies who employed a common database to effect the integration. Two further conclusions obtained from the survey show that source code is readily available for most of the CAD/CAM applications targeted for integration. This availability is a result of in-house development of much analysis code. Furthermore, it was found that there is a growing trend in companies toward stand-alone workstations in a networked environment.

To date, most of the work on integration has been done using neutral formats such as IGES (Initial Graphics Exchange Specification). Liewald and Kencott [Liew82], though advocates of integration using such formats, were also cognizant of the limitations of this method on the goal of total integration. According to a report prepared by the Boeing Commercial Airplane Company [Brau85], some of the more

serious IGES flaws include its inefficient file structure, its inflexible data definitions, and its orientation toward graphical representation of a product's design. Farish [Fari90] conveys some of the frustration companies face when using neutral formats. He reports on five experimental projects sponsored by the SMMT (Society of Motor Manufacturers and Traders) which were conducted among several well-known British companies. The goal of the test was to swap CAD information in the best-known data exchange format, IGES. Farish reports that although "IGES can handle geometry, it is not reliable at maintaining the integrity of associated information."

In response to the need to resolve the limitations of IGES, a formal study called the PDES Initiation Effort was begun in 1985 [Fur190]. The objective of PDES (now called the Product Data Exchange using STEP) is to develop a neutral exchange medium capable of completely representing product data. STEP is the proposed international exchange standard. Unfortunately, the PDES exchange standard will not be available until sometime after 1995, and therefore does not address the immediate need for a solution to integration problems.

As a result of the disappointing results obtained with neutral formats, the recent trend has been towards other means of integration, mainly databases. Encarnacao [Enca90] contends that when contrasted with classical file systems (where every file contains data whose structure matches exactly the requirements of one specific application program), database systems provide for the integration of data for all applications within a corporation. Furthermore, data redundancy, which causes storage overhead and update problems, is avoided and only a minimum of data must be replicated by the system.

Several specific and sophisticated examples of integration via database exist. Fenves et al. [Fenv90] developed an integrated software environment for building design and construction which integrates seven independent, computational programs. They claim that integrated systems in industry achieve a high level of data integration by tying CAD/CAM software and analysis programs together through a shared database - a premise which they use in the development of their integrated system. Colton and Dascanino [Colt91] designed and implemented an integrated, intelligent design environment. The system enables the engineer to design custom mechanical parts and store related data in a database which is checked by an expert system to ensure manufacturability and assembly. It is important to note that this is a dedicated integrated system, meaning that only tools employed in mechanical design are included in this environment. Lu, Myklebust, and War [Lu86] developed an interface which writes geometric representations of helicopters directly into a computer-aided design system database via the Geometry Interface Module (GIM). This system is an excellent example of proprietary interface use as well, since the geometric models described by the analysis portion of the integrated system were subsequently viewed in CADAM. Reiss [Reis90] introduced a method of integration which combines message passing in a UNIX environment with databases. The major disadvantage of Reiss' approach is that to add new tools to the system, it may be necessary to modify existing tools to be able to interpret the new messages generated by the additional programs. Meyers [Meye91] comments on this fact in his article and expands his discussion to cover canonical representations of data structures, wherein a common structure for all data models exists. The idea of having all tools operate on the same data structures is attractive; however, in order to implement this in an integration scheme, existing applications must be rewritten to utilize the new data representation. Although this type of representation may be useful in the future, it does not address the current needs of the

system integrator. According to Christman [Chri84], automobile companies are prime users of integrated CAD/CAM systems with common databases. He contends that this approach allows several engineers to access design data and work on the same part simultaneously. This type of database interaction enables designers to practice concurrent engineering. As an example of the effectiveness of concurrent engineering, Chrysler has reported productivity improvements that range from 4:1 to as high as 70:1.

Although it seems that a great deal of success has been achieved by companies who take advantage of common databases, it is only fair to admit that no comprehensive database solution exists at the moment. Commercially available relational databases are not equipped to meet the demands of an engineering application. Part of the reason for this was addressed by Kim, Lorie, McNabb, and Plouffe [Kim84] who state that the primary difference between transactions in an engineering environment and those in conventional business applications is that an engineering transaction typically lasts much longer and can effectively disable the database from being accessed by any other user of the application. Kim et al. look at solving this problem by imposing the view that a long-lived engineering transaction is really a sequence of conventional short-lived transactions. A second problem with relational databases, as applied to engineering, is the lack of accurate data models. Guting [Guti89] looks at this problem in a paper where he describes the development of Gral, a relational database system that is extensible by user-defined data types and operations. Extensions needed for geometric database systems are addressed. As Date makes clear in his book on database systems [Date89], CAD/CAM is still considered a relatively new area of application for database technology, and research in the area is being vigorously pursued.

While work is being done in the area of engineering databases, some research is focusing on the development of tools with which to effect the integration. One such example is a paper by Jayaram and Myklebust [Jaya90] describing a method by which an expert system generates interfaces semi-automatically between application programs and CAD systems. The system then creates an accurate parametric representation of the solid geometry and places it in the CAD database. A second example is EASIE [Rowe88], which provides a methodology and a set of utility routines for a design team to build, maintain, and apply CAD systems consisting of large numbers of diverse stand-alone analysis codes. EASIE contains a centralized database in which data common to applications in the system is stored. This system addresses applications that run as batch programs on Digital Equipment Corporation (DEC) VAX computers.

2.2 CASE and Its Role in Integration

Carma McClure [Mccl89] defines computer-aided software engineering as the automation of software development. According to McClure, the basic idea behind CASE is to provide a set of well-integrated, laborsaving tools which link and automate all phases of the software life cycle. Traditionally the software life cycle consists of analysis and specification of requirements, design, implementation and coding, test and release, and maintenance. The following definitions from McClure are pertinent to this discussion on CASE:

- CASE tool - a software tool that automates (at least in part) a particular software life cycle task.

- CASE toolkit - a set of integrated CASE tools that have been designed to work together and to automate (or partially automate) a phase of the software life cycle or a particular software job class.

- CASE workbench - a set of integrated CASE tools that have been designed to work together and to automate (or provide automated assistance for) the entire software life cycle, including analysis, design, coding, and testing.

It is important at this point not to confuse the terms "integration using CASE tools" and "integration of CASE tools". In this dissertation, the former refers to the process of employing CASE tools specifically created for aiding system designers in achieving the integration of dissimilar CAD/CAM software into a cohesive design environment. The latter refers to the current goal in CASE where the software engineering environment is integrated by defining a framework, or integrated project support environment (IPSE), into which CASE tools fit together. A database, or repository, is part of the framework, and this addition allows all phases of the design cycle to access information about each other, keeping redundancy to a minimum. Several existing CASE tools are reviewed in articles by Smith [Smit90] and Oman [Oman90].

Marshall and Van Dyne [Mars86] discuss a design accelerator and integrator called DesignCenter, developed by Hewlett-Packard. This again is an example of a dedicated system, although it exists in an environment in which integrated design and CASE tools are used in conjunction to design hardware and software for micro-processor development. In research reports to the IBM corporation [Mykl90-1, Mykl90-2] and a dissertation [Penn91], Pennington presents a new approach to the integration of CAD/CAM application programs. Outlined are the requirements for a CASE workbench and toolkit to effect the integration of CAD/CAM applications.

2.3 Summary

The literature review suggests that an integrated design environment is key to productivity and competitive vitality. It is clear that although many attempts have been made to provide integrated systems, the tools for producing an integration system for CAD/CAM applications are lacking.

3.0 THE CAD/CAM CASE WORKBENCH

The work presented in this dissertation is, in part, based on research conducted by Pennington [Penn91]. It is the purpose of this chapter to lay the groundwork for the discussions which follow, by outlining requirements which apply to the CAD/CAM CASE Workbench defined by the document cited above. Many of the criteria initially specified for the workbench are used as assumptions and boundary conditions in this research. The most significant will now be discussed.

3.1 CAD/CAM CASE Workbench Background

Pennington describes a CAD/CAM CASE Workbench consisting of a combination of commercial and custom tools. A product known as Teamwork from CADRE Technologies is specified as the backbone of the CAD/CAM CASE Workbench. The Teamwork product includes a structured analysis tool, a structured design tool, and a C-source code builder. In addition to the Teamwork tools, a CASE Integration Toolkit, a High-level Autonomous Integration Model (HAIM), Interleaf Technical Publishing Software, and an integration framework supplied by the IBM Corporation were specified for the workbench. For the research conducted in this dissertation, the HAIM was used as a starting point for multi-platform integration ideas. The resulting integration system generation method is not represented in the CASE workbench as a separate entity. Instead, the definition of the CASE Integration Toolkit, developed at Virginia Tech, has been expanded to include all tools and conceptual models necessary for the creation of an integration system. Furthermore, the IBM Workstation Integration Framework has been replaced by CADRE Technologies' integrated project

support environment (IPSE) following the suggestion of IBM-Manassas. An updated diagram of the workbench components appears in Figure 2.

3.2 Purpose of the CAD/CAM CASE Workbench

The goal of the workbench is to leverage the task of an integration system designer by providing him with integration tools in a CASE environment. Aside from the tools normally found in a CASE workbench, a CASE CAD/CAM Integration Toolkit is included. This toolkit contains several implements geared uniquely toward the creation and establishment of an integrated system from dissimilar CAD and CAM applications. One element of the toolkit is an analyzer, initially conceived and designed by Pennington. The analyzer is currently under modification to enable it to characterize not only those applications whose source code is available, but also applications whose source code is not. For those without source code, characteristics of input and output data must be known. In this case, the analyzer will accept input from the integration system designer as to the kinds of data to expect in an output or input file.

Specifications of this type will enable the analyzer to categorize the data found in a file targeted for analysis. Because of this, it is conceivable for applications whose code has been modified to coexist in the integrated system with applications that can only be accessed through file I/O. More specifics on how this will be achieved will be given in the section on the distributed integration solution and in the section describing the CASE Integration Toolkit in terms which relate directly to the distributed integration solution.

Once information about the application has been extracted, the data which describe the analyzed application are placed in the CAD/CAM CASE Workbench database. The

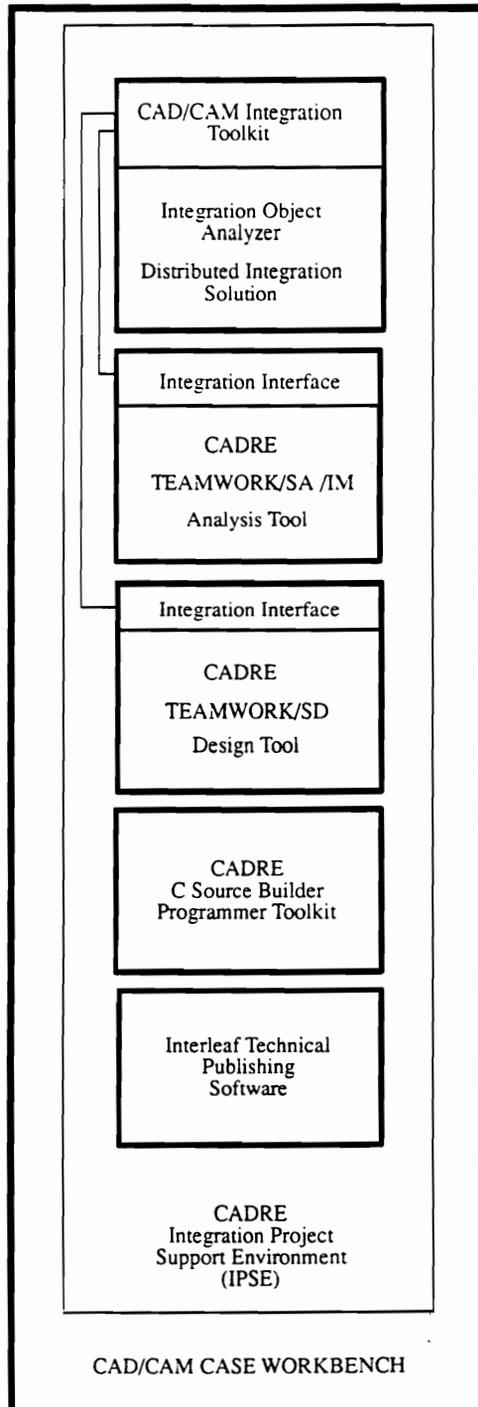


Figure 2: CAD/CAM CASE Workbench.

types of data extracted are documented by Pennington [Penn91]. Research is underway on the structuring and use of these data. At this point, the structured analysis and design tools of the CASE workbench can be used independently or in conjunction with other tools in the Integration Toolkit to design a final integrated system. In addition to the Integration Object Analyzer, there are several other tools which together form the Network Environment Integration System Enabler. The system enabler consists of data flow diagrams, structure charts, and related source code, which will assist an integration system designer in creating a complete distributed integration system. In order to evaluate the tools in this toolkit, it is important to define the meaning of integration as employed in the workbench.

3.2.1 Categories of Integration

There are a number of techniques used to integrate applications into a cohesive environment. These techniques can be generalized to fit into one of four categories [Rowe88], [Penn91]:

- 1) Rigidly connected interfacing
- 2) Rigidly connected coupling
- 3) Freely connected interfacing
- 4) Freely connected coupling

The term interfacing implies indirect data communication among programs which rely on an intermediate link to properly format the data. Coupling, in contrast, often implies the use of a database as a means of sharing and exchanging data.

Rigidly connected applications are those which are coupled or interfaced in such a manner that updates or additions to one or more of the applications mandate a

reworking of the other applications in the connected system. As an example of rigidly connected interfacing, consider a pair of applications where one application has been modified or designed to produce an output file in the form of the input file expected by the second application. Applications which employ rigidly connected coupling may appear to the user a single application, where desirable aspects of the applications have been extracted from their original location and restructured to work in unison. This method would require a common data structure to be used among the different components, in addition to being difficult to achieve.

Freely connected applications are those which are interfaced or coupled in a way which is independent of the process by which they were developed. In other words, applications can be independently added to or deleted from a system without affecting the structure of other applications in the system. An example of a freely connected interface is a neutral file format such as IGES or PDES. Freely connected coupling, on the other hand, allows applications to share data via a common database. For communication to occur, some sort of database management system is necessitated. This means that the applications themselves need to be modified in order to send and retrieve information from the database and its manager.

As was ascertained in the literature review, Pennington's industrial survey [Penn91] indicated that there is no consensus of the exact meaning of the term "integration". Because of this ambiguity, criteria for the CAD/CAM CASE Workbench were established to include the potential of generating an integrated system using any of the aforementioned integration schemes. The selection of the appropriate scheme is to be left to the integration system designer, based on the structure of the applications to be integrated and the desired end result. In an effort to facilitate the task of an integration

system designer, this research presents a distributed integration solution, employed by the CASE Integration Toolkit, which enable an integration system designer to develop systems which utilize rigidly connected interfacing, freely connected interfacing, or freely connected coupling. Although rigidly connected coupling is not specifically addressed by the distributed integration solution, the use of the analyzer in conjunction with the structured analysis and structured design tools of Teamwork will facilitate the creation of an integration based on this philosophy. Any of the other three integration methods can also be employed by using the Integration Object Analyzer and the CASE tools. The end product, the complete integrated system, will reside in one operating system immediately after creation. Various components can then be ported to other platforms, if desired.

4.0 THE DISTRIBUTED INTEGRATION SOLUTION

The term "distributed integration solution" implies an integration solution conceived for use in a distributed environment. A key goal of the distributed integration solution is to create a mechanism for integration which will be valid in a network environment, as well as one which will lend itself to development into a set of CASE tools based on the same distributed integration solution.

The requirements of the distributed integration solution are:

- possible database access and storage of pertinent CAD data
- inter-application communication
- applications running in a distributed and simultaneous environment
- functional access of other applications in the integrated environment without terminating the session on the current application
- transfer of data among applications via database and interclient communications
- a system executive which oversees and manages interclient and database interactions

It is necessary to clarify a few terms used in the requirements stated above. The system executive which will oversee interclient and database communications will be called the integration server. It will be described in detail as the discussion progresses.

Furthermore, interclient refers to the data exchange among the applications in an integrated system.

Using the requirements above as a starting point, several approaches to the problem of interprocess communication were considered as a basis for the distributed integration solution. One way of enabling two applications to communicate over a network is by using the X Protocol. The X Protocol runs above any lower-level network protocol

that provides bidirectional communication and can deliver unduplicated, sequential bytes of data [Nye90]. X provides a predefined set of queries and responses between two processes. Interprocess communication in X is effected using a mechanism called a selection. Selections allow communication between two clients on the same X server. The X server acts as an intermediary between user programs and the resources of the local system such as: the screen, keyboard, and mouse. It contains all device-specific code and insulates the applications from differences between display hardware. The applications in this scenario are clients of the X server. For the purposes of the distributed integration solution, the function of the server needed to be different than that of the X server. Instead of managing workstation resources, a server in the distributed integration solution needed to manage the exchange of data between applications. It may be possible to write an extension to the X server so that it is capable of managing interprocess communications, however, a disadvantage of this would be a reduction in efficiency if the server has to handle the management of the windowing system as well.

Another option would be to use the Remote Procedure Call (RPC) protocol which provides the same high-level communications which are used by the operating system. The RPC protocol utilizes the eXternal Data Representation (XDR) protocol which standardizes the representation of data passed in remote communications. In effect, XDR will compensate for differences in machine byte ordering. RPC relies on a transport protocol such as Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol (UDP/IP) to carry messages between communicating processes. A programmer can divide an application into a client side and a server side and use RPC as the mechanism for communication between the two. The application on the client side designates some procedures as remote, while the application on the

server side implements those procedures and declares them as part of the server [Come91]. When the client program executes one of the remote procedures, RPC collects values for the arguments and sends them in the form of a message to the server. RPC then awaits a response and returns values to the client. In this way, communication between client and server is carried out through procedure calls. A major disadvantage of the high-level RPC routines is that they are based on UDP/IP which restricts the RPC calls to 8k bytes of data. This is restrictive when considering the nature of data transactions for CAD/CAM applications.

Another possible mechanism for interprocess communications is the Network Computing System (NCS). NCS enables the distribution of application processes across resources in a network by maintaining databases that control information about the resources [IBM90]. NCS is object-oriented in that the programs are cast in terms of the objects they manipulate instead of the machines with which they communicate. An RPC runtime library handles communications for NCS. In this environment, RPC uses sockets for interprocess communications. NCS uses UDP/IP datagrams to send messages between clients and servers. Datagrams are unreliable. TCP, on the other hand, defines a reliable stream delivery that is useful for sending large volumes of data from one computer to another. Using an unreliable system for volume transfers requires programmers to build error detection and data recovery into their application programs.

Sockets were chosen as the means for communication between the integration clients and server due to their low-level flexibility and reliability when stream connections are used above the Transmission Control Protocol/Internet Protocol (TCP/IP). The socket abstraction is a product of Berkeley Software Distribution (BSD) and is essentially a

low-level applications programming interface (API) for interprocess and network I/O communications. It allows local or remote applications programs to set up a virtual two-way communication path and exchange data. The socket interprocess communication facilities reside on top of the network facilities; thus the communications are based on a reliable stream connection provided by TCP/IP. Sockets are basically file descriptors to which apply read, write, send, and receive subroutines. Just as programs open files when free access is required, application programs using BSD request the operating system to create a socket when one is needed. The system returns an integer value which the program will use to access the socket. Sockets can be created to work in one of two domains, UNIX or Internet. A socket which communicates in the UNIX domain can only interface with sockets on its host machine, while an Internet socket can communicate with sockets on its host as well as on foreign hosts. Internet is the term used to describe the technology which interconnects physical networks and makes them function as a unit. This technology hides the details of network hardware and permits machines to communicate independently of their physical network connections. Although the choice of sockets as a mechanism for communication is operating-system dependent, recent incorporation of sockets in systems other than UNIX, such as IBM's VM and MVS, allow the latitude necessary for this type of solution.

Other requirements used in the design of the distributed integration solution, including the ones mentioned above, are delineated as follows:

- 1) The distributed integration solution should be valid in a network environment.
- 2) Compatibility with the X-Window System is desired.
- 3) The emphasis is on interactive CAD/CAM applications.
- 4) There is a possibility of database support of the integrated system.

- 5) Source code for applications should not be a requirement.
- 6) The distributed integration solution is valid for 3 types of integration : rigidly-connected interfacing, freely-connected interfacing, and freely-connected coupling.

The distributed integration solution created as a result of this research is based on the definition of an "integration client/server" model. Given the constraint that the model for integration must be valid in a network environment, a client/server relationship seems well suited to the task. In this arrangement, the integration clients are actually the CAD/CAM applications programs with a socket-based interface to the server (AP/SOCK Interface) and a graphical interface to the user (GRIM widget) appended to it. It is important to note at this time that in the discussion which follows "application" refers to the CAD/CAM applications program while "client" refers to the application in the context of the integrated system. In other words, the term client implies the CAD/CAM application, the AP/SOCK Interface, and the GRIM widget as a unit. The integration clients and server communicate using sockets as a mechanism for data exchange. This configuration is shown in Figure 3. Note that clients on different workstations can communicate with the server which resides on a separate machine.

The integration server is the administrator of the integrated system, meaning that it is the server's job to determine which applications are currently connected to the system and to enable users to interact with those applications by specifying data exchanges. The server is also an intermediary through which all data, including requests, bound for applications in the integrated system must pass. The mechanism in the server which acts to convey model data is the transfer function. Each application in the system has a corresponding transfer function at the server which accepts model data from it and translates, transforms, or relays that data to another application in the system.

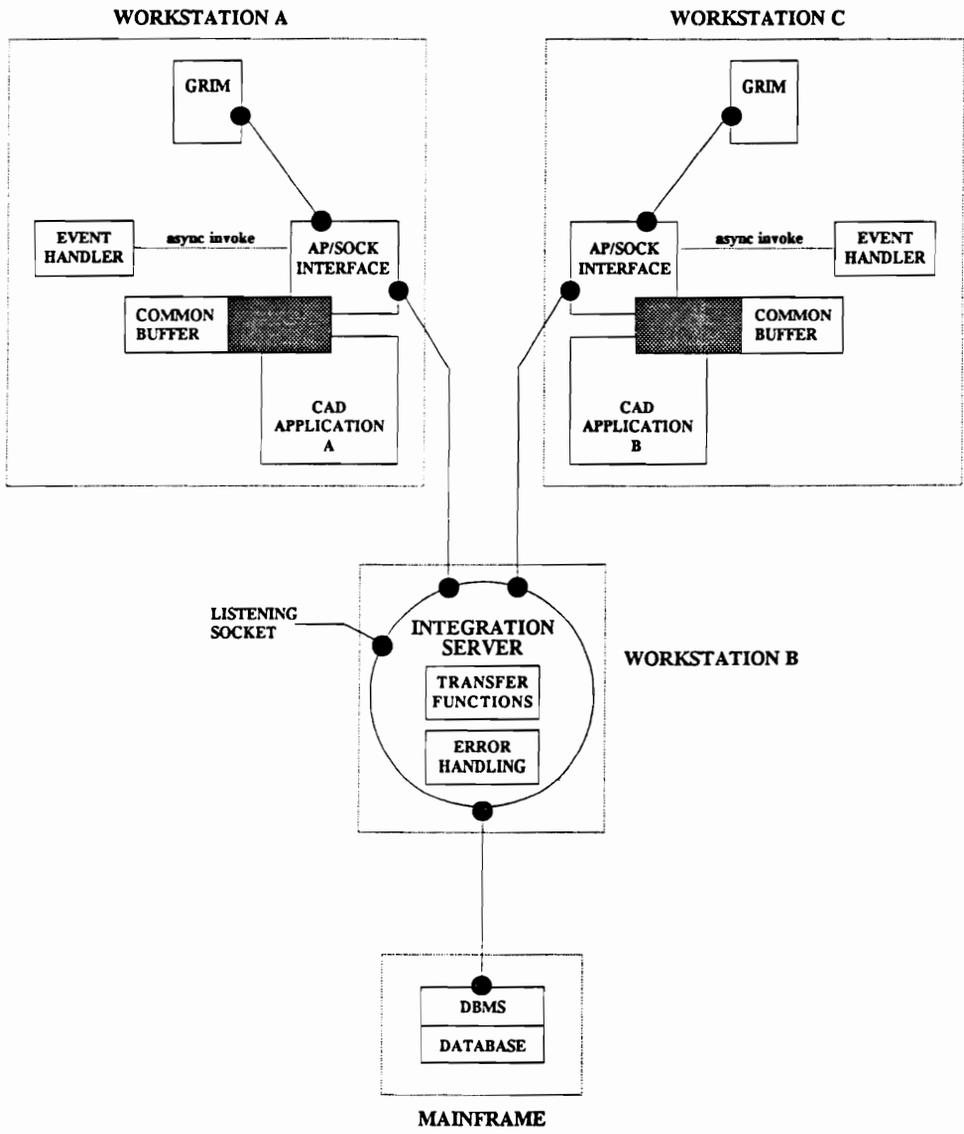


Figure 3: Integration client/server relationship.

The requirement for compatibility with a windowing environment goes hand-in-hand with the emphasis on interactive applications. The distributed integration solution is designed to handle applications in a network environment, but the possibility exists that two applications could reside on the same workstation. If the applications are of an interactive nature, two applications co-existing on one workstation necessitate a windowing environment. The X-Window System was prescribed since it is the most reliable and well-known of such systems to date. Interactive applications are not the only type which can be handled by this distributed integration solution. In fact, programs which run in batch mode are a simpler case than their interactive counterparts, since they automatically guide input data through the program and produce a defined output. In addition, a Motif widget interface was selected as the means by which individual integration clients interact with the user. Motif is a graphical user interface based on the X-Window System. The Motif toolkit sits above the XtIntrinsics toolkit which is part of X-Windows. Figure 4 shows how each of these elements is related to the others. A Motif widget is an object which provides a user-interface abstraction, in other words it combines scrollbar widgets, list widgets, button widgets, etc. to create a custom user interface. Motif was chosen for interface development because it has a consistent look and feel and because it is a toolkit which accompanies the X-Windows software, which has been chosen as a requirement for the integrated system. Since Motif is included in the X-Windows software, there is no need to mandate extra software whose sole purpose would be to provide an interface.

As can be seen in Figure 3, the proposed integration system will consist of integration clients and servers in a networked workstation environment, with the possibility of a common database used to store and transfer data. Note that this is not the same

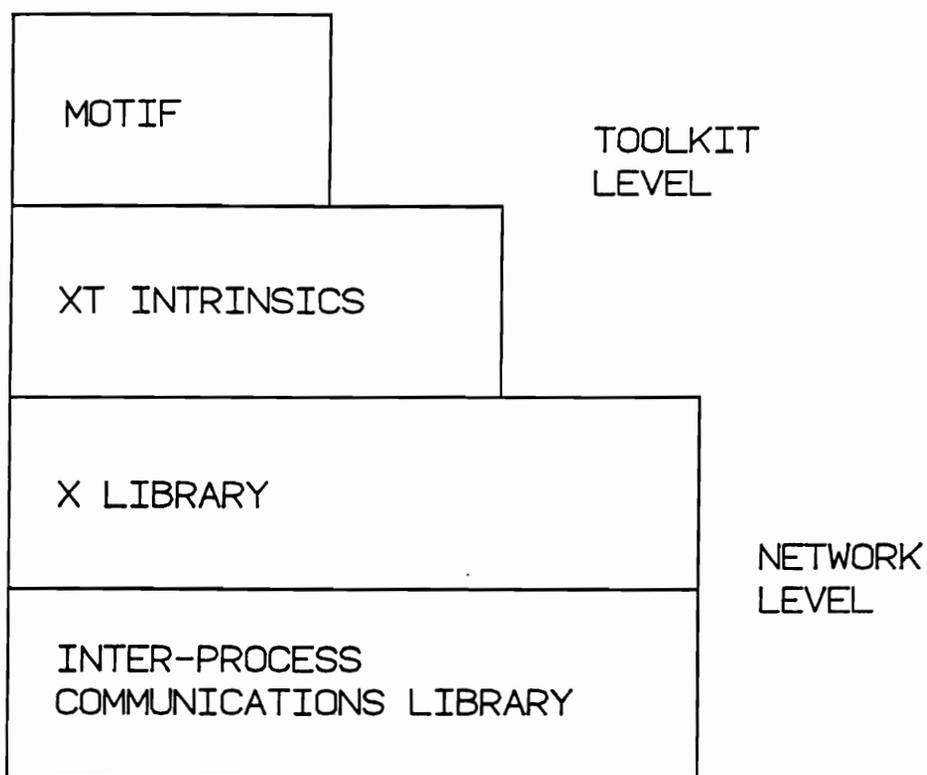


Figure 4: Motif toolkit above XtIntrinsics above X.

database that was discussed in the context of the CAD/CAM CASE Workbench. The database is an optional member of the integrated system, since the integration server can function as an intermediate link in the process of data exchange between client applications. The most likely location for the database is a mainframe environment, though a workstation platform would also be suitable. The mainframe specification was made because of the need in industry for a reliable, secure means for storing proprietary data. This is a reasonable assumption considering IBM's recent announcement of software that will transform its mainframe computers into massive database servers. The software is currently geared toward PC access, but this announcement shows the trend toward the use of mainframe computers as information warehouses [Pall91]. In the distributed integration solution, the database will most likely be used for data storage and retrieval. The database will communicate with the integration server in a manner similar to that of the integration clients, employing a database manager for data access. Both IBM VM and MVS mainframe environments support the socket abstraction, although asynchronous functionality is difficult to implement. In any case, it is not necessary for the database to utilize an asynchronous socket, since blocking sockets would not have an adverse effect on the performance of the database or on the other components of the integrated system.

Within the integrated system, applications with and without source code can coexist. In the case where application source code is available, the integration system designer can utilize the analyzer present in the Integration Toolkit to extract specifics about input and output capabilities of the application as well as data structure. The source can be modified, or more simply, modules that the application would normally use to extract data for display can be called by the client modules which interface with the integration server. This allows the integration system designer to exploit any internal databases or

data structures that the application may utilize. In general, source code should not need major modification. If source is not available, the application must be able to produce output files and read input files. If a proprietary interface is available, a layer between the client interface with the server and the application can be built to facilitate data exchange into and from a common storage area between the two interfaces. This concept is illustrated in Figure 5.

The distributed integration solution must be capable of supporting three types of integration. One way in which the distributed integration solution can be implemented is by creating a freely connected interface. In this type of system, the integration clients send data in their own format to an integration server. At the server, the receiving client is determined and the data is translated or transformed into a format compatible with its data structures. The data is then sent to the receiving client where they are displayed, analyzed, or stored. If a database is added to the scheme, as was proposed earlier in this section, a freely connected coupled relationship develops between the applications which access the database via instructions to the server. As an example of rigidly connected interfacing, consider an application which can only receive data in the form of a strictly defined input file. In a normal system, any other application wishing to communicate with the input-restricted application would have to modify its output to be structured like the input file. In the system based on the distributed integration solution, the applications send their data, as they produce it, to the server where it is reformatted to produce an input file of the type expected. An in-depth description of the integration client, the integration server, and the communication protocol used in the distributed integration solution follows.

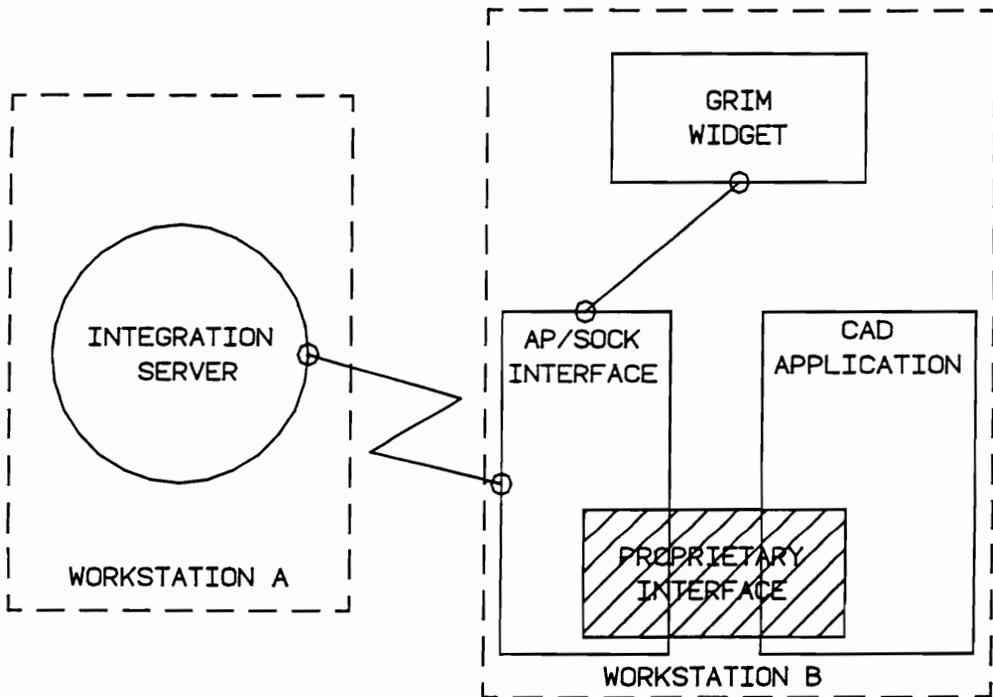


Figure 5: Client AP/SOCK connected to application by proprietary interface.

4.1 The Integration Client

In the following discussion, emphasis is on applications such as CAD, CAE or CAM. For simplicity they are referred to as CAD applications. The integration client consists of three elements: a CAD application, an AP/SOCK (Application/Socket) Interface, and a GRIM (GRaphical Interface Manager) widget. This configuration is shown in Figure 3. The relationship of these elements to one another and to the integrated system will now be discussed.

The AP/SOCK Interface acts as a front end to the CAD application in the integrated system. One of its purposes is to invoke the CAD application after the AP/SOCK has initialized sockets for communication with the GRIM widget and the integration server. The combination of the AP/SOCK and CAD application (excluding the GRIM widget) will, from now on, be referred to as the client application. The AP/SOCK and the CAD application communicate via a common buffer. During development of an integration client, the integration system designer defines a common buffer area through which the CAD application and the AP/SOCK Interface interact. The analyzer contained in the CASE Integration Toolkit aids the integration system designer in locating code related to important data for inclusion in common buffer storage. More discussion on how this is achieved will soon follow. Though the current trend is to develop CAD applications using the C programming language, there exist a large number of applications still on the market which were written in either FORTRAN or Pascal. Since the AP/SOCK Interface is C based, the problem of compatible data buffer areas for programs written in Pascal and FORTRAN must be addressed. Pascal allows pointers to memory locations. Therefore, a link between the interface and a Pascal-based application need only exchange a pointer location representing a common

data buffer. FORTRAN, on the other hand, presents more of a challenge, since it hides memory manipulations from the user. In this case, several arrays containing integer and real variables will be sent as arguments to the application program from the interface. This is a feasible solution since FORTRAN treats arguments passed into a subroutine as pointers. The mixing of languages presents few problems in a UNIX environment.

The AP/SOCK and the GRIM widget are connected by a socket link which enables the two processes to run in parallel and communicate only when user action at the widget is detected. Because they run separately, the CAD application is able to execute independently. Similarly, the GRIM widget, which is event driven in nature, can use its own management system to poll for incoming events. Motif widgets are essentially event driven, meaning they await events generated by users, then act on these events. There is a way for a widget to break out of the event loop, but this procedure is effective only if the process executed after the break takes less than a few seconds to complete. This being true, there is no effective way of combining the GRIM widget with the CAD application without a major overhaul of the application's source code. It is much more desirable to separate the two components and have them run in parallel, communicating only when user interaction is detected by the widget.

It is important to note at this point that all data, (which can be in the form of a message or actual information) which originate from any of the three components of the integration client and which are bound for the server, are sent to the integration server on the socket controlled by the AP/SOCK interface. This concept is illustrated by Figure 6. This figure shows choice data generating a request at the GRIM which is transmitted to the AP/SOCK Interface. Either this request or data compiled in response

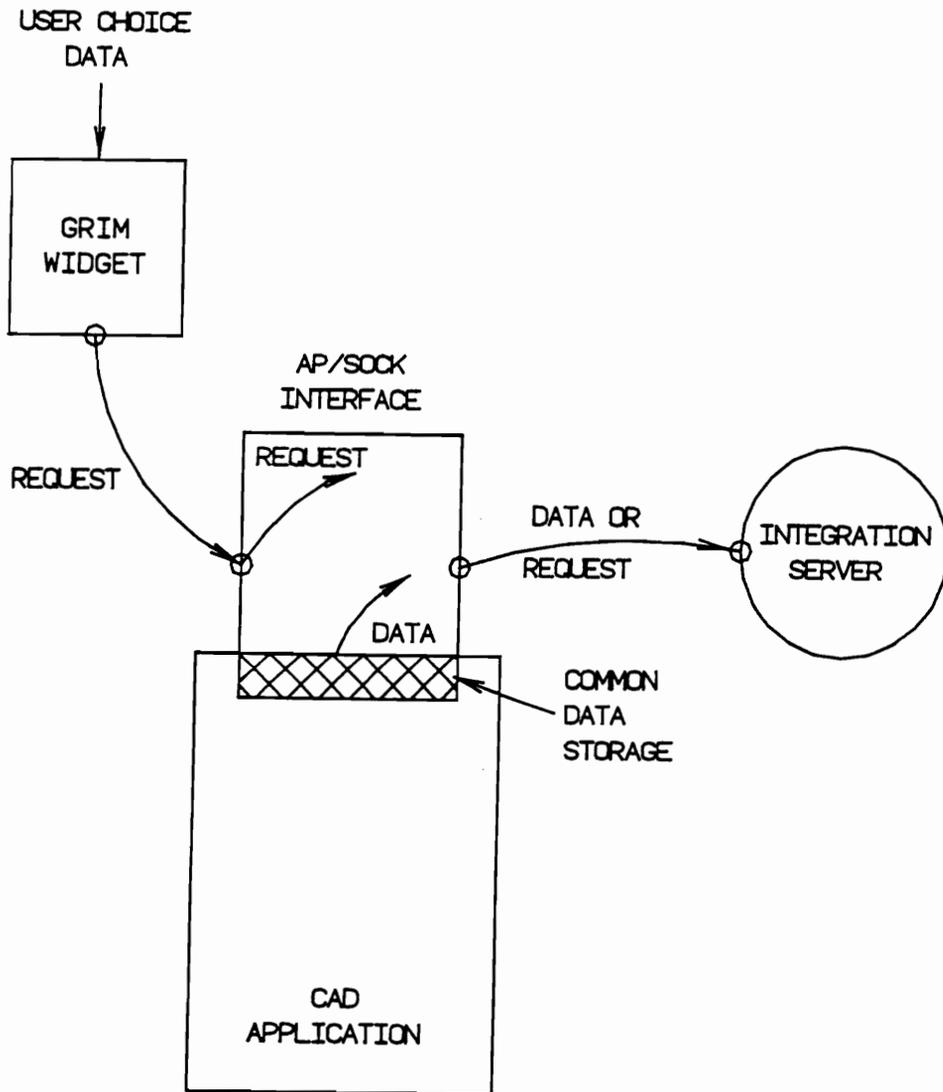


Figure 6: Client data routed through AP/SOCK Interface.

to a different request can be sent to the integration server. The main purpose of the AP/SOCK Interface is to act as an intermediary between the application program and the integration server, and the GRIM widget and the integration server.

4.1.1 AP/SOCK Interface

The first task of the AP/SOCK interface is to create an asynchronous Internet socket for communication with the integration server. Internet sockets are able to communicate with sockets based on a foreign host that are connected by an Internet network. Next, the interface creates an asynchronous UNIX socket for communication with the GRIM widget. UNIX sockets are valid for communication on the same workstation. This means that the client application and the GRIM widget, though in separate windows, appear on the same workstation. If the CAD application is run in batch mode with no graphics, the only interface displayed to the user will be the client application's corresponding GRIM widget. When both sockets exist, the AP/SOCK sends connection requests to the integration server and GRIM. The client socket interface will wait to proceed until both connections have been accepted.

With connections established, the interface invokes the CAD application and passes it any variables that have been declared as common. It may not be necessary to send arguments if the application is C based. In this case, a "global extern" statement will work. But, as previously discussed, Pascal- and FORTRAN-based programs will require an argument list. By declaring variables from the CAD application in the AP/SOCK, the common data buffer is established. It is necessary to access the application source code and delete initializations of these variables since the CAD application is no longer the controlling program. After invoking the CAD application, the interface sends a request to the server for a list of clients currently active in the

integrated system, from which its client application can request data. This list, when received, is relayed to the GRIM for display as user choices.

Because asynchronous sockets were created, an event handler must be established to check for incoming signals, without interrupting execution of the CAD application. The nature of asynchronous sockets is to allow the CAD application to be manipulated by the user until a signal is detected on one of the sockets. Once a signal occurs, the application is suspended and the signal is evaluated to determine on which socket it occurred and what it contains.

4.1.2 The CAD Application

The CAD applications discussed in this section are interactive in nature. One must bear in mind that batch programs are a simplified case since they require defined inputs and outputs. There are two cases to consider: application source code is available and application source code is not available. The former is the case considered in this research for the creation of a prototypical system, since it tends to be the more complex of the two.

In the case where application source code is available, the integration system designer analyzes the contents of the application in the early stages of integration system design. Based on the information extracted with the analyzer, the designer can define data structures for transfer to other application clients within the integrated system. This information is also used to establish a common data buffer which links the client socket interface (AP/SOCK) with the CAD application. It is through this common data buffer that the two components (interface and application) communicate. For example, when there is incoming data which must be taken from the socket by the AP/SOCK Interface

and stored in an area to which the application program has access, it is the common data buffer which receives this information. Recall that one of the assumptions made in this research is that the CAD applications are of an interactive nature. This means that in order to transfer the data sent by another application to the desired module within the receiving CAD application, user interaction is a necessity. For instance, wing data may be requested by a CAD aircraft design application that performs finite element analysis. When the data are received by this application, they could be in a module which only displays the geometry and does not analyze it. To proceed to the correct module, the user must interactively select menus until the desired analysis section is reached. It is not logical to automate the process of data placement within an interactive application, because that would be contrary to the nature of the application. Application programs which run in batch mode can also be treated under the distributed integration solution, since they need data passed directly into a receiving module, and so no further manual interaction is necessary.

If source code for the application is not available, there must be a way to characterize the output and input files so the Integration Object Analyzer, contained in the CASE Integration Toolkit, can be tailored to filter important information regarding the output file for use in the integrated system. This means that the analyzer must be capable of accepting information from an integration system designer about the format of the output file in question. The analyzer can then characterize the contents of the file and allow for its incorporation into the integrated system. For instance, the integration system designer can input data to the analyzer which, in turn, describes the kinds of data which will occur in a targeted input or output file. He basically specifies data types and format constraints for the file and then uses the analyzer to determine exactly what each input or output file of that type contains. Furthermore, applications which

lack source code must be able to accept input in the form of an input file, or must employ a proprietary interface as a means of receiving data from an exterior source. In either case, the format of the input file, or the mechanism used by the proprietary interface to receive data must be known in order for the appropriate transfer function to handle data in the integration server. An application without source code is incorporated into the integrated system by encompassing the application with the AP/SOCK Interface as usual; however, the event handler must be structured in a slightly different manner. The event handler will be responsible for extracting data from and passing data to these applications using output and input files. For example, when data is received by the interface which is destined for the application, the interface constructs an input file and then, if necessary, invokes the application so that the file can be read. If the application does not need to be restarted in order to read the input file, a message will appear in the application's GRIM widget instructing the user to manually take steps to read in an input file of the given name. When an application in the integrated system requests information from the sourceless application, the client application's widget once again instructs the user to manually create a named output file containing the current model displayed and gives it a specific filename. When the file has been created, the user informs the widget by selecting a proceed button which alerts the handler to send the file to the server for transfer to the requesting application. The steps previously outlined will, of course, depend on the integration system designer's implementation and the structure of the CAD application.

4.1.3 The GRIM Widget

The GRIM is a Motif widget-based user interface. Given that Motif interfaces are essentially event driven, widgets await events generated by users, then act on them. In an event driven application, the user is in charge since the application is always waiting

to react to a user command. A function called `XtMainLoop` checks for user input and work procedures. Work procedures are instructions for the main event loop to branch from event checking and perform a specified function when no user input is detected. This basically means that work procedures can only be executed if user input is not coming in with great frequency, otherwise the event queue gets priority. Another concept that is key to the understanding of widgets is the callback function. Callback functions are associated with widgets (for example, a pushbutton widget) when it is created. This enables an action to be ascribed to the widget (for example, quit program when pushbutton activated). If a widget is meant to do something, it has a callback which is invoked when the widget is activated.

The GRIM widget runs in parallel with the client application (AP/SOCK interface and CAD application). It is connected to the AP/SOCK of its owning client application by a socket, so that communication can occur when user interaction at the widget is detected. The term "owning application" implies the application for which the GRIM provides an interface. The socket connection between the two interfaces allows the application to proceed without having to manage an external interface for data exchange in the integrated system. This configuration makes it possible to display a user interface for the client even when application source code is not available for modification. The key concept is that there is independence of the user interface with the integrated system and the CAD application. Because they run in parallel, the CAD application and the GRIM widget will be contained in separate windows on the workstation. Remember that they will both be on the same machine since the GRIM and the client application are connected by UNIX sockets. The GRIM widgets window is not in any way attached to the client application's window, and therefore it is necessary to indicate that the GRIM widget belongs to a specific client application.

This is done by having the widget display the name of its owning application above the selection list which contains client names for data requests.

The GRIM widget is basically a generic entity in the integration system. This means that all GRIM widgets have identical code structure and content, with the exception of the pathname used to define its UNIX socket file. This pathname must be unique for every GRIM in the integrated system and must be known by the owning client application so that communication can take place.

The GRIM widget is similar to the integration server in that it uses a listening socket to detect connection requests. However, unlike the integration server, the GRIM only expects one client, its owning client application, to attempt to connect to it. When the client application does request connection, the GRIM accepts and requests the owning application's name for display in the widget. Using Figure 7 as a guide, we will now explain what makes up a basic GRIM widget. The idea behind the creation of this widget is, first, to establish a main window widget which will house other widget types. To the main window we add a menu bar widget which allows us to place menu choices in a menu bar format. The item of this type present in the figure is labelled ACTION. When selected, ACTION allows the user to reset the widget toggles or to exit and destroy the widget.

Once the menu bar has been completely defined, a selection box is added to the main window widget. The selection box includes two sub-widgets: the selection list widget, and the selection dialog widget. The selection list will contain the names of the clients connected to the integrated system from which the widget's owner (owning client

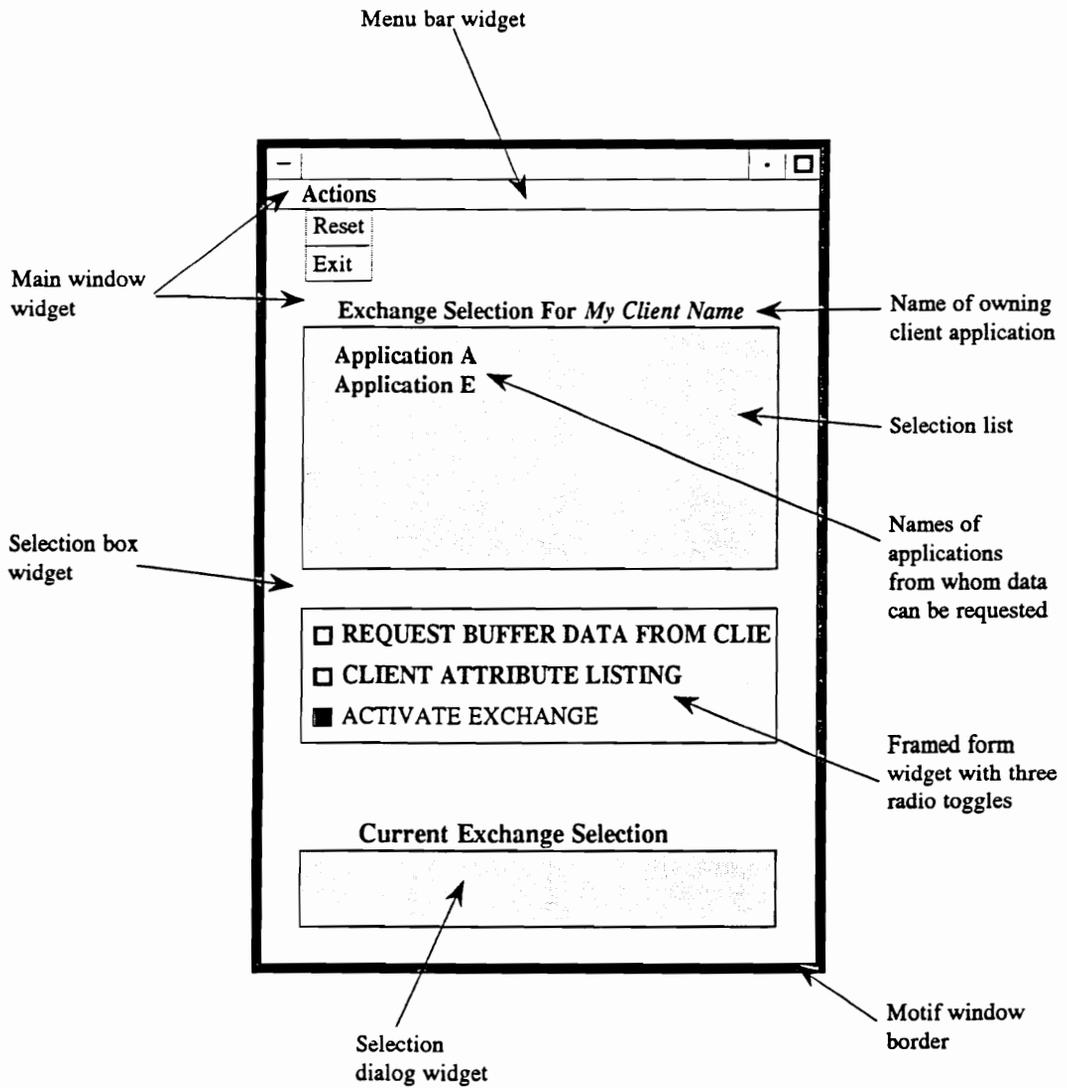


Figure 7: Basic GRIM widget.

application) can request data. Until a client has connected to the GRIM, this list remains empty. This is because the GRIM has no way to access the integration server to request the list until the client's AP/SOCK intervenes. The selection dialog is shown at the bottom of the widget.

A form widget is then added to the selection box widget. The form widget is enclosed by a frame and three radio toggle buttons are created for it. These toggles are defined as:

- 1) Request Buffer Data From Client
- 2) Client Attribute List
- 3) Activate Exchange

Each toggle has a callback associated with it. The activate exchange toggle, as seen in Figure 7, is initially greyed out until enough information (request for buffer and responding application's name) has been gathered to effect an exchange. When it is selected, it will display an information panel with the name of the application who will respond to the request. At this time, the user has the option of proceeding with the request or canceling it.

Although there are three toggle buttons, they represent two possible actions:

- 1) need current buffer data from a client in the integrated system
- 2) need some component of data from a client in the integrated system (an example would be to request only the wings from an aircraft model)

To request buffer data is to request that the model or data currently displayed by a client application in the integrated system be transferred to the requesting client. If

only a component of that data is desired, a client must first request that the client application with the desired data supply a client attribute list.

For applications in the integrated system that are able to supply data to other applications, a client attribute list is needed. The client attribute list (CAL), or simply attribute list, is a list which delineates the kinds of data which can be extracted from the application. The attributes are a component-like list describing the kinds of data which are representative of the current model. For instance, if the current model in an application represents an aircraft, the attribute list could supply separate components as options. The list is employed to give the user the option of extracting specific data from a model, instead of requesting the entire set.

During creation of the GRIM widget, each button, list, etc., is associated with a callback function. This means that if an event is detected at that widget element, a callback function corresponding to the action is invoked. The callback defines the action to be performed based on the event.

After the GRIM widget has been created and realized, the XtMainLoop takes control and checks the work procedure when no events are queued. The work procedure used by the GRIM allows the socket functions to be checked and processed. The function used to check for socket activity is a BSD socket subroutine called *select*. Keep in mind that sockets are basically file descriptors which correspond to a process instead of a file. The select subroutine checks the specified socket descriptors to see if they are ready for receiving, sending, or if an exceptional condition is pending. The select procedure allows a server (in this case the GRIM widget interface) to interrupt an activity (polling for events), check for incoming data, and then continue processing the

activity. In this respect the GRIM acts as a server where its only client is the owning client application. Another difference between the GRIM "server" and the integration server is that the GRIM functions in the UNIX domain instead of the Internet domain as the integration server does. This means that socket connection is described by a pathname instead of a port number. It does, however, use a listening socket which assigns a new socket descriptor to the client application which requested connection.

It has been established that the GRIM and the client application (AP/SOCK interface and the CAD application) must work together, but run independently of one another. In order to reduce the complexity of starting several processes, a script is used to start the client. The script starts the GRIM widget as a background process, sleeps (waits) for three seconds, then starts the client application. Background processes do not need a shell in which to run. By running the GRIM as in the background, only one shell is needed for the client instead of two (one for the client application and one for the GRIM). The sleep is necessary because the GRIM widget needs sufficient time to initialize and set up its listening socket before the client application attempts connection. The following is an example of a script file created for this purpose:

```
widget_myclient &  
sleep 3  
my_client
```

where **widget_myclient** is the executable for the widget, **&** makes it a background process, **sleep** tells the system to wait three seconds before invoking **my_client** which is the executable for the client application.

4.2 The Integration Server

The integration server is the core of the integrated system; however, a user of the integrated system could very well be ignorant of the fact that the integration server exists. This is because the integration server is a background process which is continuously running on the host workstation, allowing integration clients to connect and disconnect. The server has a dedicated port number which corresponds to its listening socket. This port number cannot be duplicated by any other process running on the workstation. Only clients that recognize this number and the internet address of the host machine can connect to the integration server. By concatenating the unique internet address with the socket address (port number) an internet socket address is produced, which enables clients to locate and utilize the server. As a guideline, port numbers up to 255 are reserved for official Internet services. Port numbers 256-1023 are reserved for other common services. Some operating systems have additional constraints on port numbers. For example, IBM workstations with the graPHIGS API installed have socket ports reserved for the interprocess communications between the graPHIGS shell and nucleus. The prototype that was developed in this research to test the validity of the distributed integration solution uses an integration server with a port number of 2000.

If the process for which a connection request was directed is listening at the well-known port, it services the request and either uses that same port for the duration of the connection (as does ftp - file transfer protocol), or creates a new port which is assigned to the client process. By freeing the listening port in this way, the server process can continue to accept connections from other client processes. Requests for connection are handled so that multiple clients may access the integration server. As clients are accepted for connection, the server adds the client's socket identifier to a data structure

which allows descriptors to be cross-referenced with client name. Later this is how the integration server will determine where to send data when only given the receiving application's name.

The listening socket is a key element of the integration server. When first invoked, the integration server sets up a socket which listens for incoming connections from integration clients. These connection requests are directed through the AP/SOCK Interface of an integration client during a client's initialization stage. When a connection request is detected, the listening socket accepts the connection, thereby creating a new and dedicated socket on which communication between the integration client and the server takes place. The listening socket then returns to its task of waiting for more incoming connections. This process is interrupted when a signal is detected on one of the sockets dedicated to an integration client.

Once a connection has been established between sockets at the client and server processes, the sending and receiving of data can occur. There are several ways in which to accomplish this; the method chosen in the prototype is to use *send* and *receive* subroutines supplied in the BSD libraries.

The integration server uses the *select* subroutine to check for activity on the sockets managed by the server. When a signal arrives, the socket on which it occurred is determined. If it occurred on the listening socket, this indicates a client request for connection, and a new socket dedicated to that client is created. If the signal came in on one of the other existing sockets (connected to integration clients), a block of data called a header is read from the socket and evaluated. The header will be described in more detail in the succeeding section on communications. In brief, the header contains

a major and minor opcode (operation code) which combined give the integration server enough information to locate the module which will evaluate the signal.

The integration server contains a library of transfer functions which are used to transform or translate data from one application to the format of another. These transfer functions are unique in that they must be specifically designed to handle the order and type of data sent by one application and expected by another. Therefore, it is conceivable that for every n applications in the integrated system there will be n transfer functions designed to accept data from each application and $n-1$ sub-categories to each transfer function. By sub-categories it is meant that based on the receiving application, there is a switch statement in the transfer function which determines a module which will reformat the data sent by the transmitting application. This process is illustrated by Figure 8. It is feasible that a transfer function will have more than $n-1$ sub-categories. For instance, it may be necessary to take data from an application and transform it into another coordinate system before returning it to the originating application. This in essence would allow one to add external functions to an application program, and store those functions at the server. Also, several formats may be necessary for a single application in the system. All these variables are left to the integration system designer to determine. These functions must be written, or designed, in the CAD/CAM CASE Workbench for each set of applications in the integrated system that wish to exchange data. Information obtained by the analyzer about the applications is contained in the database of the workbench and can be utilized to facilitate creation of the transfer functions.

The transfer functions make up a library which resides in the integration server. These functions are used to translate or transform the data which is being transferred within

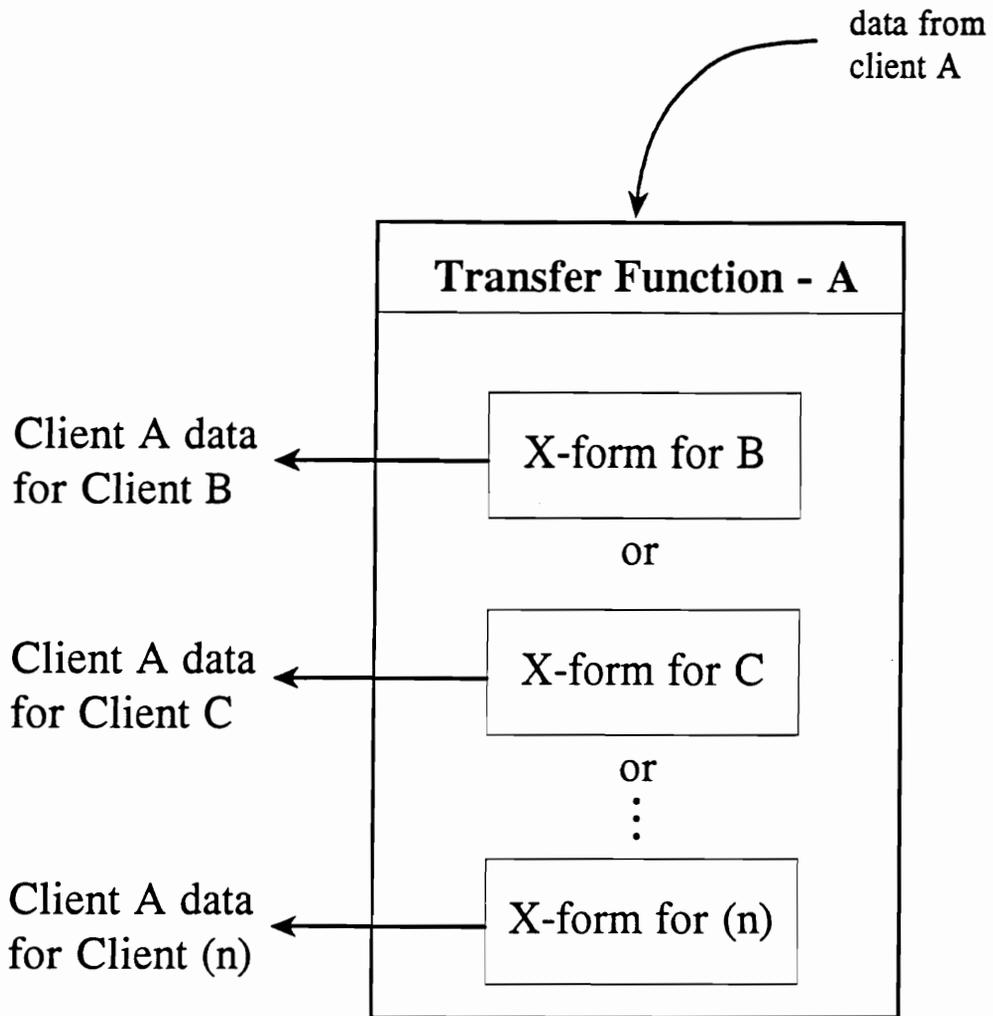


Figure 8: Sample transfer function.

the integration system from one application to another. This is a very important step considering the differences in data structure, ordering, and representation across CAD applications. However, in the event that no transformation or translation of the data format is necessary (data structures in two applications are identical), the integration server will simply apply a new opcode to the message containing the data, and relay it to the second application.

The integration server's design allows it to remain as autonomous as possible, establishing and terminating connections with integration clients in the system. This autonomy allows clients to connect and disconnect without affecting the remaining client applications in the integrated environment. Another benefit which stems from this design is that the server can reside anywhere in the network, and is not obligated to be on a specific machine. The only stipulation is that the clients know where the server is located and use this information when they attempt to connect to the server using the socket they have created for this purpose.

4.3 Communications

For effective communication between the integration client and server, messages sent must be in a form which is predictable and meaningful. A protocol was developed to facilitate the building and resolving of messages being passed in the integrated system. Every message passed contains a data structure called a header which precedes all other data. The header contains the size of the succeeding message and major and minor operation codes (opcodes) to allow for proper evaluation of successive messages by the client or server.

The header data structure is defined as containing the following information:

- 1) *size_in_bytes*: most often used to tell the receiving client how much data to expect after the header. It can also be used to carry flags.
- 2) *maj_opcode*: operation code used to locate the major category of the message.
- 3) *min_opcode*: operation code used to locate the subordinate category of the message.

The major opcode is an integer which denotes a category of message. The integer and corresponding category follow:

- | | |
|---|----------------------|
| 0 | Initialization |
| 1 | Request |
| 2 | Update/Data Transfer |
| 3 | Response |
| 4 | Error Message |

Minor opcodes are used to locate, within the major category, the function or module which will handle the incoming data. This process is discussed in more depth in the section on the distributed integration solution prototype.

5.0 STRUCTURED ANALYSIS AND STRUCTURED DESIGN

As a precursor to the discussion of the creation of CASE tools based on the distributed integration solution, it is necessary to describe the processes of software development. The steps which traditionally comprise the software development and life cycle are shown in Figure 9. These five steps include:

- 1) Analysis and specification of requirements
- 2) Design
- 3) Implementation/Coding
- 4) Testing
- 5) Maintenance

In this section we will treat the issues of Requirements Analysis and Design as applied to the distributed integration solution. The methodologies which correspond to those used by the CASE tools employed in this research to aid in the completion of the tasks above are structured analysis and structured design.

5.1 Structured Analysis and Requirements Specification

Structured analysis was developed by Edward Yourdon and Tom DeMarco to provide a method for focusing on an application's data flow, rather than its control flow. The goal is to produce a graphical structured specification of the application. For the purposes of the distributed integration solution, a structured specification was created using the following tools:

- 1) data flow diagrams: used during analysis to define the problem components and the data transferred among them. It is a graphical depiction of the different data items in a system and their movement.

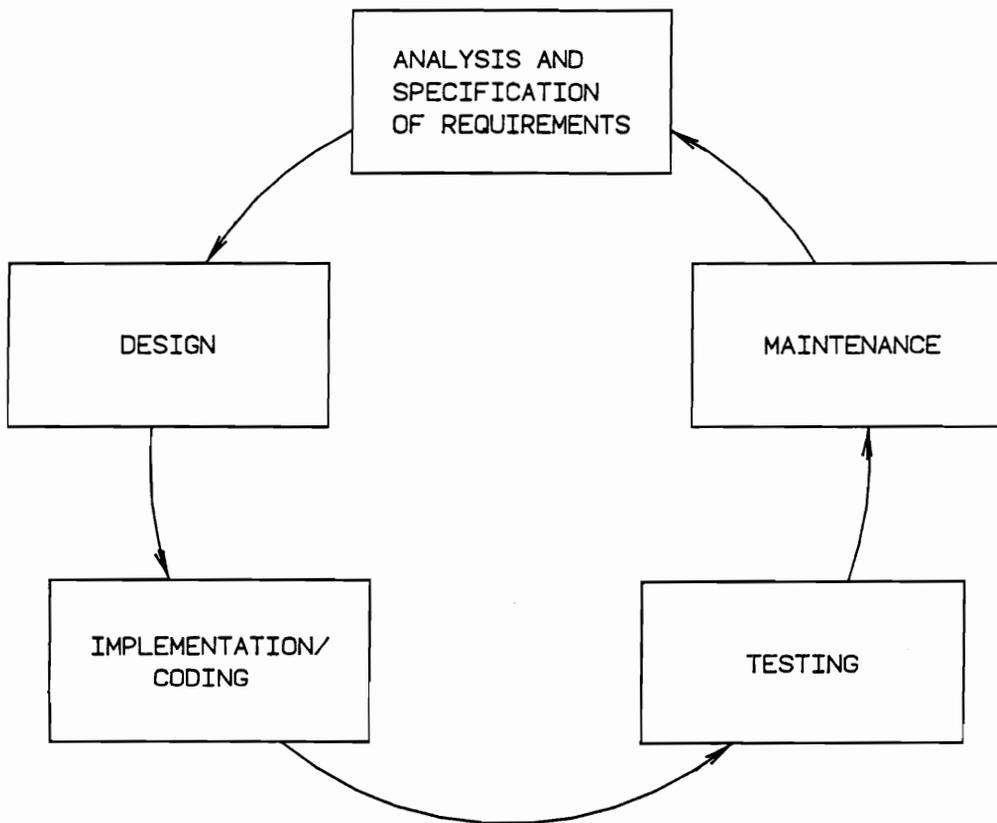


Figure 9: Software development life-cycle.

- 2) data dictionary: a catalog of all data items found in the data flow diagrams.
- 3) process specifications: also called mini-specs they document data transformations occurring in data flow diagrams. Decision tables, decision trees, structured English, and pseudocode are all process specification techniques.

The first step in the creation of the structured specification was the creation of a data flow diagram for the entire integration system. Data flow diagrams offer a top-down view of the system to be created from a data perspective. In a data flow diagram, data elements flow from process node to process node, where they are transformed. There is no notion of control flow. Because of this, data flow diagrams best depict a system as viewed by the end user.

Data flow diagrams consist of four graphical components as shown in Figure 10. The bottom-level process in a data flow diagram has a process specification associated with it. The process specification describes the transformation of the data input to the process into output. Another important element of the data flow diagram is the data dictionary. Also shown in the figure are a few of the notations employed in the data dictionary which are used to interpret the data flow diagrams. A data dictionary is a reference of all the data elements found in a data flow diagram. All attributes of a particular piece of data can be found in the data dictionary.

In summary, the important characteristics of the data flow diagram are:

- 1) graphical specifications of requirements
- 2) hierarchical and multi-level in nature
- 3) emphasis on data flow instead of control flow
- 4) specification of software requirements, not software design

Data Dictionary Entry Operators

Operator	Example	Definition
+	a+b	a together with b
[]	[a/b]	select either a or b
**	*comment*	comment

Data Flow Diagram Components

Symbol	Name	Function
→	Data flows	Data structures input to and output from the process
○	Process nodes	Transform incoming data flows to outgoing data flows
□	Data sources + sinks	External originators and receivers of data flows
— —	Data stores	Repositories which allow addition and retrieval of data

Figure 10: Components of a data flow diagram.

5.2 Structured Design

Structured analysis and specification is a first step to the achievement of a structured design. Structured design articulates a software system's internal architecture, while structured analysis methodologies, such as data flow diagrams, emphasize a system's external or user's view. It is a process whereby system requirements are transformed into a plan for implementing the requirement. Figure 11 shows the process of structured design as defined by Yourdon [Page88]. Notice that structured analysis is a necessary first step.

The CASE workbench aids in the transition from structured analysis to structured design by supplying aids to transform analysis requirements into design specifications for implementation. These design specifications are usually in the form of a structure chart. The three steps used to create structure charts, as defined by Page-Jones, [Page88] are:

- 1) Break the system into similar units using transaction analysis.
- 2) Convert each unit into a structure chart using transform analysis.
- 3) Construct an overall system implementation from the separate units.

Transaction analysis identifies the transaction types of a system and uses them as the units of design. Transaction types are identified on the data flow model of the system by studying the discrete event types that drive the system.

Transformation analysis is the strategy used to convert each unit of the data flow diagram, which was isolated in the transaction analysis, into a structure chart. This method consists of the following five steps:

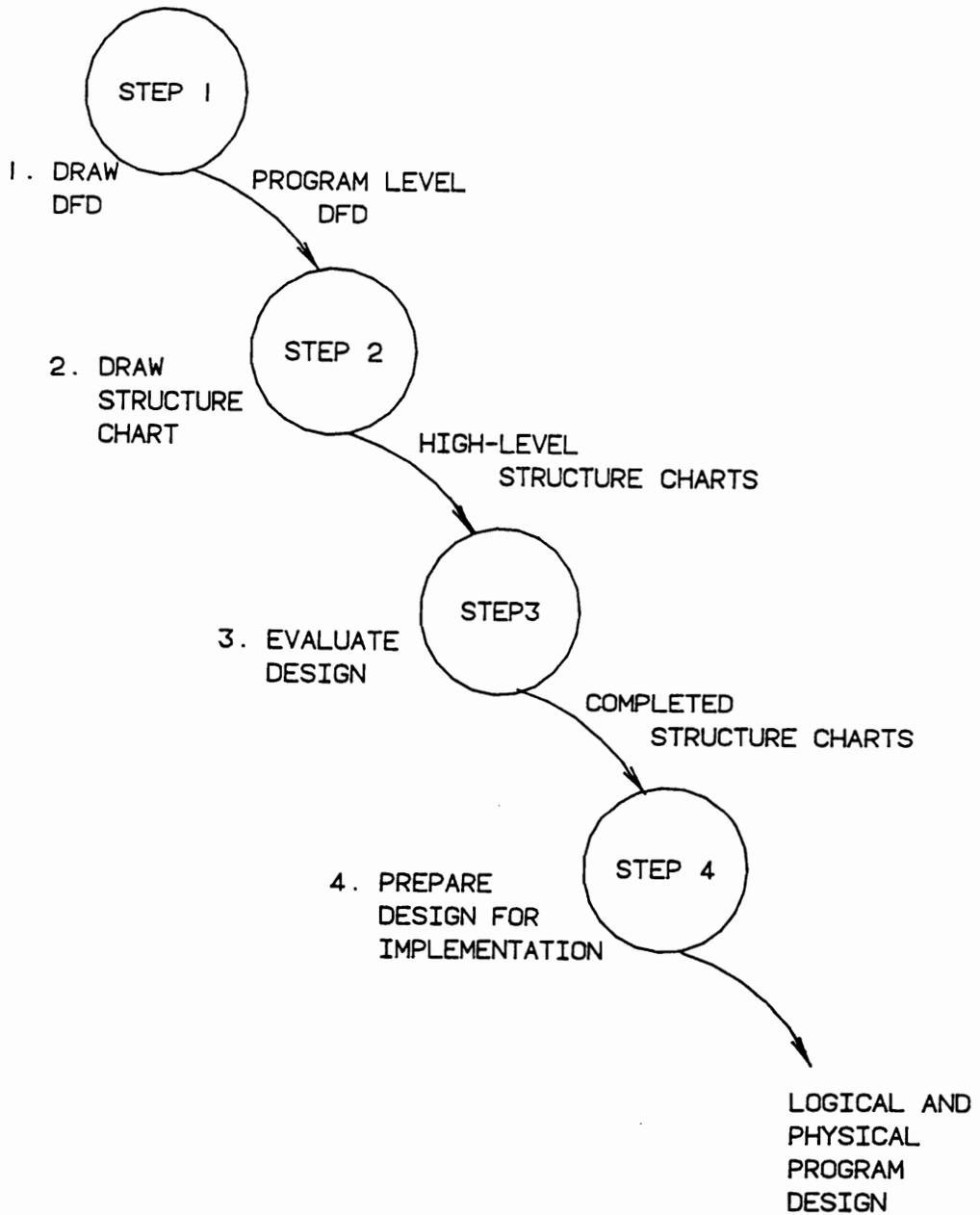


Figure 11: The Yourdon structured design process [Page88].

- 1) Drawing a DFD (data flow diagram) of a transaction type.
- 2) Finding the central functions of the DFD.
- 3) Converting the DFD into a first draft of the structure chart.
- 4) Refining the DFD.
- 5) Verifying the structure chart with respect to the original DFD.

The process of transforming data flow diagrams representing transactions into structure charts is not algorithmic. Instead, transform analysis is a strategy which produces a rough draft of the system, requiring several iterations to perfect.

When completed, structure charts represent the software system in a hierarchical and modular fashion. Data transactions between modules, called data couples, are represented as well. In the figure which defines structured design according to Yourdon, Steps 1 and 2 refer to requirements analysis and structured design. Step 3 is designed to evaluate the quality of the design. This evaluation is made based on data coupling and cohesion. Data coupling is a measure of the complexity of connections between modules. In this respect, the simpler the connections, the better. Cohesion is a measure of the strength of functional relationships within a module. Step 4 is an iterative process whereby modules are divided until the lowest level can be used for implementation of the design.

Structure charts show only the overall structure of a system, with very little procedural detail. In order to enable the programmer to make the transition from structure chart to actual code, some sort of procedural information must be given about each module.

There are two possible methods by which this can be accomplished:

- 1) module specifications (m-spec): give input and output expected from the module and the function the module is expected to perform. No particular structural

information pertaining to the code is necessarily specified, although the possibility exists for actual source code to be placed in the m-spec, which is then embedded in the source code of the system when complete.

- 2) specification by pseudocode: specifies how the module should be programmed by using an informal language similar to structured English.

Figure 12 shows a sample structure chart and defines commonly used notations.

Using module specifications, a system designer is able to incorporate implementation specific details using C language syntax. Attributes which further describe these details are then added to the data dictionary. The C Source Builder is then used to generate the C-source code files from the structured design model (structure charts and module specifications). The resulting source code is compiled and linked to produce an executable system. The compile/link process is done outside of the CASE workbench environment.

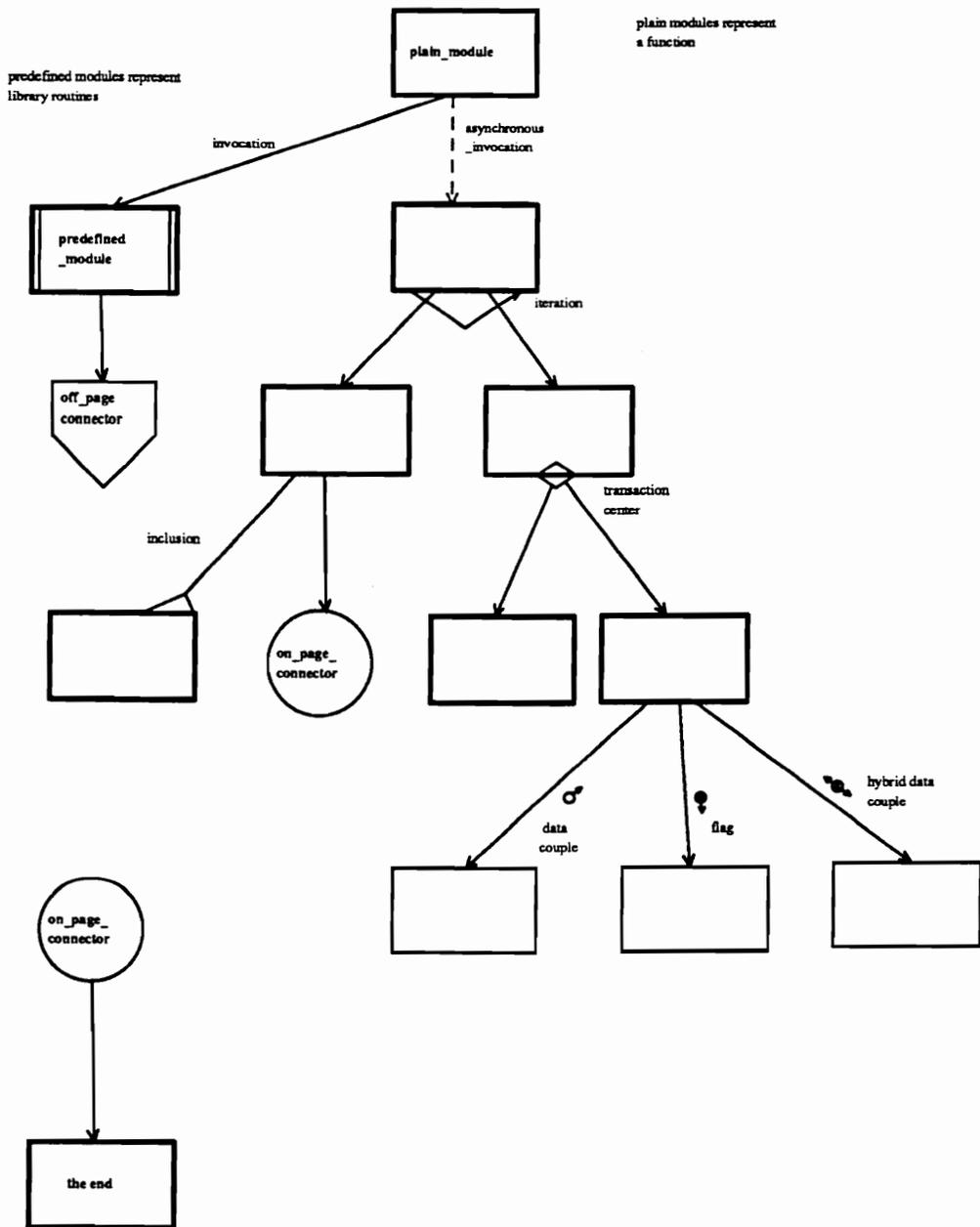


Figure 12: An example showing structure chart components.

6.0 INTEGRATION TOOLKIT

CASE tools are tools which leverage the requirements and design specifications phases of the software development cycle, or that generate code. In this respect, the Integration Toolkit is a CASE tool because it aids the integration system designer in generating an integrated system.

The CASE workbench by CADRE is essentially converted into what is defined in this research as the CAD/CAM CASE Workbench by using specific components of *Teamwork* (the CADRE CASE product) in conjunction with the Integration Toolkit. The Integration Toolkit will appear to the user as a part of *Teamwork* because it will be integrated into the generic CASE workbench using the integrated product support environment (IPSE) supplied by CADRE. The following *Teamwork* components are used in addition to the Integration Toolkit to form the CAD/CAM CASE Workbench:

Teamwork/SA Based on Yourdon-DeMarco methodologies, this tool is an editor for creating and editing data flow diagrams and mini-specs (process specifications). It builds input and output lists for every process node in the data flow diagram and allows the developer to attach code to the individual process node mini-specs. This tool aids the integration system designer in the completion of the structured analysis and requirements stage of the software development life-cycle.

Teamwork/SD An implementation of the Yourdon-Constantine structured design methodology. The structured design module provides for the definition of software modules in a top-down structured chart. It also builds input and output lists, based on data couples, for each module in the structure chart.

Developers can incorporate code in each module's m-spec (module specification) which is used by the source code generator. This tool helps the integration system designer complete the second stage of the software development life-cycle.

- Teamwork/IPSE** The Integrated Product Support Environment (IPSE) is a tool which serves as a framework into which other tools fit. It enables tools and toolkits from various sources to be joined in a common manner. Data sharing between components of the workbench is enabled. The IPSE provides a common user interface for tools.
- C-Source Builder** Using the models created during the structured design phase, the C-Source Builder automatically generates C-based source files. The source code generator constructs external (global) and function definitions from information found in module specifications. The system designer can chose to place implementation details in the module specifications and data dictionary to ensure complete code generation. In fact, it is possible to place C code directly in the m-spec which will later be embedded into the final product. Source code files built by the C-Source Builder can be compiled and linked upon completion. The source builder enables easy editing of module specifications and data dictionaries, as well as the ability to store source code for common or repeated use. This tool leverages the third step in the software development life-cycle; implementation and coding.

Figure 13 shows how the Integration Toolkit option appears within the *Teamwork* framework. From the figure it is evident that the Integration Toolkit, as it appears to the user, consists of the following menu selections:

CHIEFTAIN BOND
50% COTTON FIBER

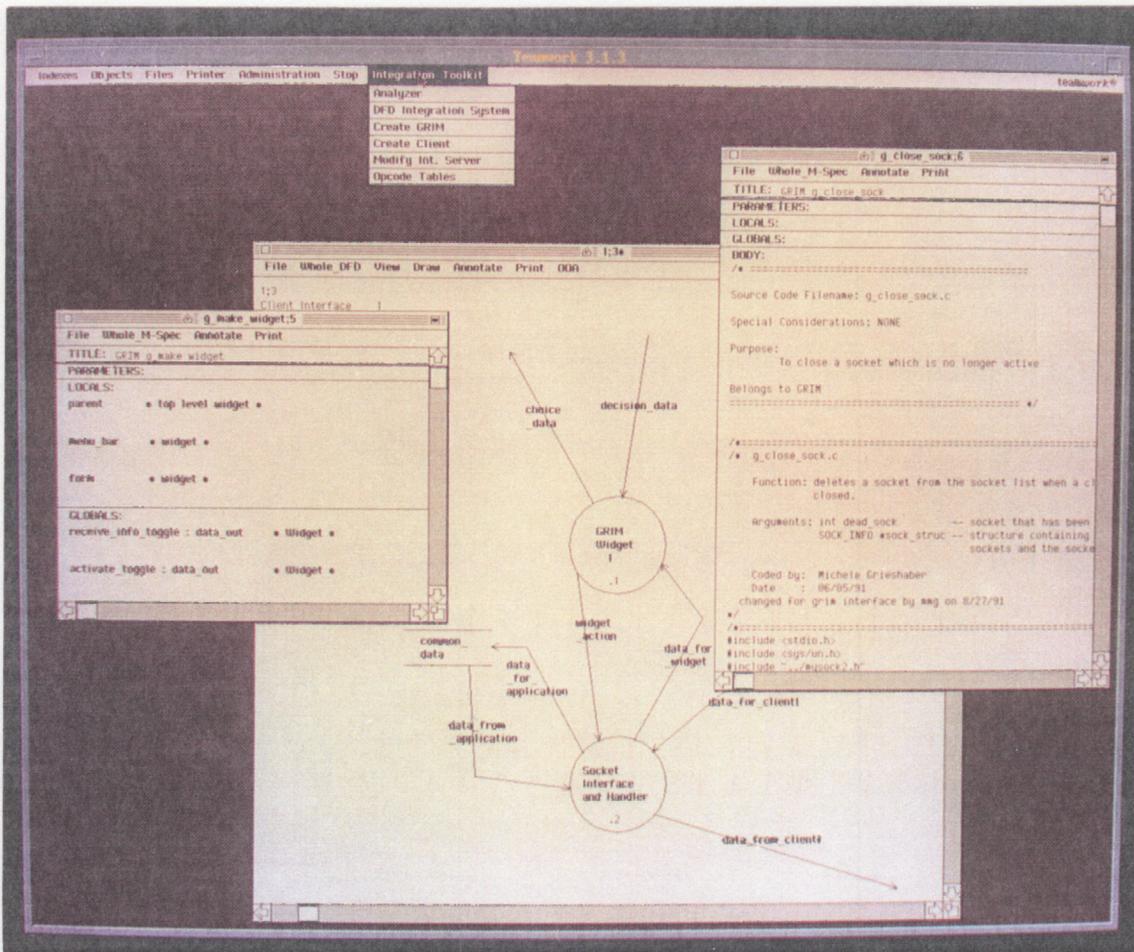


Figure 13: Integration toolkit in CASE environment.

- Analyzer
- DFD Integration System
- Create GRIM
- Create Client
- Modify Integration Server
- Opcode Tables

The analyzer parses application source code and extracts information on data structure, I/O capability, subroutines, etc. which are pertinent to the task of the integration system designer. This information is stored in the workbench database.

The data flow diagrams represent the integration system as a whole and demonstrate each component's role from a data perspective. These diagrams will be used by the integration system designer to map the flow of any new data that he may need to add to the system. A base set of diagrams is provided which defines the minimum data flow in a valid system. Process specifications, or mini-specs, are produced for the lowest-level process nodes in the data flow diagrams. These specifications describe how the input data are transformed into output. A data dictionary defines the data elements found in the data flow diagrams. It may also be used to store data definitions generated as output from the analyzer. This aids the integration system designer in incorporating application data into the data flow diagrams and structure charts.

For each of the menu selections designating a component of the integrated system, a set of structure charts is provided which represents the base system. The term "base" implies the minimum configuration of client and server which can be implemented effectively. In order to expand the base system, the integration system designer will

need to add modules to the client which are specifically geared to interface with the CAD application. Transfer functions corresponding to the client application will be appended to the integration server. In effect, the integration system designer will add modules and m-specs to the base diagram to tailor the system to fit the new applications and his needs. Module specifications exist for each module in the integrated system. In the case of the base system, they contain actual source code used in system creation. Application specific modules are indicated in the m-specs and need to be tailored to the function or application by the integration system designer. Such m-specs could contain source code, pseudocode, or functional descriptions. The m-specs will be used in conjunction with the C-Source Builder to generate compilable code.

Opcodes tables are used by the integration system designer to trace the flow of data through the integrated system based on the communication protocol header which precedes all data. These tables are graphical representations of the method by which the modules responsible for handling the incoming signal will determine which function processes the signal. Functions are placed in the table to indicate their positions with respect to header data. The table has columns representing the five major opcode categories, and rows representing the minor opcodes. These tables will be used in the following discussions to illustrate the communication mechanism in the integrated system.

In the following sections, a description of the DFD's is followed by detailed discussions of each component's structure charts. The structure charts and data flow diagrams accessed in the Integration Toolkit will be accompanied by module specifications (used for code generation), process specifications, and a data dictionary.

6.1 Data Flow Perspective of the Distributed Integration Solution

In order to describe the flow of information through the distributed integration system, the product of the structured analysis phase of development, the data flow diagram, will be used. Definitions of the data flows discussed in this section may be found in the data dictionary included in Appendix A. The data circulating in the distributed integration system is in actuality a request, response, or initialization information. It is difficult to specify much of the data in any terms other than conceptual. For this reason, actual data flow names may not translate directly to the structure chart representations. Instead, the concept of the data flow is represented by using terms descriptive of the data and its function in lieu of actual variable names.

The data flow diagrams developed for incorporation into the Integration Toolkit are found in Appendix B. The process specifications which correspond to the lowest-level process nodes in the DFD's are located in the same appendix following the diagram they describe. There are several tools used to write process specifications. These methods include structured English, decision tables, pre/post conditions, flowcharts, etc. Many organizations tend to utilize only one tool to write process specifications. According to Yourdon [Your89], the use of a single tool is a mistake. Instead, he contends that a combination of the tools mentioned above should be used. The decision of which tool to use is based on user preference, programmer preference, and the idiosyncratic nature of the various processes. The process specifications located in Appendix B use decision tables, structured English, and pre/post conditions to define the process nodes of the data flow diagrams. The reader may find it helpful to refer to the data dictionary and process specifications when deciphering the data flow diagrams.

We will now discuss a few of the data flow diagrams. In the course of describing the diagrams, the analysis process which went into the design of the distributed integration solution will be described. The first diagram one must consider is the context diagram, shown in Figure 14. The context diagram is defined as the top-level of a hierarchical set of data flow diagrams. It represents the entire system in terms of a single process, shown as bubble 0. The diagram is used to delineate the scope of the analysis and define the system in terms of inputs and outputs. The context diagram, in conjunction with the data flow diagrams derived from it, enables the integration system designer to identify the major transactions of a system in terms of inputs and outputs. In order to determine a solution to the problem of integration of CAD/CAM applications in a network environment, the scope of the system in terms of inputs and outputs must be defined. The context diagram shown in Figure 14 demonstrates the external influences on the integration system. The user is considered a terminator (an external source of data input and/or output) because based on the presence of choices, he will make decisions and transmit them to the integration system. In an effort to retain control of data exchanges in the integration system, data requests are the only method by which to effect data exchanges. This means that it is not possible to send data to a second application in the integrated system unless it has been requested by a user at the second application. This eliminates the possibility that a user could be creating a model at an application and suddenly have it overwritten by data that has been sent into it from another application. Furthermore, it does not make sense for a user to send data to another application unless he will be using that data himself. Given this scenario, the constraint that all data transactions must be requested from the application which will receive the data is valid. In addition, Figure 14 shows the applications as terminators. The applications are considered as external to the integration system; however, they do interact with it by sending and receiving application data. By designing the

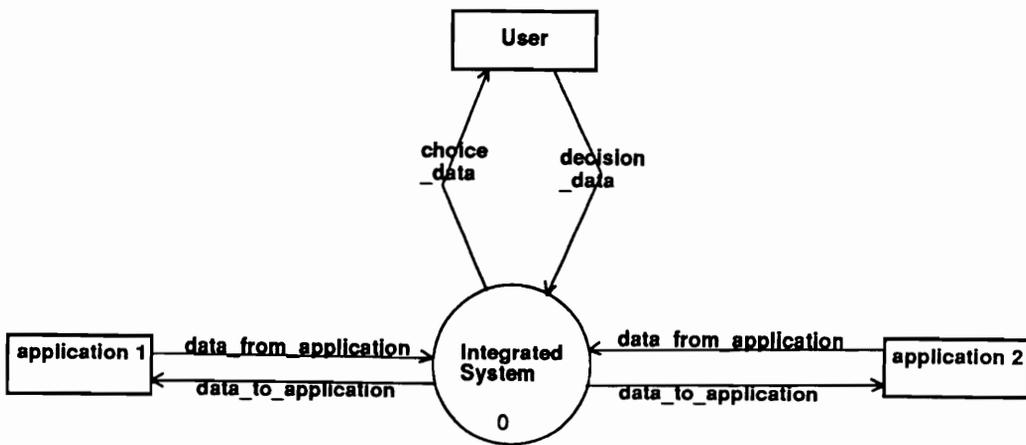


Figure 14: Context Diagram of the integration system.

applications as external, it is easier to isolate applications from changes applied to other applications in the integrated system. All necessary modifications will be made internal to the integration system instead of to the application itself. It is important to mention here that only two applications have been used in these diagrams in order to simplify the data model. However, the rules developed for two applications can be extended to cover n applications.

The next data flow diagram described is one level below the context diagram, data flow diagram (DFD) 0. This diagram is shown in Figure 15. This diagram shows the integration system broken down into the integration server and client interfaces to the CAD/CAM applications. The integration server was designed such that the users of the applications in the integrated system can be unaware of its function or existence. This means that the integration server will be able to run as a background process on one of the machines in the network, allowing clients to connect to and disconnect from the server at any time. By hiding the integration server, instead of making the application users responsible for initiating contact between the server and their applications, the proper integration client/server relationship is retained. The client interfaces represent the portion of the integration clients which will act as an interface between the CAD/CAM application, the user, and the integration server. The interfaces and even the integration server help to isolate the applications in the integration system from modifications to applications with which they exchange data. The reason this isolation is possible, is that transfer functions at the integration server can be updated to reflect changes in data being sent from a modified application. On the other hand, if an application is modified such that it needs to receive more or new types of data from a second application already defined in the system, the second application must be

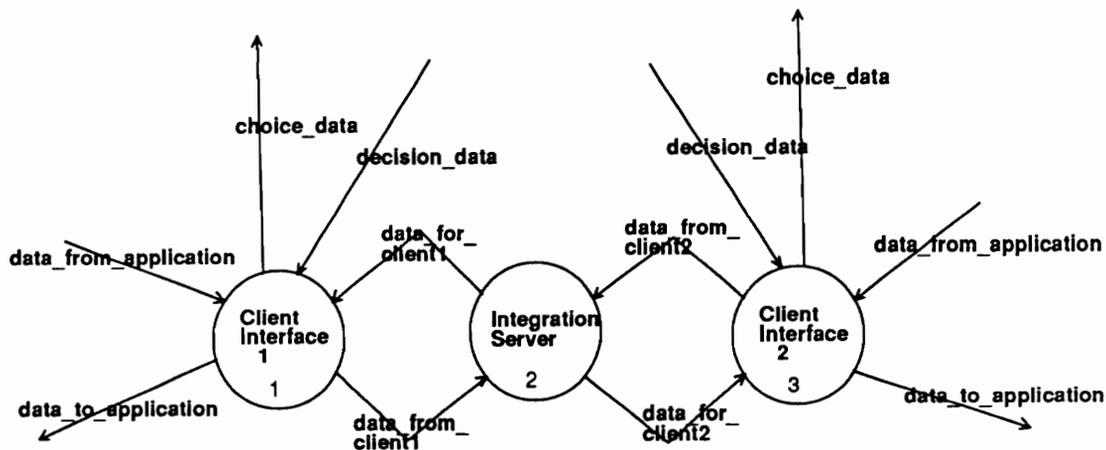


Figure 15: Data flow diagram of the integration server and clients - DFD 0.

modified at the interface level as well as modifying the transfer function. This still minimizes the amount of change to applications in the overall system.

Figure 16 shows DFD 1, which represents the client interface in terms of the GRIM (user interface to the integration system) and the socket interface (between application, GRIM, and integration server). This diagram was developed under the assumption that source code for the application is available. If the source code can be obtained, the application can be modified to share a common buffer with the socket interface into which data from and for the application are placed. The data which are sent to the socket interface from the integration server can either be destined for the application, the GRIM or the socket interface. Data destined for the application originated from another application in the integration system (as a result of a request generated by the user at the receiving application). The data have been transformed or translated by the transfer functions of the integration server into a format acceptable to the socket interface which receives the data and places them in the common buffer for access by the application. Data destined for the GRIM, such as a list of clients in the system from which data may be requested, are received by the socket interface and relayed to the GRIM. The socket interface, often referred to as the AP/SOCK interface, and the GRIM run in parallel and communicate over asynchronous sockets. This configuration was chosen because the task of integrating a user interface to the integration system directly into the CAD/CAM application would mandate the utilization of source code and would be time consuming to effect. Instead, the approach of designing a generic interface which will work with all applications regardless of source code availability was taken. This means that the GRIM manages both user input and signal reception from the AP/SOCK. By separating the GRIM and the client application (AP/SOCK interface and CAD/CAM application), the application is able to execute without

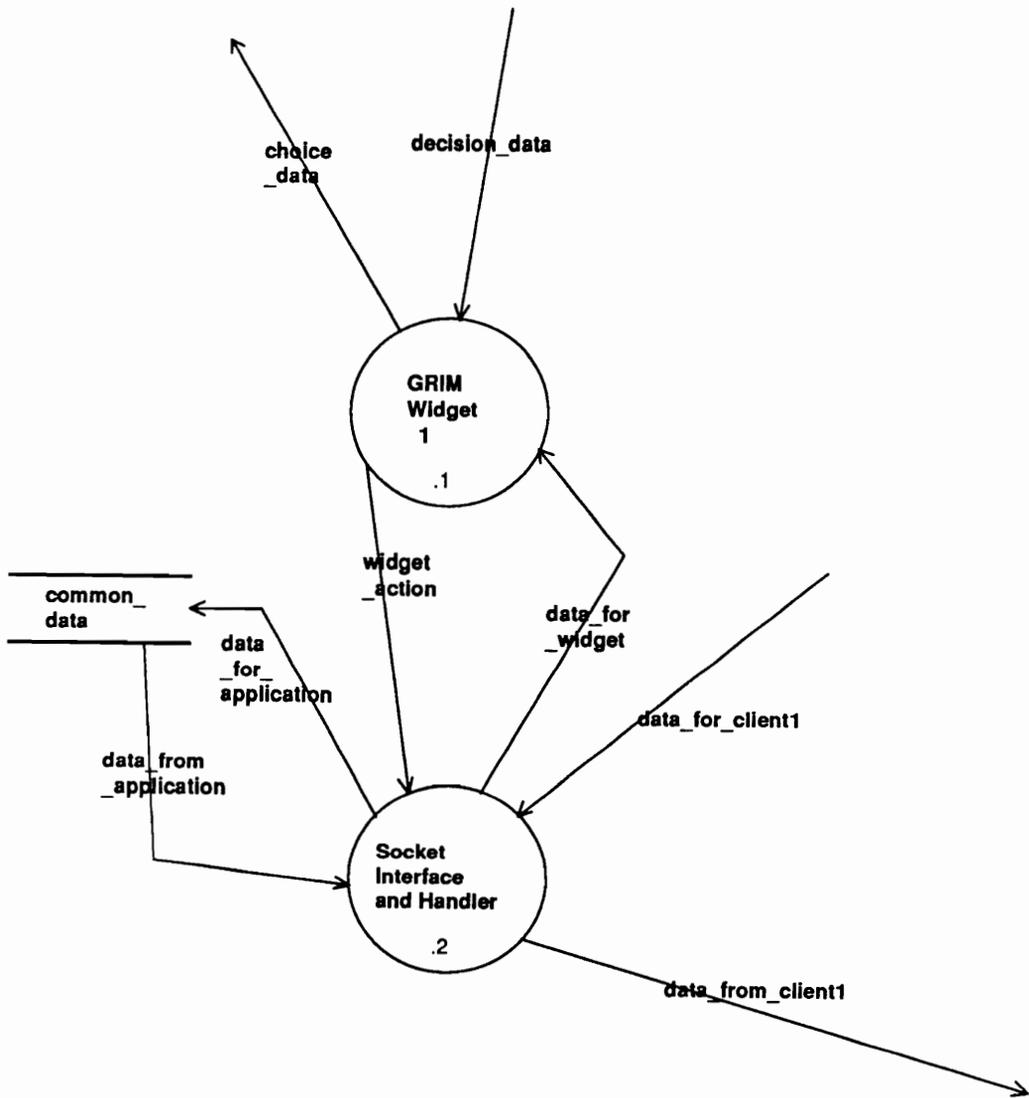


Figure 16: Data flow diagram of the client interface - DFD 1.

managing the user interface. When a sequence of user interactions at the GRIM triggers a request event, the GRIM sends a signal to the AP/SOCK interface. The occurrence of a signal at the AP/SOCK will suspend the application's execution. Once the signal has been evaluated and handled, for instance when the user request is forwarded to the integration server for further evaluation, control returns to the application at the point it was suspended. Data which are destined for the AP/SOCK interface include requests for the applications name. Requests of this sort are generally used by the integration server for management purposes. For example, the integration server keeps a list of all client applications currently connected to the integration system and is able to cross-reference application name with the socket descriptor which defines the integration server/client communication path.

Figure 17 shows DFD 2.1 which describes how the integration server handles data from a client in the integration system (in this case the client is called client 1). As was mentioned above, the integration server is able to determine the socket descriptor at the server which communicates with a client given the client's name. The server also sets up a data structure containing exchange relations. This data structure is created during the initialization of the integration server, and enables the server to determine which clients can request data and which can send data. By using the relation data structure in conjunction with the current list of connected clients, the integration server can issue a list of clients from whom a newly connected client can request data. The relations file is read at initialization of the server in order to avoid rereading the file every time a new client requests connection with the server. Data sent to the server from a client in the integrated system can be one of three types: data generated by the user at the GRIM which needs to be forwarded to another client in the system, data necessary for

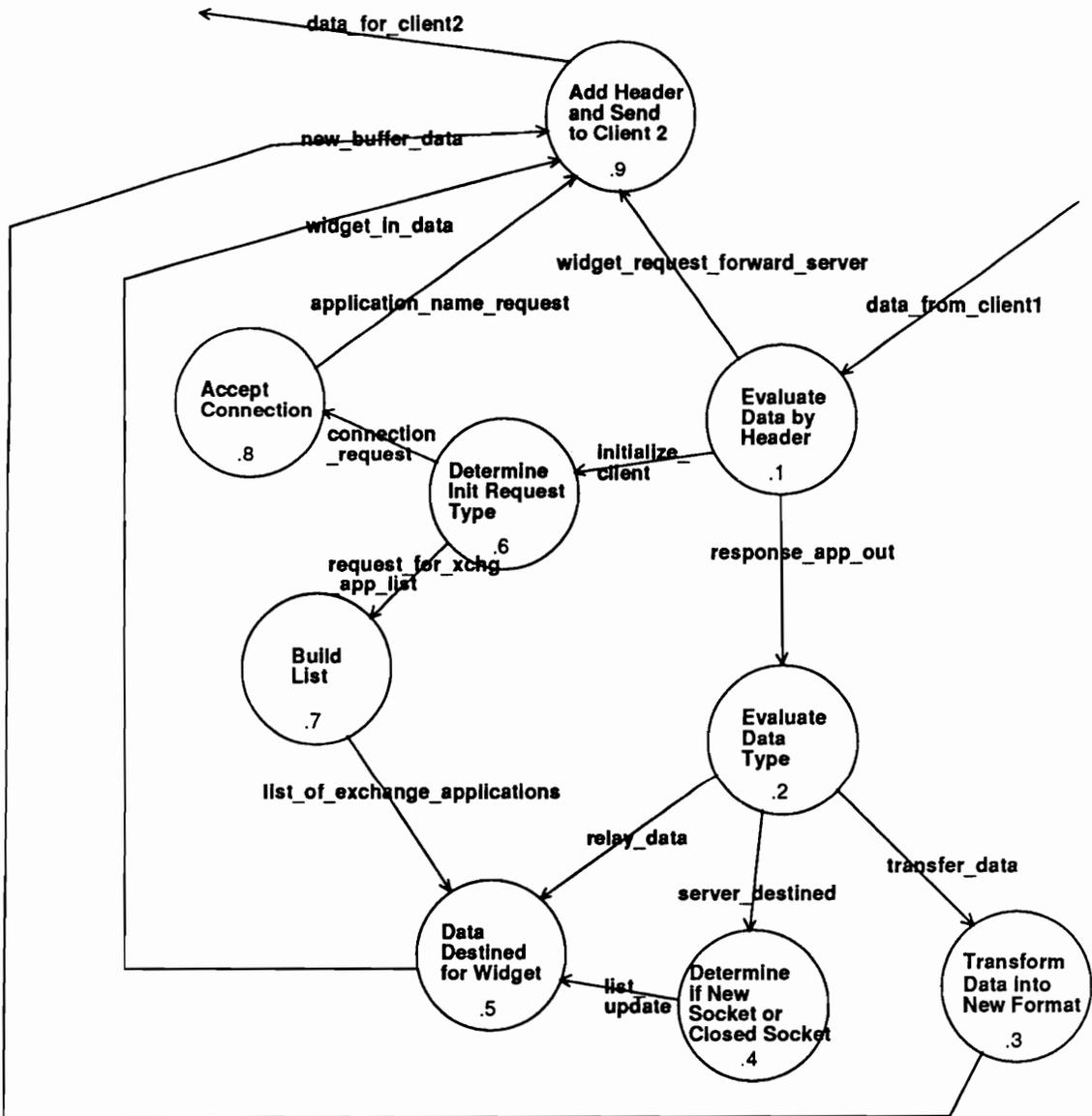


Figure 17: Data flow diagram of the integration server - DFD 2.1.

connection or initialization the client, or data which must be treated by the integration server. Figure 17 labels these data items as `widget_request_forward_server`, `initialize_client`, and `response_app_out`, respectively. Data which needs attention from the server includes data which must be sent to a transfer function in the server before being sent to another client in the system, data which represents the termination of a client in the system, or data which needs no treatment from the server and simply needs to be relayed to another client in the system. These three types of data are shown in Figure 17 as `transfer_data`, `server_destined`, and `relay_data`, respectively. Before sending data on to a second client in the integration system, the integration server prepends a header to the data which will enable the message to be evaluated correctly.

The data flow diagrams discussed in this chapter have been used to describe the analysis and requirements phases which were used to create a solution for integration in a network environment. More detail on the data flow diagrams which describe the system can be found in Appendix B.

6.2 The Distributed integration solution in Structure Charts

The structure charts, which an integration system designer will modify to produce an integration system, have been split into the GRIM, the client application, and the integration server. Since the GRIM and the client application run independently of one another, they are developed separately. Hence they have separate structure charts and menu items in the Integration Toolkit. A description of each set of charts is contained in this section. The structure charts for each component and their module specifications appear in Appendices C, D, and E. Appendix F contains, among other things, the module specifications for commonly used utility routines. The utilities can

also be used by more than one component in the integrated system and are grouped together for reuse. Appendix F also contains the header "mysock.h" which defines common structures used in all components of the integration system. Such definitions include the header and the socket structure. The following description of the structure charts is more or less on a conceptual level. For details of implementation, it will be necessary to read the m-specs for the set of structure charts described. The m-specs contain actual C source code which will be used to build the integration system. Comments interspersed throughout the m-specs will enable those persons interested in implementation detail to understand the function of each module. Please note that during the discussion of structure charts, module names are bold and data couples are italicized.

6.2.1 GRIM Structure Charts

The GRIM widget is essentially a generic element of the integration system before a client application connects to it and assumes ownership. This is one reason why a separate group of structure charts has been created for the GRIM.

Every client application in the integrated system must have its own GRIM. The method by which this is ensured is by defining identical pathnames, which correspond to a unique UNIX socket identifier, in both the GRIM and client application. To do this, the integration system designer must modify the top module in the client application and several modules in the GRIM where pathname is defined. Modifying a module implies that its module specification is edited or changed in some way. When this has been done for the GRIM, its source code may be compiled and linked. The GRIM executable should be given a unique name which allows it to be associated with its owning client.

The Integration Toolkit essentially uses this method to produce a GRIM widget for a given client application. Since only pathname needs to be modified, the module specifications containing source code for the widget will need to be updated. The resulting m-spec will be used in conjunction with the CADRE C-Source Builder to produce the executable widget interface. Selection of the "Create GRIM" menu in the Integration Toolkit will cause the display of an information panel with the following message:

"In order to complete the task of creating a new GGraphical Interface Manager, three m-spec (module specifications) objects must be updated:

- 1) GRIM
- 2) g_close_sock
- 3) g_make_widget

When you click OK, the m-specs for these modules will appear in three separate windows [in the *Teamwork* environment]. You should correct the value of the <pathname> constant in each of the three files. This will be the first line of source code in each m-spec. After correcting this line [in each object], close the object file and run the C-Source Builder to generate the C source for the GRIM application."

If the integration system designer selects the OK button, three objects are opened in the CADRE environment. Each of these object files needs the pathname definition edited. After the new pathname has been added, the object is saved and exited. This process is repeated (edit pathname, save and exit file) until all necessary files have been modified. The C-Source Builder is then invoked to produce an executable with a unique name.

The client application is subsequently modified to accommodate a CAD/CAM application which is targeted for integration into the system. When the client application has been completed and an executable has been made, a UNIX script file is

used to invoke the GRIM and the client application in such a manner that they run independently and cooperatively.

For the sake of completeness the GRIM structure charts have been included in the Integration Toolkit. These GRIM structure charts and corresponding module specifications can be located in Appendix C. A description of the structure charts and their modules follows.

GRIM is the main module of the GRIM widget. Its first task is to open a socket on which to listen for connection requests from the client application. In order to uniquely define the socket in the UNIX domain, the pathname defined at the top of this module must be modified. The resulting pathname must be known by the client application which owns the widget in order for communication to occur. The socket, *Sock*, is created as the listening socket and is known globally within the GRIM program.

The next module invoked from **GRIM** is called **g_make_widget**. This is the module responsible for creating and displaying the basic widget which will be used in conjunction with the client application. The basic GRIM widget is shown in Figure 7. All code for the modules called from **g_make_widget** is included in the same m-spec. Following the structure charts, we can see that the main window widget is created and a menu bar widget is added to it. A pull-down menu called **ACTION** is established as part of the menu bar. The creation of the reset and exit choices for the pull down is shown in the module **MakeMenuBar** as **CreateMenuButtons**. The reset and exit options have callback functions associated with them which define the required action taken when either is selected by the user. It is necessary to note here that the callbacks are drawn in the GRIM's structure charts as being invoked asynchronously. This is

because they could be called at any time based solely on user input. In the remaining discussion, callback functions are not explicitly stated as being present. Instead the description of a widget's action should imply that a callback is responsible for executing that function when necessary.

As can be seen in **MakeSelectionBox**, the selection box contains several possible sub-widgets, two of which are used here, as shown in Figure 7. They are the selection list widget and a selection dialog. The selection dialog is not shown in the structure charts, but its purpose is to display the name of the last-chosen selection list item. Other widgets normally used with the selection box widget are not necessary and are therefore unmanaged. In **MakeOtherStuff**, a framed form widget containing three toggle buttons is added to the selection box widget.

Once the basic widget has been created and realized, a work procedure is checked. A work procedure forces XtMainLoop to branch to a function called **g_select_loop** when there are no input events on the GRIM widget. The **g_select_loop** function enables the GRIM to check for signals arriving from the client application. In order to do this, a *read_mask* must be defined. The mask is essentially a filter which is set to contain all socket descriptors on which the GRIM expects to receive signals. Before the client application connects to the GRIM, the only socket descriptor in the mask is the listening socket, *Sock*. Once connected, the client application's socket (defined at the acceptance of the connection) is included in the readmask. The *select* function (from the BSD socket library subroutines) checks all sockets in *read_mask*. If a signal is detected, it modifies the readmask to contain only the socket descriptor on which the signal occurred. If no signals are present, the work procedure returns control to XtMainLoop.

If a signal is present, however, the readmask is passed to `g_eval_sel`. This module determines on which socket the signal occurred by comparing the socket data structure with the readmask. If the signal came in on the listening socket, the client application is attempting to connect to the GRIM. In this case, the connection is accepted and a message requesting the client application's name is sent. This process occurs only during initialization of the GRIM. If the signal came in on the socket connected to the client application, a header is read from the signal. If there is no data on the socket, the signal was meant to indicate that the client application terminated. As a result, the socket is closed and the widget self-terminates since the client it served is no longer active in the integrated system. If there are header data on the socket, the header is sent to `g_sw_op` where it is resolved. In the GRIM structure charts, the term *read_sock* refers to the socket on which the signal occurred. It is called *read_sock* because the data present on the socket must be read in order to determine its contents. Table 1 illustrates how the header is resolved based on major and minor operation codes contained in the header. A description of each module handling signal data follows:

g_add_name A header with a major opcode of 0 and minor opcode of 0 is handled by `add_name`. `Add_name` reads the name of the client application from the socket and creates a motif-based string from it. The string is then added to the widget to show the client to which the widget is dedicated.

g_add_to_list This module builds the client selection list one item at a time. The number of items in the list is sent in the `size_in_bytes` portion of the header structure. All client names which comprise

Table 1: GRIM opcode table

	major opcode 0	major opcode 1	major opcode 2	major opcode 3	major opcode 4
minor opcode 0	g_add_name		g_get_new_list		
minor opcode 1					
minor opcode 2	g_add_to_list				
minor opcode 3			g_make_attrib _list		

the list are read from the socket and placed in the widget for display.

g_get_new_list This module uses the `size_in_bytes` portion of the header as a flag to determine whether to add or delete a client from the selection list. Based on the flag, the client name read from the socket is added to or deleted from the existing selection list.

g_make_attr_list This function is called when the list items of another client's attribute list must be displayed. The name of the client who sent the list is read. A bulletin board widget is then created which will be appended to the selection box widget of the basic widget created earlier.

The bulletin board widget displaying an attribute list is shown in Figure 18. As shown in the figure, the new widget consists of an OK button, a cancel button, and a list of attributes. By choosing a list item and the OK button, a request for that data item is sent to the client from which the attribute list originated. Cancel will exit the bulletin board.

6.2.2 Client Application Structure Charts

A base representation of a client application is contained in the Integration Toolkit. This representation is in the form of structure charts and m-specs included in Appendix D. There are several modules included in the client application structure charts which are application specific. These modules have m-specs which declare the fact that the module must be tailored to fit the CAD/CAM application into the integrated system.

For example, the module `main_ap` is a CAD application which has been modified to be subordinate to the `ap_sock` module. The m-spec corresponding to its module alerts the

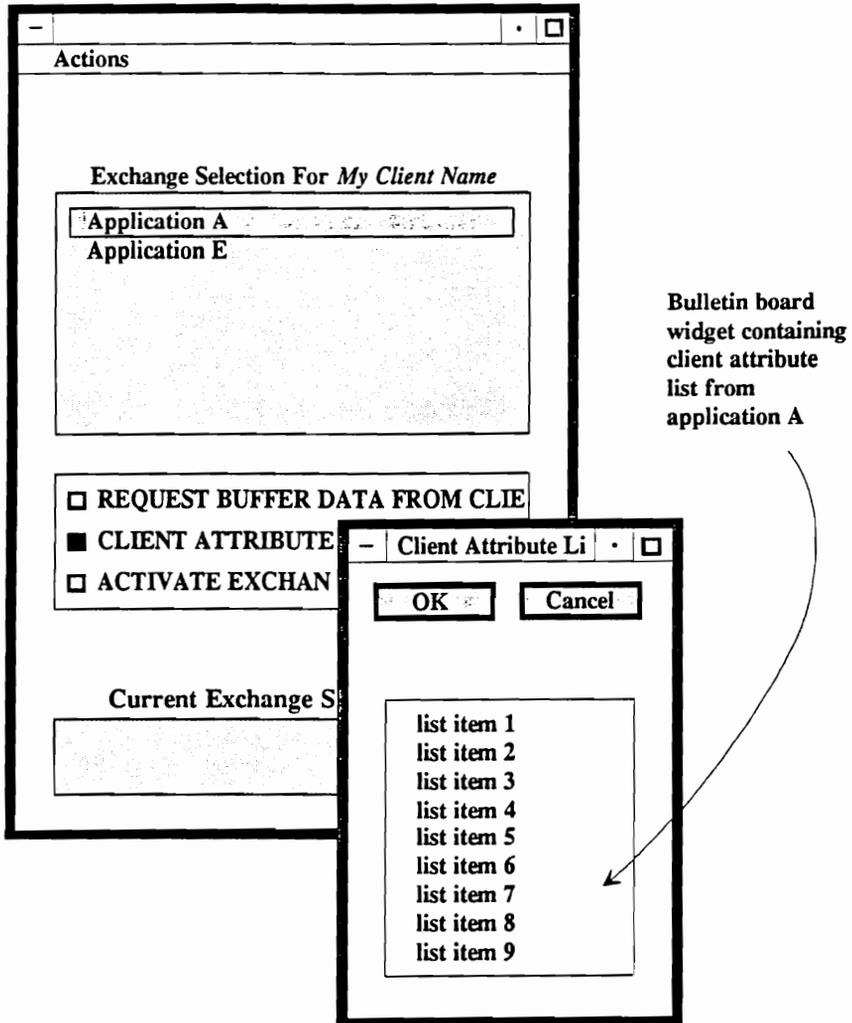


Figure 18: GRIM widget displaying attribute list.

integration system designer to this fact. By identifying the modules where application specific code must be inserted, the integration system designer can easily construct a minimally configured integration system. In other words, more sophisticated data exchanges and messages may be added to the clients and integration server once a thorough understanding of the communication process and system architecture is achieved. This is done by first identifying a data exchange transaction. The integration system designer then designs a module located at the responding client which will supply data, and a module located at the requesting client to receive data. He then creates a transfer function for the integration server which will transform or translate the data from the responder to the requester. The correct headers must be constructed in each module that is sending data. Function names will then be placed in the opcode table such that they will handle the data meant for them. An example of such an opcode table, Table 1, was seen in the previous section. These tables are graphical representations of the function used to determine which module to call based on header information. Opcode tables are included in the Integration Toolkit to aid an integration system designer in tracking the flow of data through the system.

An important element in client application modules is the use of two global variables: `ACTIVE_WIDGET` and `ACTIVE_SERVER`. Initially, they are set true when connections to the GRIM widget and integration server are established. The variables are changed to false if either the GRIM widget or the server terminates before the client application. These indicators are checked before any data are sent to either server. This avoids the occurrence of serious errors in the system. To describe the client application as contained in the Integration Toolkit, we will use the client's structure charts as a guide. As a reminder, module names appear in bold and data couples are

italicized. Also, the terms requester and responder, as used in several modules, refer to the client initiating a data exchange and the client supplying data for the exchange.

Ap_sock is the main program of the client application portion of the integration client. The first task it performs is the creation of two asynchronous sockets, *Sock* and *Sock2*. The first is an Internet socket which will be used for communication with the integration server. For this purpose, the port number and Internet address of the host machine on which the server resides must be known. The second socket, *Sock2*, is used by the client application for communication with its dedicated GRIM widget in the UNIX domain. For this to be possible, the pathname of the socket file used must be identical in the client application and GRIM. Once these sockets are created, the client application issues connection requests until both sockets are accepted. After connections have been established, the client application asks the server for a list of clients from which he can request data. This list will be used to initialize the client's GRIM widget selection list. **Ap_sock** then invokes the CAD application. In order for the AP/SOCK and the application to share vital information with one another (assuming application source code is available), common data must be initialized in the **ap_sock** module and passed to, or referenced in, the CAD application. All initializations of common data elements must be deleted from the CAD application. In this manner, a common data buffer is established.

Because the sockets used by the client application are asynchronous, an event handler is created to check continuously for signals incoming on the sockets. This is done without disrupting the execution of the CAD application. When a signal does arrive, the CAD application is suspended until the signal has been "handled". When

processing of the signal has been completed, control is returned to the CAD application which resumes at the point it left off.

The signal is detected in the handler by the *select* subroutine which sets *read_mask* to include both active sockets. When a signal is detected, the *read_mask* is reset to contain only the socket descriptor on which the signal occurred. This means that either *Sock* (the server socket) or *Sock2* (the GRIM socket) is the *read_sock* sent to the function *cl_rdmsg*. *Cl_rdmsg* reads a header's worth of data from *read_sock*. If no data are present, the signal indicates that the connected process has terminated. If there are data, the header is read and sent to the function *cl_swop*. This module is responsible for evaluating the header based on major and minor opcodes. Table 2 shows a graphical depiction of how modules are located based on the header opcodes.

send_name This module will send the application's name to the process connected to the socket by the *read_sock*. The process could be either the integration server or the GRIM widget since they both need this information.

get_cl_list The module accepts the list of exchange clients from the integration server. The list is then relayed to the client application's GRIM widget for display. In this case, *read_sock* represents the server socket (*Sock*) and *write_sock* the GRIM socket (*Sock2*).

relay_data_request This function reads the name of the client that will respond to the request (*responder*). The module then builds a header which will direct the server to the module that will handle the relay of this request for buffer data. The *header*, *responder*, and *requester* (name of the client requesting the data) are then sent to the integration server. In this scenario, the

Table 2: Client application opcode table

	major opcode 0	major opcode 1	major opcode 2	major opcode 3	major opcode 4
minor opcode 0	send_name		update_widget		
minor opcode 1		relay_data_request	receive_buffer	respond_to_buffer_request	
minor opcode 2	get_cl_list				
minor opcode 3		req_attrib_list	relay_attrib_list	give_attrib_list	
minor opcode 4		req_from_attrib_list		respond_attrib_item	

read_sock represents the GRIM socket, since this is where requests for buffer data originate. *Write_sock* corresponds to the server socket.

req_attrib_list The first step performed by the module is to build a header which will direct the integration server to the function that will relay the request for an attribute list to a client in the integrated system. The name of that client is read as *responder*, and subsequently the *header*, *requester*, and *responder* are sent to the server. Again, *read_sock* is the GRIM socket and *write_sock* the server socket.

req_from_attrib_list This function first builds and writes a header that will allow the integration server to locate the module which will receive the requested list item from an attribute list. This information is then relayed back to the client which owns the attribute list. The name of the client that will respond (*responder*) is read. Subsequently, *responder* and *requester* (client requesting the transaction) are written to the integration server. The list item number, *list_num*, is then read from the GRIM socket and written to the server socket.

update_widget This module receives a name from the server along with a flag carried in the *size_in_bytes* field of the data header. This information is passed onto the client application's GRIM widget where the flag indicates whether to add or delete the name from the selection list (list of exchange clients).

The last five functions that will be discussed are all application specific.

receive_buffer This module is used to receive buffer data that were requested from a client in the integrated system. The data structure of the receiving CAD application will dictate the order and form with which the data from the integration server are read. Once this

information has been determined, it is then possible to design the portion of the server module that will be sending these data to the receiving client. It is important to remember that the order in which the data are sent to a socket is the order in which they must be read.

- relay_attrib_list** This function is dependent on the order and format with which the requested attribute list is sent from the server. It is possible that this could be a generic module with the number of list items being variable, but the items themselves are text strings. In this case, the example given in this module's m-spec can be used in the base system as it appears.
- respond_to_buffer** This module supplies data in response to a request from a client in the integrated system for its current buffer data. Again, the data structure of the current model determines the form of the message generated. The transfer function at the server which receives this data must be aware of the order and form in which these data are transmitted.
- give_attrib_list** It is the task of this module to compile a list of data-related attributes that other clients in the integrated system can ask to see. This list can be dynamic in nature if it depends on the current model for information.
- respond_attrib_item** This module produces the data which correspond to items on the attribute list which this client has sent to other clients in the integrated system (see **give_attrib_list**). Its first task will be to read in the list item identifier and gather data which the list item represents.

6.2.3 Integration Server Structure Charts

The server component of the integrated system is implemented in the Integration Toolkit as a series of structure charts. These charts and the m-specs which describe them are in Appendix E. The structure charts are composed of modules, each of which have a corresponding module specification (m-spec) that contains actual C-source code when possible. The only changes or additions to the integration server occur in the module called **resolve_header** which invokes transfer functions to treat incoming data based on the major and minor opcodes contained in the signal header. This will be explained in more detail as the discussion progresses.

Just as in the case of the client application structure charts, the minimum system configuration can be achieved by filling in modules that are only stubs in the structure chart. "Stub" means that no C-source code exists in the module specification because the module is application specific and cannot be generalized. A minimum configuration is one in which clients request data as defined in the data flow diagrams described earlier. It is imaginable that information other than that defined in the data flow diagrams needs to be passed in the system. The integration system designer can modify the server and clients, using the Integration Toolkit and the CASE workbench, to generate and receive these new data. The methodology used to do this was described at the beginning of the previous section on client application structure charts.

The server is meant to run continuously as a background process with a well-known port number, allowing clients to connect and disconnect at will. Client activity of this sort does not have an adverse effect on the integrated system as a whole. The integration server is also modular since functions can be added to the subroutine that manages the transfer functions (**resolve_header**). To get a sense of how the integration

server functions, a module-by-module description of the structure charts follows.

Please note that the data structure `sock_struct` contains the number of sockets currently connected to the server, an array of those socket descriptors, and an array of client names which correspond to the processes connected to each of the socket descriptors. This data structure is defined in the include file `mysock2.h` which is located in Appendix F.

Serv is the main module of the integration server. Its first task is to read the file which contains a list of relationships between clients in the integrated system. In other words, a list of clients who can send data and those who can request or receive it. For every relation listed in this file, there must be a transfer function at the server that can handle the exchange of data between those two clients. An example of file entries is shown in Figure 19. The term "sender" in the relation file implies the application which can generate data and "receiver" the application that can request and receive these data. From the figure, we can deduce that client X can request data from client Y and vice versa. However, the ACSYNT/B-Spline Toolkit data exchange is valid only in one direction. Using information obtained from this file, a global data structure called `xchg_struct` is filled. This data structure will be used to compile a list of exchange clients for new clients in the system. It is also referenced when updating clients in the integration system because a client process has terminated and can no longer supply data. In the C source contained in the top-level integration server `m-spec`, the filename of the relation file is defined as `exchange_buds`. This is the name of the file used in the prototype. This name is easily changed by editing the `m-spec` and changing the filename.

```
NUMBER OF EXCHANGES IN FILE = 5

SENDER = ACSYNT/
RECEIVER = B-SPLINE TOOLKIT/

SENDER = ACSYNT/
RECEIVER = B-SPLINE TOOLKIT 520/

SENDER = ACSYNT/
RECEIVER = B-SPLINE TOOLKIT SGI/

SENDER = CLIENT X/
RECEIVER = CLIENT Y/

SENDER = CLIENT Y/
RECEIVER = CLIENT X/
```

Figure 19: Sample relation file entries.

When the exchange data structure (*xchg_struct*) is complete, the program opens a socket on which to listen for connections. A socket data structure (*sock_struct*) is used to keep track of the socket descriptors and the name of the client processes with which they communicate. Once the initialization phase is complete, the integration server repeatedly executes *set_sel*. This function will continuously check to see if there is any activity on any of its sockets. A *read_mask* is set to include all sockets managed by the integration server. The select subroutine looks at all the sockets in the *read_mask* for activity. If a signal has occurred, the *read_mask* is modified to contain only that socket which has received the signal. The *read_mask* and the *sock_struct* (containing all socket descriptors and their client names) are passed to *is_eval_sel*. This function checks the *read_mask* against all sockets in *sock_struct* to determine on which socket the signal occurred. If the signal occurs on the listening socket, it is a connection request from a new client. The connection is accepted and the resulting socket is put into *sock_struct*. In order to cross-reference this socket descriptor with its client name, the integration server must obtain the name of the client connected to it. To obtain this information, the server sends a request to the new client for its name. Additionally, all clients who can request data from the new client are instructed to add its name to their selection lists.

If the signal occurs on any socket other than the listening socket, the socket descriptor (*read_sock*) and *sock_struct* are passed to *rd_msg*. This module is responsible for reading the header from the active socket (*read_sock*). If no header data are present, the client process to which the *read_sock* is connected has terminated. The socket is closed and all clients in the integrated system which had listed that client in their selection lists are informed to delete its name from the list. If there are data on the socket, the header is read and sent to *resolve_header* along with the *sock_struct*.

Resolve_header determines which module will handle the signal based on the `maj_opcode` and `min_opcode` fields of the header data. Table 3 graphically depicts how the major and minor opcodes are used to locate modules within the integration server.

put_cl This module receives the name of a new client in the integrated system and puts the name in `sock_struct` to allow the new client's socket descriptor to be referenced by name as well. The client name is then sent to **update_widget** which will compare the new client name with the sender elements of the exchange structure (`xchg_struct`). Whenever there is a match, the receiver in the exchange structure array which corresponds to the sender is a client which can display the new client's name in its selection list. `Write_sock` is the socket descriptor which corresponds to these clients. The new client's name is sent to all clients who can request data from it.

det_list This module fills the request for an initial list of clients for the selection list. This request is issued by a client who has recently connected to the integrated system. The module first reads the name of the new client and then passes the name, the `read_sock` (which is the new client's socket) and the `sock_struct`. The next step is to match the new client's name with the receiver field of the `xchg_struct` (structure containing sender/receiver data exchange relationships). A list of clients who can send data to the new client is compiled, a header is constructed, and both are written to the new client. The header field `size_in_bytes` contains the number of list items the new client should expect to receive.

request_data Requester and responder are used by the server to receive data from one client, determine the responding client's socket descriptor based on its name, and then to pass on the data to the responding client. This module's purpose is to relay a request for buffer data from the requester to the responder. The function

Table 3: Integration server opcode table

	major opcode 0	major opcode 1	major opcode 2	major opcode 3	major opcode 4
minor opcode 0	put_cl				
minor opcode 1	det_list	request_data	transfer_l		
minor opcode 2					
minor opcode 3		request_ attrib_list	relay_attrib _list_s		
minor opcode 4		request_from _attrib_list			

reads in the requester and responder names. The socket on which the responder communicates with the server is determined by cross-referencing the responder with its socket descriptor. The server function then constructs a header and sends it along with the requester's name to the responding client.

request_attrib_list The purpose of this module is to relay a request for an attribute list from the requester to the responder. Both names are read by the server function, the responding socket is located (using **sock_det**) and a header is built. The header, the requester's name, and the responder's name are sent. The responder is sent its own name because it will in turn send it back to the server in its responding message along with the attribute list .

request_from_attrib_list The purpose of this module is to relay an item number from the requester to the responder. This item number corresponds to an element of the attribute list belonging to the responder. The function reads *responder* and *requester* from the socket connected to the requester. The responder's name is used to locate its corresponding socket descriptor, *response_sock*. A header is built and sent over *response_sock* along with the requester's name and the list item which requires action.

The last two functions discussed are application specific.

transfer_1 The purpose of this module is to receive data from a particular client in the integration system and transform or translate that data into the format used by the requesting client. The easiest way to structure this module is to have sub-modules called from the main transfer function which are each dedicated to one receiving client. In this manner, the data sent to the transfer function can be treated specially for each client and modified, manipulated, or even just relayed to the receiver (client which requested the data). The choice of sub-module can be made

using a switch statement based on the receiver's name. The client that sends data to the transfer function can send all the data it can compile which will satisfy all data possibilities demanded by the sub-modules. By doing this, the server takes the responsibility away from the sending client.

relay_attrib_list_s The purpose of this module is to accept a list of attributes from a responding client and send it on to the client which requested it. It is left as application dependent at this point, though it could be generalized if the attribute list were limited to text strings. In this case it would suffice to send the number of list items in the header field `size_in_bytes`, followed by the list. Requester and responder will need to be sent as well.

In summary, it is necessary to give an overall sense of how the integration clients and server interact. The clients can be arranged such that they are all on different workstations which are part of a network, all on the same workstation, or a combination of the two preceding possibilities. If one or more clients occur on the same workstation, they will appear in separate windows, as will each of their GRIM widgets. For n clients on one workstation there will potentially be $2n$ windows. The integration server can be located on the same machine as one or all of the clients, or on a different workstation in the network. The integration server executes as a background process; therefore, there will not be a window dedicated to the server process. This description will become more obvious in the next chapter which discusses the distributed integration solution prototype.

7.0 DISTRIBUTED INTEGRATION SOLUTION PROTOTYPE

To test the validity of the integration solution, a prototype integration system has been developed. This prototype uses two significant CAD applications developed in the Computer-Aided Design Laboratory of Virginia Tech which are integrated using the method of freely connected interfacing. The first application is called ACSYNT (AirCRAFT SYNThesis), which was developed jointly at the Virginia Tech CAD Laboratory and NASA Ames Research Center. It is an interactive design and analysis tool used to develop conceptual models of advanced aircraft. The application uses the FORTRAN, C, and PHIGS standards. The main module of ACSYNT was written in FORTRAN, and it is this module which must be invoked from the AP/SOCK interface. The second application in the prototypical system is the ACSYNT B-Spline Module (B-Spline Toolkit). Although the name implies that the B-Spline Module is part of ACSYNT, the two applications are separate and independent programs which were designed to complement each other. The applications were designed such that the B-Spline Module is able to read hermite data files created by ACSYNT. The B-Spline Module is an interactive CAD application which converts the geometry descriptions commonly used in conceptual aircraft design codes to descriptions which meet the requirements of preliminary design systems. The module enables designers to compute intersections of surfaces described using non-uniform bi-cubic B-Splines and uses a filleting algorithm to blend surfaces along iso-parametric curves. This application is C-based and also uses the PHIGS standard for graphics. Though these two applications were created to work together, they do not have similar data structures. ACSYNT produces geometric models which use hermite surface representations. The B-Spline Module can read files containing hermite surface data, but internally, surfaces are represented as nonuniform B-Splines.

The machine chosen to host the integration server is an IBM RISC System/6000 Model 530. In this implementation, the two clients can be accessed on the 530 or an IBM RISC System/6000 Model 520. The communication protocol is TCP/IP using Ethernet adapters at the workstations. The BSD socket libraries vary slightly on workstations from different vendors, but the porting process from one UNIX-based workstation to another is relatively simple. For example, the clients, initially developed on the IBM RISC System/6000 under the AIX (IBM's implementation of UNIX) operating system, were ported to the SGI platform in a single afternoon with the exception of the GRIM interface. This is because the current implementation of the SGI in the lab runs using the windowing system called NeWS instead of X-Windows. Because of this discrepancy in windowing environments, the GRIM would need to be ported to utilize the interface toolkit called 4Sight, which is the NeWS equivalent of Motif. The Silicon Graphics platform is also capable of operating under X-Windows, and if this were the case, no port of the GRIM interface would be necessary.

The first step in the design process was to determine the data exchanges possible within the integrated system. It was decided that the B-Spline Module should be able to request data pertaining to the model in ACSYNT, but not vice versa. The transaction path is one-way because, although the B-Spline Toolkit can handle hermite surfaces and modify them to produce nonuniform B-Spline surface representations, the ACSYNT application presently has no ability to utilize B-Spline surface representations in the analysis portion of the program. With this in mind a file of exchange relations, called `exchange_buds`, was created and the ACSYNT application was defined as the *sender*, while the B-Spline Toolkit was defined as the *receiver*. This file will be read by the

integration server at initialization and used later to compile exchange client lists which are displayed in the GRIM widget of each client application.

In order for the user to request data exchanges in the integrated system, a GRIM widget must be created for each client application. The pathname in the GRIM modules was changed to a unique value for both the B-Spline Module and ACSYNT's widgets, then each was compiled using an unduplicated name for the executable file. The executable widget which is dedicated to the B-Spline Module is called **grimmy**, while that which belongs to ACSYNT is called **grim2**. The pathnames used to produce the widgets must be duplicated in their owning client application's main module if communication is to occur between them.

Next, the client applications must be created. Using the base system structure charts in the Integration Toolkit, it is easy to determine which modules are application specific (this information is contained in the m-specs). The main module of each client application must be modified to include the unique pathname of its GRIM widget and to make global any data that are common to the client interface and the CAD application it manages. A description of how ACSYNT and the B-Spline Module were modified to fit into the form of the client application follows. Often the exact order of module execution is not preserved when describing a module's function. This is done when the function of the module is more easily understood when events are explained in a modified sequence. In any case, the concept remains intact.

7.1 The ACSYNT Client Application

Following the flow of the client application structure charts contained in the Integration Toolkit, the first module modified was `ap_sock`. In this module, the pathname definition was changed to match that of the pathname defined in `grim2` (ACSYNT's GRIM widget). The pathname was defined as `/u/michele/grim/acsynt/s.acssock`, where `s.acssock` is the UNIX socket filename. Next, the main module of ACSYNT had to be changed to a subroutine, thus allowing it to be invoked from the module `main_ap`. It is not a problem that the calling module is C-based and the subroutine is FORTRAN-based. Since ACSYNT is primarily a FORTRAN-based application, any data used in the common buffer shared with the client interface would need to be passed in an argument list to the subroutine which was formerly ACSYNT's main module. In the case of this CAD application, a common data buffer was not needed. Instead, functions defined in ACSYNT were utilized which access a geometry database containing current information on the displayed model were utilized. More explanation on these functions will be given as the discussion progresses, but the main point is that no data needed to be passed into ACSYNT from the client interface; therefore, no data were declared as global between the two.

In order to send data to the integration server for eventual transfer to the B-Spline Module, the modules called by `cl_swop` which transmit model data need modification for use with ACSYNT. There are three modules which will need development. These are the modules which respond to a request for buffer data, compile an attribute list for ACSYNT models, and respond to a request for an item from the attribute list. No action is necessary for modules designed to receive buffer data from another client or to relay the attribute list from a second client to its dedicated GRIM widget. This is because the only other client in the integrated system, the B-Spline Module, will not

send that kind of data since its only function in the system is to receive data from ACSYNT. As a result these modules will not be used, and are therefore ignored. They will, however, be addressed in the B-Spline section.

To extract data from the ACSYNT geometry databases, several of the ACSYNT database utilities were accessed from the modules called by `cl_swop`. Again, we will mix the C-code of the module with calls to the utility subroutines which are written in FORTRAN. Pointers must be used to pass data from a C function to a FORTRAN subroutine. This means names of arrays, which are pointers to a location in memory, need no special consideration, but reals and integers need their addresses passed to the subroutine instead of their values. Take for example the utility function used to get a component list from the ACSYNT geometry database:

```
(void) gtgmpk(&ncomps, comps);
```

where `ncomps` is defined as an integer and `comps` as an array of integers. The ampersand preceding `ncomps` represents the address of the variable in C code. The `(void)` in front of the subroutine name is necessary when calling FORTRAN functions from a module written in C.

The ACSYNT module written to send data to the integration server because of a buffer request from another client is called `respond_to_request`. This module uses several geometry database utility functions from ACSYNT to construct a data representation of the current model. Before sending data, the header is built such that `size_in_bytes` contains the number of components in the current model. The major and minor opcodes are 2 and 1, respectively, which direct the data received by the server to the

transfer function which will handle it. By paging forward to Table 6, the opcodes can be used to locate the module at the server which will handle data transfer. The data are sent to the server component by component. Data transmitted includes component name, component number, color, number of cross-sections, number of points per cross-section, and finally a list of points. The transfer function located at the integration server which corresponds to the ACSYNT client application must receive the data in the exact order they were sent.

The module used by ACSYNT to compile an attribute list is called **give_attrib_list**. It too uses utilities for accessing the geometry data structure. In fact it makes use of the subroutine used in the above example to get a current list of the components in the current model. The header is built such that the `size_in_bytes` contains the number of components in the list. The major and minor opcodes are 2 and 3, respectively, which guide the signal received by the server to the module which will handle the relay of the attribute list to the requesting client. Again, using Table 6, the function at the integration server which will treat this data is easily located.

The module which will react to the request for data based on the choice of an element from ACSYNT's attribute list is called **respond_attrib_list**. The list item is identified by a number which corresponds to the component number in ACSYNT's geometry data structure. The data representing the component are extracted from the data structure and sent to the same transfer function at the server as were the buffer data. This is because the form of the data is identical. The only difference is that a buffer data request sends several components worth of data, while the data of only one component will be sent for the response to the attribute list choice request.

In order to place these modules in the proper location with respect to the opcode tables, the major and minor opcodes to which they will respond must be defined. The major opcode is already specified with respect to purpose (see Chapter 4.3). Since all of the modules described for the ACSYNT client application are used to respond to requests, they are of major opcode category 3 - Response. The only factor left to resolve is the minor opcode. There is really no methodology for choosing the minor opcode, except that it could be used to denote a transaction level. A transaction level could be thought of as a request/response sequence. This sequence is illustrated in Figure 20. Note that the flow of the sequence is from left to right.

As can be seen from the figure, when a client requests data, the request is relayed through the server to the responding client. When the responding client sends data in fulfillment of the request, the data goes first to the server, then finally to the requesting client. All major opcodes for this sequence are pre-defined according to what the data does, but the minor opcode should be the same (or mostly the same) for all members of the transaction. Table 4 shows the opcode values for the modules described in this section, as well as the modules which are used for initialization.

When the client application has been compiled and linked to form an executable, in this case called `acsynt`, the last step is to create a UNIX script which will start the GRIM (`grim2`) as a background process, sleep for about three seconds, then invoke the client application.

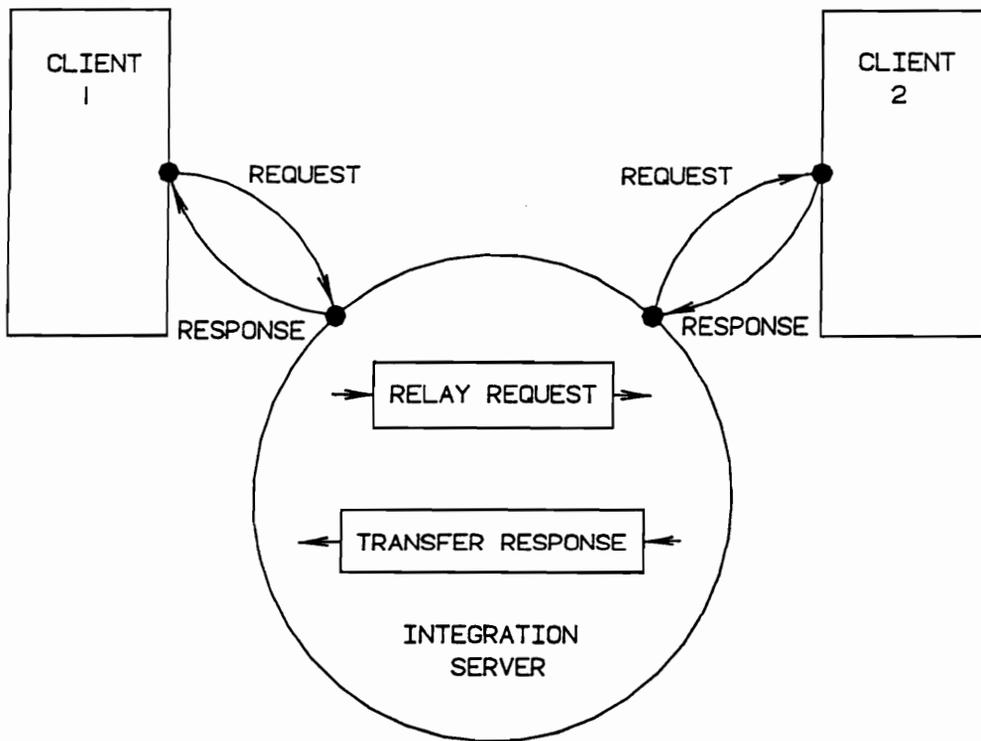


Figure 20: Request/response sequence.

Table 4: The ACSYNT client application opcode table.

	major opcode 0	major opcode 1	major opcode 2	major opcode 3	major opcode 4
minor opcode 0	send_name		update_widget		
minor opcode 1				respond_ to_request	
minor opcode 2	get_cl_list				
minor opcode 3				give_attrib_list	
minor opcode 4				respond_attrib _list	

7.2 B-Spline Client Application

The B-Spline client application was created by modifying the main module to include the same pathname definition used with the B-Spline's GRIM widget. The socket filename used is `s.grimsock` and the pathname is the path location of the socket file (in this case `/u/michele/grim/execs/s.grimsock`). The source code of the B-Spline Toolkit was then modified such that the main module was now declared as a sub-function. In addition, common data were declared between the client interface and the B-Spline Module. The data common to the two is a data structure called `MODEL`, which contains all the data necessary for the display of the `B_Spline` model. Figure 21 shows the declaration of the `MODEL` data structure. Note that `MODEL` contains a pointer to another structure called `comp_data`. This structure contains data specific to a single component. It is represented by a pointer which allows `MODEL` to allocate space for one component at a time, instead of statically allocating space during initialization of the program.

When incoming data are to be displayed by the B-Spline Toolkit, it is read into the `MODEL` data structure, then displayed. It is only because the client interface has access to the `MODEL` structure that data can be read directly into it. Since this data structure is initialized in the `ap_sock` module as a global, it is referenced in the former main module of the B-Spline Toolkit as an external variable. All initializations of the `MODEL` variable that were previously performed by the former main module are deleted. The other modules which need to be modified to form the B-Spline client application are those which deal with the request and reception of data. Since the B-Spline Module will not produce data to send to ACSYNT, no modules which are

Model Data Structure

```
typedef struct {
    int num_comp;           /* number of components in model */
    int acs_root;          /* root structure id */
    int nubs_root;        /* Non-Uniform B-Spline root id */
    int int_root;         /* Structure id for intersection data */
    int fillet_root;
    comp_data *comp;      /* pointer to beginning of linked list */
    struct intersection_type *intlist; /* list of intersections */
}MODEL;
```

Component Data Structure

```
typedef struct compdata_type {
    int comp_number;      /* component number */
    char comp_name[20];  /* component name */
    int acs_id;           /* structure id */
    int nubs_id;
    int *hull_id;
    int fillet_id;
    int open[2];         /* open flag 1 closed 0 open */
    int color;           /* component color */
    int existence;       /* 1 exists 0 does not exist */
    int nu;              /* rendering in u */
    int nw;              /* rendering in w */
    int acs_ncross;      /* number of cross sections */
    int acs_npts;        /* number of pts per xsection */
    float ***acs_pts;    /* pointer to component pts */
    float ***acs_utan;   /* pointer to tangents in u dir */
    float ***acs_wtan;   /* pointer to tangents in w dir */
    int nu_knots;        /* number of u knots */
    int nw_knots;        /* number of w knots */
    float *u_knot;       /* u knot array */
    float *w_knot;       /* w knot array */
    float ***hull;       /* control hull */
    struct compdata_type *next; /* pointer to next component */
}comp_data;
```

Figure 21: B-Spline MODEL data structure.

responsive to model data requests are necessary. The modules which need modification include:

- The module which will request an element from the attribute list of another client (remember this could be considered generic if the stipulation is made that integers are used to identify list items).
- The module which relays the attribute list from the responding client to the GRIM widget for display (also possibly generic).
- The module which receives buffer data requested from another client.
- The module which receives attribute data requested from another client.

In most cases the module which receives buffer data will also handle the attribute data. This is because the transfer function at the integration server is cognizant of the data structure of the receiving client and gears the sending of transfer data to that structure. It is possible, however, that a special case exists. In the case of the base structure charts for the client application, both types of data are thought to be handled by the **receive_buffer** module.

The two modules mentioned above which deal with aspects of relaying the attribute list can be generalized; thus, the modules in the Integration Toolkit need no modification. This is because the stipulation is made that attribute lists consist of text strings and the item identifiers passed during a request based on the attribute list are integers. This is the case in the client applications used in the prototype system. The module used to relay the request for an attribute list is called **req_attribute_list**. It sends a header to the integration server with major and minor opcodes of 1 and 3, respectively. Again, refer to Table 6. The module which accepts the list of attributes in the form of text strings and passes them on to the GRIM is called **relay_attrib_list**. This module builds a header with a major and minor opcode of 2 and 3 which is sent with the data to the

GRIM (see Table 1). The module which sends the identifying list item, in the form of an integer identifier, is called `req_from_attrib_list`. This function builds a header with major and minor opcodes of 1 and 4, respectively, and sends it to the server. Remember, a major opcode of 1 is defined for requests, while 2 is for buffer updates or transfers.

The only module left to be modified is the one which receives data from the transfer function at the integration server. This module is called `receive_buffer` and its first task is to clear out the existing MODEL structure (current data model). Next, initialization of the several elements of the MODEL data is performed and the number of components contained in the incoming data is read. For each component, the component name, number, color, number of cross-section, number of points per cross-section, and a list of points is read. Note that this read order must be respected by the order data is sent from the integration server. After all data are read from the integration server, the MODEL structure is passed to a function which computes the hermite tangents for the surfaces. The model is then ready for display.

Table 5 defines the major and minor opcodes as defined in `cl_swop`. These opcodes enable the B-Spline client application's event handler to locate the proper module to handle the signal data.

When the client application has been compiled and linked to form an executable, in this case called `acsnums`, the last step is to create a UNIX script which will start the GRIM (`grimmy`) as a background process, sleep for about three seconds, then invoke the client application.

Table 5: The B-Spline client application opcode table.

	major opcode 0	major opcode 1	major opcode 2	major opcode 3	major opcode 4
minor opcode 0	send_name		update_widget		
minor opcode 1		relay_data_request	rcv_acsynt		
minor opcode 2	get_cl_list				
minor opcode 3		req_attrib_list	relay_attrib_list		
minor opcode 4		req_from_attrib_list			

7.3 The Prototype Integration Server

The only modules which need to be modified or created are those which handle the signal data incoming from the integration clients in the system. In other words, modules called from the **resolve_header** function (the function which directs signal data to a certain module based on opcodes contained in the signal header). Of the modules called by **resolve_header**, several are generic and do not need modification. The ones that are application specific, however, do need to be modified or even added. Such modules include transfer functions which receive input data from one client in the system (ACSYNT) and transform or modify it to send to another (B-Spline Toolkit). In the design of a transfer function, it is necessary to know the order of the input data so that it can be read off of the socket. It is also necessary to know the method in which the receiving client expects to read sent from the transfer function. Given these two constraints, the transfer function then defines a method to either transform or translate the incoming data into output.

The transfer function which the integration server uses to transfer data from ACSYNT to the B-Spline Toolkit is called **acsynt_to_bspline**. This module receives the data sent by ACSYNT in the order they were sent out. It also builds a header to send to the receiving client in which the **size_in_bytes** contains the number of components to expect and the major and minor opcodes are 2 and 1, respectively. These opcodes correspond to the position of the **receive_buffer** module of the B-Spline client application in Table 5. The data received from ACSYNT are evaluated and the points in each cross-section are reordered to be consistent with the representation used in the B-Spline Module. The data are then transmitted to the receiving client (B-Spline Toolkit) in the order it expects to read the data from the receiving socket.

There are two other modules which may need modification. One relays the attribute list compiled by one client to another which will display it. The other relays the attribute list item which is being sent back to the client who owns the attribute list for data which correspond to the item. Both of these modules can be considered generic if, as mentioned before, the attribute list is limited to text and the item returned is an integer. For the purposes of this prototype this constraint is valid and the modules are considered general. For the sake of understanding data flow in the system, the module which requests the attribute list from ACSYNT, **request_attrib_list**, builds a header of maj_opcode 3, min_opcode 3. From Table 4, it is seen that **give_attrib_list** will respond. As was explained previously, **give_attrib_list** compiles a list and sends it along with a header of major and minor opcode 2, 3. This corresponds to **relay_attrib_list_s** in Table 6. This module passes on the list data to the B-Spline client also with a header of 2, 3. Referring back to Table 5, we see that the opcode sequence locates **relay_attrib_list** at the B-Spline client application. The B-Spline module keeps the same header definitions (major 2, minor 3) and transmits header and list data to its GRIM widget. Using Table 1, we see that the header corresponds to **g_make_attrib_list**, which will read in the list data and display them to the user. Table 6 shows how the opcodes contained in the header structure of data incoming to the integration server are directed to the module which will handle the signal.

By studying the headers defined by each module contained in Tables 1,4,5 and 6, the flow of data in the system can be traced. These are the tables contained in the opcode table portion of the Integration Toolkit. For integration system designers who want to add new modules to the integrated system, a table of opcodes for each client application and the server is a graphical aid when defining new headers and locating modules.

Conceptually the integration server should be able to handle any number of client applications. However, in the prototype coded for this research, a limit of 50 clients was set because of the array sizes in the socket data structure. Linked lists would eradicate this limitation, but for the purpose of this research, arrays were faster to code and to execute. Code specific to the integration server portion of this prototype integrated system has been delivered to the research sponsor, though some of the application-specific code produced for the prototype appears as examples in the m-specs contained in Appendices C, D, and E.

7.4 Data Exchange in the Prototypical Integration System

Once the integration client and server have been created, the system is ready for implementation. The server is started as a background process running on the IBM RISC System/6000 Model 530. The server can be run continuously if desired. The clients can then be started at any time. If the server is not available and a client is started, an error will result. The B-Spline client application can be run on either the IBM RISC System/6000 Model 530 or Model 520. ACSYNT is also available on both of these platforms. These machines are connected by a local area network using TCP/IP and Ethernet. For the sake of an example, let us consider the scenario where the server is running as a background process on the Model 530, the B-Spline client application is on the Model 520, and the ACSYNT client application on the SGI 4D/80GT. This configuration is shown in Figure 22.

The ACSYNT client application is started using the `racsynt exec` which can be found in Appendix F. The resulting client is shown in Figure 23. At this time, the ACSYNT

Table 6: Integration server prototype opcode table.

	major opcode 0	major opcode 1	major opcode 2	major opcode 3	major opcode 4
minor opcode 0	put_cl				
minor opcode 1	det_list	request_data	acsynt_to _bspline		
minor opcode 2					
minor opcode 3		request_ attrib_list	relay_attrib _list_s		
minor opcode 4		request_from _attrib_list			

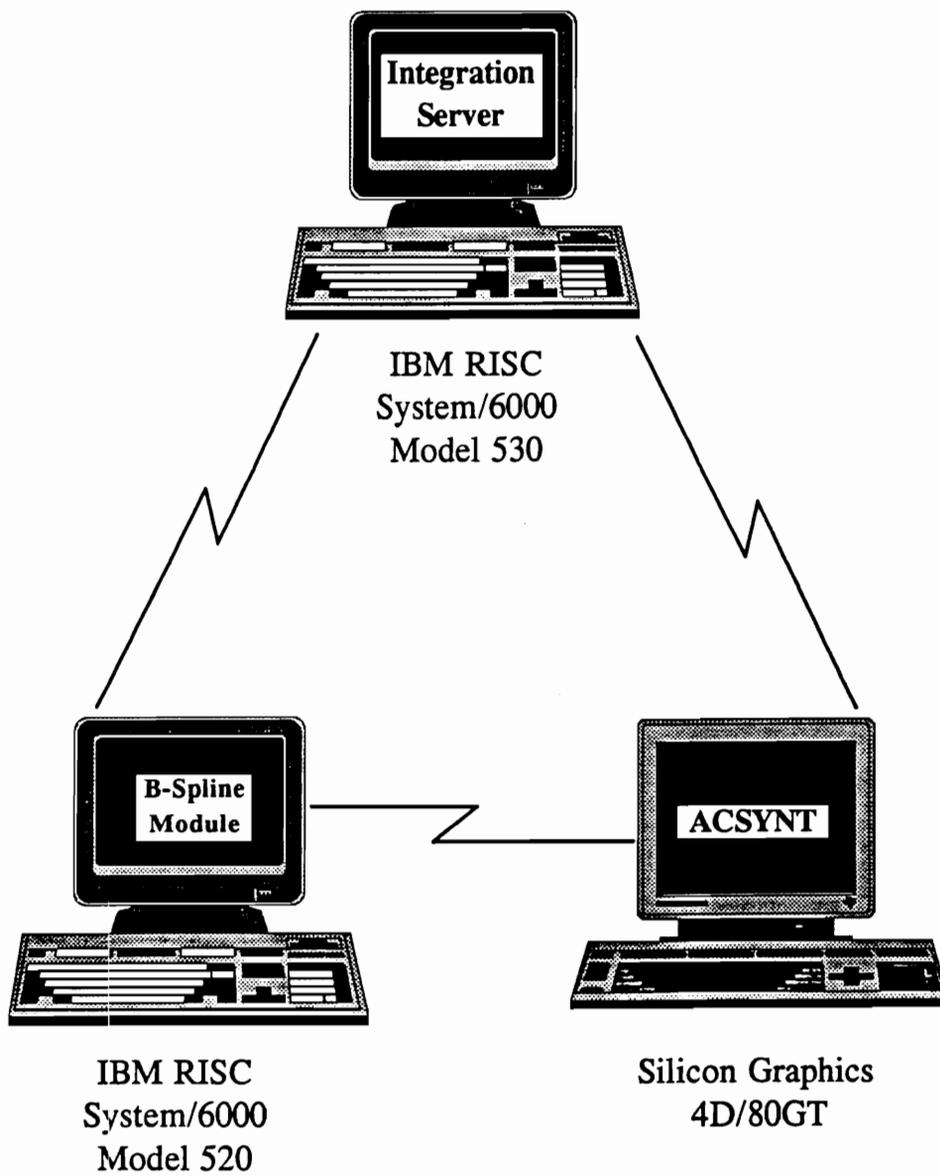


Figure 22: Prototype client applications and integration server.



Figure 23: The ACSYNT client application.

client application is connected to both its dedicated GRIM widget and the server. The client application has sent its name to the GRIM for display above the selection list, as seen in Figure 23, and has sent a request to the integration server for a list of exchange clients. No data will be returned by the server in fulfillment of this request since there are no other applications currently connected to the integration server, and because there are no sender/receiver relationships listed in the server's exchange structure where ACSYNT is a receiver. Therefore the selection list remains empty. A user can now proceed to use the client application as if it were a stand-alone program (not connected to the integrated system). A conceptual-level model of a General Dynamics F-14 fighter jet is created and displayed.

The B-Spline client application is started using the `racs` script described earlier. This `exec` can be found in Appendix F. The dedicated GRIM starts, a few seconds pass, then the B-Spline application client appears. The client application connects to the GRIM and soon thereafter sends its name as the response to a request. The name is displayed in the GRIM widget above the selection list. The client also connects to the integration server, and requests a list of exchange clients. The server responds with the names of clients currently connected from which the B-Spline client application can request data. This name, ACSYNT, is placed in the widget's selection list as shown in Figure 24. Note that the B-Spline option is not included in the selection list of the ACSYNT application since the inverse data exchange was not defined in the exchange relations file. In other words, there is no provision for data passed from the B-Spline Module to be sent to ACSYNT; thereby obviating the need for a user at ACSYNT to request data from the B-Spline Module.

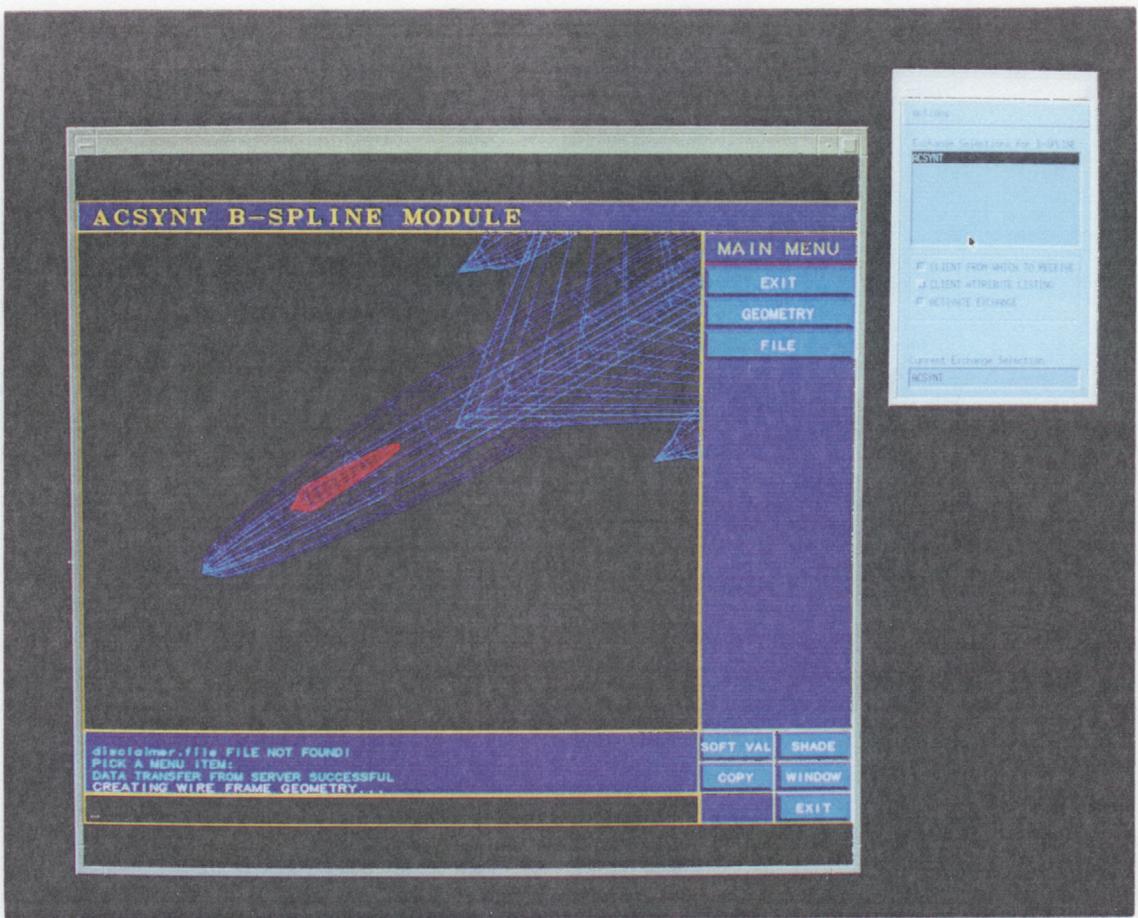


Figure 24: The B-Spline client application prototype.

The B-Spline client application is now ready to request data from ACSYNT. If the current model displayed by ACSYNT (F-14) is desired, the following sequence of widget manipulations must occur. The first toggle button in the widget, *request buffer data from client*, is depressed, "ACSYNT" is chosen from the selection list, and the *activate exchange* toggle is selected. The activate exchange toggle will display a panel with the name of the sending application and wait for the user to select one of two buttons for further action. If the OK button is chosen, the data exchange takes place. If the CANCEL button is selected, the exchange is abandoned.

Alternatively, the B-Spline client could request a single component from ACSYNT. To do this, the *client attribute listing* toggle is depressed and the "ACSYNT" client name is chosen from the selection list. ACSYNT responds by supplying a list of data attributes, which are displayed by the B-Spline client application's widget. The user then has the option of choosing one of the items from the list and then selecting the OK button to send the data request. If the user is not interested in the list items he can use the CANCEL button to exit the attribute list without further action. The client attribute list of a client in the integrated system is dynamic since it often depends on information about the current model being displayed. For example, in the ACSYNT client application, the attribute list consists of components of the current aircraft model. Since not every model contains the same components, the list is model dependent and must be re-created each time it is requested by another client in the system.

The Figure 25 shows an example where the B-Spline and ACSYNT clients were executed on the same workstation. This is intended to show how the clients appear after the B-Spline client has requested the wing component data from the ACSYNT client's current model.

GHIPTAIN BOND

50% COTTON FIBER

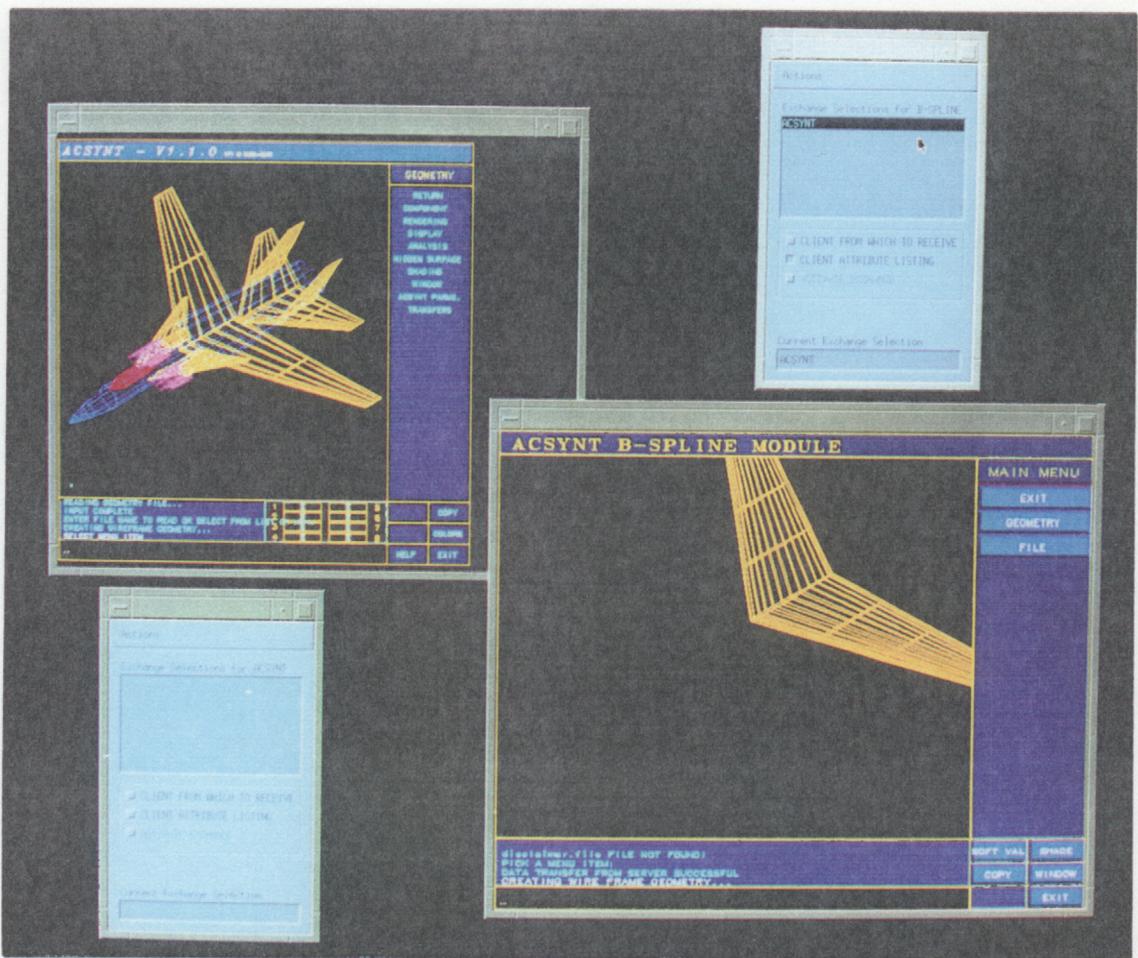


Figure 25: B-Spline and ACSYNT clients running on the same workstation.

8.0 CONCLUSIONS

The prototype integration system has proven that it is feasible to implement the distributed integration solution. The prototype was effective in demonstrating the transfer of data among CAD/CAM applications residing in a network environment on remote and local platforms. A user at an application connected to the integration server can request the current model data from a second application connected to the server. The user also has the option of requesting some component of that data in lieu of the entire model. Once the model data have been transferred to the requesting application, the model can be modified and manipulated since it now belongs to the application which imported it. It is worth mentioning that a third CAD application was added to the prototype integration system with minimal effort. The application is a GL-based surface modeler called SURF. The modeler was developed at this laboratory and uses a data structure similar to that of the model used in the B-Spline Toolkit. Because of this similarity, the transfer function at the integration server which transforms hermite surface representations into nonuniform B-Spline surfaces was utilized to send data to the new application. A client interface for SURF was constructed following the structure charts and module specifications contained in the Integration Toolkit. The reusability of modules, in this case the transfer function, and the presence of CASE tools significantly reduced the amount of time necessary to integrate a new application into the integrated system.

The prototype integration system was used to test the validity of the tools used to generate an integration system and the effectiveness of the distributed integration solution. As a direct result of the research described in this document, five objectives were achieved :

- 1) Creation of a communication protocol for data passing in the context of an integration system.
- 2) Addition to the components of an Integration Toolkit which is part of the CAD/CAM CASE Workbench.
- 3) Creation of a distributed integration solution which is implemented in the Integration Toolkit.
- 4) Development of tools in the form of data flow diagrams, process specifications, data dictionary, structure charts, and module specifications to aid an integration system designer in generating an integration system based on the distributed integration solution.
- 5) Demonstration of the validity of the distributed integration solution by a prototype system which was effected using two CAD applications.

These five items are a product of the research objectives presented in the introduction of this dissertation. Explanation of each objective appears in the order presented at the beginning of the document. First, the socket-based communication protocol used in the integration system stemmed from an investigation of mechanisms used for interclient communication. Second, the distributed integration solution was created such that it includes a core element, the integration server, which manages the exchange of data and information between integrated applications. Tools were developed to aid in the generation of integration systems based on the distributed integration solution. And finally, a prototype integrated system was implemented using the distributed integration solution as a basis for integrating two CAD applications.

In addition to the objectives stated above, requirements for the distributed integration solution were also specified. These objectives will be described one at a time in order to clarify them. The first objective was database access and storage of pertinent CAD/CAM data. Note that although a database was not included in the prototype integrated system, it was originally described as an optional member of the system. The database would essentially be another client in the integration system, connecting to the integration server and sending and receiving data through it. The second objective specified inter-application communication. Inter-application communications are complex in that they rely on several components in the integration system (such as the GRIM, the AP/SOCK Interface, and the integration server) for implementation. Thirdly, a goal of the distributed integration solution was to enable applications to run in a distributed and simultaneous environment. The applications described in this research are usually interactive in nature and are able to execute in a network environment in a concurrent fashion. A fourth specification was functional access of other applications in the integrated environment without terminating the session of the current application. This is made possible by the use of asynchronous sockets for communication. Asynchronous sockets allow the CAD/CAM application to proceed as usual until the occurrence of a signal. When a signal does appear, the application is suspended, not terminated, until the data can be taken from the socket and resolved. The fifth objective was for transfer of data among applications. This is accomplished by allowing the user to specify data exchange transactions using a widget interface (GRIM) which belongs to the client that will receive the data. All data exchanges are request oriented. Last of all, there was a specification for a system executive which oversees and manages interclient and database interactions. In the distributed integration solution, this system executive is called the integration server. All of the intended goals have been met by the distributed integration solution.

There are a few more advantages of the distributed integration solution which are worth mentioning. First of all, it allows the system integration designer to exploit the capabilities of different applications instead of locking him into a procedure for data extraction and exchange. Using the geometry database utility functions of ACSYNT to extract data is a good example of this. It can handle the integration of CAD applications whose source code may or may not be available. Clients connecting to or disconnecting from the integration server do not adversely affect the system as a whole. It enables the system integration designer to choose from three types of integration schemes. It is valid in a network environment for n clients. And finally, the use of the Integration Toolkit greatly facilitates the task of the integration system designer by giving him graphical guidelines to follow.

*

REFERENCES

- [Brau85] Brauner, K. and Briggs, D. The Second Draft of the Ad Hoc Committee on the Content and Methodology of the IGES Version 3 (The Second PDES Report), revision B, Jan. 1985.
- [Chri84] Christman, A.M., "Update on CAD, CAM, and CIM", *I&CS - The Industrial and Process Control Magazine*, vol. 57, no. 5, May 1984, pp.53-57.
- [Colt91] Colton, J.S. and Dascanio, J.L., "An Integrated, Intelligent Design Environment", *Engineering with Computers*, vol. 7, no. 5, winter 1991, pp.11-22.
- [Come91] Comer, D.E., Internetworking with TCP/IP - Volume I, Prentice Hall, Englewood Cliffs, New Jersey, copyright 1991.
- [Date90] Date, C.J., An Introduction to Database Systems, Volume I, Addison-Wesley Publishing Company, copyright 1990.
- [Enca90] Encarnacao, J.L. and Lockemann, P.C., Engineering Databases: Connecting Islands of Automation Through Databases, Springer-Verlag, copyright 1990.
- [Fari90] Farish, M., "Splendid Isolation", *Engineering*, vol. 230, no. 8, Sept. 1990, pp.20-22.
- [Fenv90] Fenves, S., Flemming U., Hendrickson, C., Maher, M. and Schmitt, G., "Integrated Software Environment for Building Design and Construction", *CAD*, vol. 22, no. 1, Jan/Feb 1990, pp.27-36.
- [Furl90] Furliani, C., Wellington, J. and Kemmerer, S., Status of PDES-Related Activities (Standards and Testing), National PDES Testbed Report Series, U.S. Department of Commerce, October 1990.
- [Guti89] Gutin, R.H., "Gral: An Extensible Relational Database System for Geometric Applications", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam 1989, pp.33-44.
- [IBM90] IBM Communications Programming Concepts - AIX Version 3 for RISC System/6000, First Edition (March 1990), copyright International Business Machines Corporation, publication #SC23-2206-00.

- [Jaya90] Jayaram, S. and Myklebust, A., "Automatic Generation of Geometry Interfaces Between Applications Programs and CAD/CAM Systems", *CAD*, vol. 22, no. 1, Jan/Feb 1990, pp.50-56.
- [John91] Johnson, E.F. and Reichard, K., Power Programming ... MOTIF, Management Information Source, Inc., copyright 1991.
- [Kim84] Kim, W., Lorie, R., McNabb, D. and Plouffe, W., "A Transaction Mechanism for Engineering Design Databases", *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore 1984, pp.355-362.
- [Liew82] Liewald, M.H. and Kennicott, P.R., "Intersystem Data Transfer via IGES", *IEEE Computer Graphics & Applications*, May 1982.
- [Lu86] Lu, L., Myklebust, A. and War, S., "Integration of a Helicopter Sizing Code with a Computer-Aided Design System", *Journal of the American Helicopter Society*, Oct 1987, pp.16-27.
- [Mars86] Marshall, J. and Van Dyne, D., "Integrating CAE, CAD, and CASE", *Digital Design*, vol. 57, no. 6, June 1986, pp.40-46.
- [Mccl89] McClure, C., CASE is Software Automation, Prentice Hall, Englewood Cliffs, New Jersey, copyright 1989.
- [Meye91] Meyers, S., "Difficulties in Integrating Multiview Development Systems", *IEEE Software*, vol. 8, no. 1, Jan 1991, pp.49-57.
- [Mykl90-1] Myklebust, A. and Pennington, S.L., A Research Report to the IBM Corporation, July 1990.
- [Mykl90-2] Myklebust, A. and Pennington, S.L., A Research Report to the IBM Corporation, Dec 1990.
- [Pall91] Pallatto, J. "IBM to Erect Vast Database Warehouse", *PC Week*, vol. 8, no. 31, Aug 5, 1991, pp.1 & 8.
- [Nye90] Nye, A. (editor), X Protocol Reference Manual, vol. 0, O'Reilly and Associates, Inc., Sebastopol, California, copyright 1990.
- [Page88] Page-Jones, M. A Practical Guide to Structured System Design, Yourdon Press, Prentice Hall Building, Englewood Cliffs, New Jersey, copyright 1988.

- [Penn91] Pennington, S.L., A Software Engineering Approach to the Integration of CAD/CAM Systems, Doctoral Dissertation, Virginia Polytechnic Institute and State University, March 1991.
- [Smit90] Smith, D. and Oman, P.W., "CASE Analysis and Design Tools", *IEEE Software*, vol. 7, no. 3, May 1990, pp.15-19.
- [Oman90] Oman, P.W., "CASE Analysis and Design Tool", *IEEE Software*, vol. 7, no. 3, May 1990, pp.37-43.
- [Reis90] Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, vol. 7, no. 4, July 1990, pp.57-63.
- [Rowe88] Rowell, L.F., Schwing, J.L. and Jones, K.H., "Software Tools for the Integration and Execution of Multidisciplinary Analysis Programs", *AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting*, Atlanta, Georgia, Sept 1988 (AIAA-88-4448).
- [Your89] Yourdon, E. Modern Structured Analysis, Yourdon Press, Prentice Hall Building, Englewood Cliffs, New Jersey, copyright 1989.

APPENDIX A: DATA DICTIONARY

application_in_data (data flow) =
[request_application_info
 | new_buffer_data
].
* The data is split into requests for the application
and new information for the application to display. *

application_out_data (data flow) =
[response_app_out
 | request_app_out
].
* The output from the application will be either in the
form of actual information (such as name, buffer data, etc)
or request data. *

attribute_list_choice (data flow) =
 responder
 + list_item .
* The responder's name is necessary to allow the integration
server to determine to which of the processes connected
to the server is the responding application. The header
is used to instruct the receiving process on how to
evaluate the data which follows. The list item is
the item number from the attribute list which was
supplied by the responding application in a preceding
request/response sequence. *

callback (data flow) =
[request_buffer_callback
 | request_attribute_list_callback
 | request_attribute_item_callback
].
* Defines the actions based on the event generated
at the widget interface by the user.*

choice_data (data flow) =
[buffer_data
 + application_name
]
| [attribute_list
 + application_name
]
| [item_from_attribute_list
].
* The user can perform one of three actions at the
widget: a request for buffer data from a named
application, a request for the attribute list of
a named client, or the choice of an element from
a previously requested attribute list. These three request
choices correspond to the three [choices] above *

client_in_data (data flow) =
[widget_in_data
 | application_in_data
].
* These are the standard data items sent by the server to clients in the integrated system. *

client_out_data (data flow) =
[widget_request_forward_server
 | response_app_out
 | initialize_client
].
* The widget_request_forward_server is data sent from the widget to the client for transmission (relay) to the server. Response_app_out contains actual information from the application itself (such as buffer data, attribute list, etc). Initialize_client is performed only once at setup of the client. *

common_buffer (store) =
[application_name
 | attribute_list
 | attribute_data
 | buffer_data
].
*The attribute list is given to a requesting client. With it the requester can choose a component of the current model being displayed, instead of requesting the entire model. Attribute data is generated in response to a request resulting from use of the attribute list. The buffer data is the current model. *

connection_info (data flow) =
[connection_request
 | socket_closed
].
* There are two types of data that affect the GRIM widget's socket with its client. The first is a connection request which establishes the socket, and the second alerts the widget that the client is no longer available for communication. *

connection_request (data flow) =
a request sent by client to server for socket-based communications

data_for_application (data flow) =
new_buffer_data .
* The application gets information for the current model

from the common data store and displays it in the current buffer. *

data_for_client1 (data flow) =
header
+ client_in_data .
* These are the standard data items sent by the server to clients in the integration system. *

data_for_client2 (data flow) =
header
+ client_in_data .
* These are the standard data items sent by the server to clients in the integration system. *

data_for_widget (data flow) =
header
+ w_data .
*data_for_widget is in one of two forms, a request for connection with the GRIM widget from a client, or data from the client which will be displayed in the widget for selection by the user *

data_from_application (data flow) =
[application_name
| attribute_list
| attribute_data
| buffer_data
].
* Information accessible from the application. This includes a list of data attributes that other clients can request, data from the application which corresponds to items in the attribute list, or data which describes the current buffer being displayed. *

data_from_client1 (data flow) =
header
+ client_out_data .
* client_out_data includes all data sent from the client, including requests for data and connection, as well as responses to requests from another client. *

data_from_client2 (data flow) =
header
+ client_out_data .
* client_out_data includes all data sent from the client - including requests for data and connection, as well as responses to requests from another client. *

data_to_application (data flow) =

```
[ x-formed_buffer_data
  | x_formed_attribute_data
].
```

* This is data which has been changed in format and sent to the application. Data of this form is data requested from a client in the system and data requested from an attribute list of a client in the system. *

```
decision_data (data flow) =
[ request_current_buffer
+ application_name
+ activate_exchange
]
|[ request_attribute_list
+ application_name
]
| attribute_list_choice .
```

* There are three actions the user may perform:
Request the current buffer from application 2 (as named in application name), request a list of data attributes from application 2 (again, as named in application name), and request data based on a choice from the attribute list supplied by application 2 (again, again, as named in application name). *

```
event (data flow) =
[ request_buffer_event
| request_attribute_list_event
| request_attribute_item_event
].
```

* These define possible user actions *

```
header (data flow) =
size_in_bytes
+ maj_opcode
+ min_opcode .
```

* The header is the first chunk of information read from the socket by the receiving process. Using the size_in_bytes the process can expect how much data will follow. The major and minor opcodes are used by the receiving process to "handle" the incoming data. *

```
initialize_client (data flow) =
[ connection_request
| request_for_xchg_app_list
].
```

* These are requests used to set up the client who has requested a connection with the server. *

```
list_choice (store) =
```

current_list_item .

* Denotes the list item chosen from the client attribute list. *

list_update_info (data flow) =

add_or_delete

+ list_name .

* Based on add_or_delete the list name which follows will be put in the exchange_list of the widget, or taken out. *

new_buffer_data (data flow) =

[x-formed_buffer_data

| x-formed_attribute_data

].

* This is the data which will revise the model displayed in the application which receives it into its common data area. *

relay_data (data flow) =

attribute_list .

* This is the attribute list of application client. It does not need to be transformed in any way, merely relayed by the server to the client which requested this data. *

request_app_out (data flow) =

[connection_request

| request_for_xchg_app_list

| request_buffer

| request_attribute_list

| request_attribute_data

].

* Characterizes the types of requests the client will send to the server. *

request_application_info (data flow) =

[application_name_request

| buffer_data_request

| attribute_list_request

| attribute_data_request

].

* These requests apply only to the state of the application.*

request_attribute_item_callback (data flow) =

attribute_list_callback

+ attrib_list_item_callback

+ ok_callback .

* The callback which will request an item from an attribute list is a combination of the request for the list, a list item, and then an ok... to go ahead with the request. The sequence used to invoke the callback is attribute list toggle callback function, followed by an item from the list

generated, then an OK button callback function to acknowledge choice *

request_attribute_item_event (data flow) =
attribute_list_choice

request_attribute_list_callback (data flow) =
attribute_callback
+ list_callback .
* This request is a combination of a choice for the attribute list and the name (from list) of the responding application. The attribute callback corresponds to a request for the attribute list, and the list callback corresponds to the an application name from the selection list being chosen. *

request_attribute_list_event (data flow) =
request_attribute_list
+ responder .
* This event is generated after a series of actions. Order is important. *

request_attribute_list_from_application2 (data flow) =
header
+ responder .
* The responder name is necessary to allow the server to determine the socket location of the application responding to the request. The header is used to instruct the receiving process how to evaluate the data that follows. *

request_buffer_callback (data flow) =
receive_callback
+ list_callback
+ activate_callback
+ active_ok_callback .
* This request consists of a receive data choice followed by a choice of application from list. In order to effect the request, the activate choice is necessary followed by an ok.
The receive callback is triggered by a toggle, the list callback by selection of an application name, the activate callback by a toggle and the ok button by a pushbutton callback. *

request_buffer_event (data flow) =
request_current_buffer
+ responder
+ activate_exchange .
* This event occurs due to a sequence of actions on the part of the user. Order is important. *

request_buffer_from_application2 (data flow) =
header
+ responder .
* The responder name is necessary for the server to determine on which socket the responding application is connected. The header will allow the client to receive the signal from the GRIM widget and evaluate it properly. *

request_conn_client_name (data flow) =
request from the GRIM widget to the client application for his identifying name.

respond_to_attribute_list_choice (data flow) =
header
+ attribute_list_choice .
* This is actually a request based on the choice of an element of the attribute list supplied by another client. *

responder (data flow) =
application_name .
* name of the application which will respond to a given request.*

responding_application_name (store) =
responder .
* Contains the name of the application who will respond to the request generated by the user. *

response_app_out (data flow) =
[server_destined
| transfer_data
| relay_data
] .
* Server destined data is the name of the client application, while transform data is data from the client application that must be changed into a format which is compatible with the client that requested that data from the sending application. Relay data is data that does not need to be transformed, merely needs a new header added to its message. *

server_destined (data flow) =
[application_name
| close_sock
] .
* The name of client applicatio or the client socket which has just recently been closed to communication. *

socket_data_structure (store) =

socket_descriptor
+ client_name .

* This information is stored for each client connected to a server. It allows the socket descriptor to be cross-referenced by client name. *

transfer_data (data flow) =

[buffer_data
| attribute_data
].

* Both of these types of data need to be changed from the format sent by client application 1 into that of the requesting client application (application 2). *

w_data (data flow) =

[connection_info
| widget_display_data
].

* w_data is in one of two forms, a request for connection with the GRIM widget from the owning application, or data from the client application which will be displayed in the widget for selection by the user. *

widget_action (data flow) =

header
+ widget_request .

* Based on choices the user makes at the GRIM widget interface, the widget sends requests for either the current buffer (geometric data, text, graphs, etc) from application 2, a list of attributes that application 2 can supply, or the actual transaction item from the attribute list. The client name request is generated by the widget when a new socket connection is opened. *

widget_display_data (data flow) =

[owning_application_name
| exchange_application_list
| attribute_list
| list_update_info
].

* widget_displayed_data can be the name of the client who owns the GRIM widget (displayed above the selection list in the widget), a list of applications with whom the GRIM's owner can request data (displayed in the selection list), or a list of attributes sent from an application with whom the GRIM's owner can request data. The attribute list is used to request specific pieces of data (as

defined by the application who will supply them). *

```
widget_in_data (data flow) =  
  [ list_of_exchange_applications  
    | attribute_list  
    | list_update  
  ].
```

```
widget_out_data (data flow) =  
  [ connection_request  
    | client_name  
    | exchange_application_list  
    | attribute_list  
    | list_update_info  
  ].
```

* The application that will own the GRIM widget, sends out a connection request ONE TIME to establish a communication link. The widget needs the remaining four pieces of information to display choices on the widget for the user. *

```
widget_request (data flow) =  
  [ widget_request_forward_server  
    | widget_request_client_info  
  ].
```

* Some of the data passed to the client by the server is meant to be forwarded on to the server. Other data are to be supplied by the client itself. *

```
widget_request_client_info (data flow) =  
  request_client_app_name .
```

* At the present time this is the only information the widget needs from the client application. *

```
widget_request_forward_server (data flow) =  
  [ request_buffer_from_application_named  
    | request_attribute_list_from_application_named  
    | attribute_list_choice  
  ].
```

* Requests for buffer and attribute list must be accompanied by the name of the responding application, while attribute list choice knows which application responds because of the preceding request for the attribute list. The application named in the request for buffer and attribute list is also known as the responder. *

APPENDIX B: DATA FLOW DIAGRAMS / P-SPECS

This appendix contains a description of each data flow diagram followed by the data flow diagrams and process specifications themselves. It is important to remember that the data entities used in the data flow diagrams are often not found in the structure charts of the components they represent. This is because there are data in the integration system which have no way of being named. What this means is that often, requests for data are determined simply by the major and minor operation codes contained in the message header, and the response to the request is initiated immediately upon resolution of the opcodes. In view of the fact that request data may not be explicitly defined in terms of variable names, the data flow diagrams represent requests in conceptual terms; in other words, a name is given to request data that does not translate to the component structure charts. This process of giving names to data which do not explicitly appear elsewhere is what is meant by the term "conceptual data". The purpose of using conceptual data is to describe the types of messages being passed in the integration system. Messages in the system all are preceded by a header block which is used by the receiving process to locate the module in the event handler which will receive any further data, or will produce data as a direct consequence of the header. Please note that italicized variables indicate data flows.

Context Diagram

The first diagram created is called the context diagram. The context diagram is defined as the top-level of a hierarchical set of data flow diagrams. It represents the entire system in terms of a single process, shown as bubble 0. The diagram is used to delineate the scope of the analysis and define the system in terms of its inputs and outputs. The context diagram, in conjunction with the data flow diagrams derived from

it, enables the integration system designer to identify the major transactions of a system in terms of inputs and outputs.

The context diagram of the distributed integration solution shows three data terminators labeled application 1, application 2, and user. It is important to mention here that only two applications have been used in these diagrams in order to simplify the data model. However, the rules developed for two applications can be extended to cover n applications.

As shown, each application has data it can send to and receive from the integrated system. The *data_from_application* is defined as the application's name, a list of data attributes (often called the Client Attribute List (CAL)) that other clients can request, data corresponding to those attributes, and data representing the current buffer. The *data_to_application* can be either buffer data that were requested from a client in the system, or attribute data (data supplied in response to a choice from the attribute list) from a client in the integrated system. Note here that it is possible that an application could send data to the integration server to be transformed in some manner and then sent back. In this scenario, an application would essentially request data from itself. This could be useful for applications whose source is not available, since a function external to the application would appear to be part of the application.

The user of the integrated system is presented with *choice_data* which is defined as a sequence of operations the user must perform. The user has the ability to request the current buffer from an application in the system, the attribute list from an application in the system, or data which correspond to an item selected from that attribute list. As a result of evaluating the choice data and taking action, the user creates *decision_data*

which is transmitted to the integrated system. Decision data indicate that the user wants to request buffer data from a specified application, request a list of attributes from a specified application, or request data based on an attribute list previously requested.

DFD 0 - Integrated System

As was described in Chapter 4, the term "client" refers to a CAD application, AP/SOCK Interface, and GRIM widget, while "application" refers only to the CAD application itself. The term "client interface" implies the combination of the GRIM interface and the AP/SOCK Interface.

In DFD (Data Flow Diagram) 0, we see the integration server with two client interfaces connected. These clients interfaces correspond to applications 1 and 2, shown in the context diagram, as they are embodied in the integrated system. Both interfaces are identical; therefore, it suffices to only explain one of them indepth. For this purpose we choose client interface 1.

Client interface 1 receives three inputs: *data_from_application*, *decision_data*, and *data_for_client1*. *Data_for_client1* consists of a *header* and *client_in_data*.

Client_in_data can be either data bound for the GRIM widget or data bound for the application. The data output from the client interface 1 is *data_to_application*, *choice_data*, or *data_from_client1*. *Data_from_client1* consists of a *header* and *client_out_data*. *Client_out_data* includes all data sent from the client, including requests for data and connection, as well as responses to requests from the integration server or another client. The integration server process shows the transition of *data_from_client1* to *data_for_client2* and *data_from_client2* to *data_for_client1*.

DFD 1 - Client Interface

The client interface data flow diagram shows the components of the client application which enable the CAD application to interface with the user and the integration server. These two components are the AP/SOCK interface and the GRIM widget. In the diagram, each component is represented by its own process node.

The socket interface (AP/SOCK) is responsible for receiving *data_for_client1* and determining if the data should be given to the application or to the GRIM widget. These decisions are indicated by the data flows *data_for_application* and *data_for_widget*. The data pertaining to the application are placed in the common data buffer of the socket interface and the application. This will enable access to the data by the CAD application. The data which pertain to the GRIM widget, such as *widget_display_data* or *connection_info*, are gathered and sent to the widget for action. The widget will take some of that data and display it, thus producing *choice_data* for the user. When the user performs an action based on the choices, he produces *decision_data* which is sent back to the widget, where it may be combined with other information to form *widget_action*. An action for the widget is defined as requests for data or information generated by the user, or requests for information about the owning client application. In the socket interface process node, either the input from the widget or *data_from_application* will be used to create *data_from_client1*. The data produced by the application are stored in common with the socket interface, such as buffer data, or the CAD application's name. The data leaving the socket interface node are all data the client can produce, including requests for data or responses to requests.

DFD 1.1 - GRIM Widget

This data flow diagram represents the component of the client which enables the user to interface with the integrated system. *Data_for_widget* is evaluated to be either *widget_display_data* or *connection_info* from the owning client application. Since the GRIM widget is essentially a server in its design, the term client application is warranted. Client application implies the CAD application and its system interface (AP/SOCK) for which the GRIM supplies the user interface. *Widget_display_data* is transformed into a format for presentation to the user as choice data. *Connection_info* is evaluated and a request is generated which is sent to the client application. When *decision_data* is generated by the user, it is transformed into an *event*. An event or a request for client application name constitutes a *widget_action*.

DFD 1.1.1 - Evaluate Decision

This process generates an event based on decision data received as input. Decision data breaks down into the following: *application_name*, *buffer_request*, *activate_data_exchange*, *request_attribute_list*, and *attribute_choice*. The events generated are *request_buffer_event*, *request_attribute_list_event*, and *request_attribute_item_event*.

DFD 1.1.2 - Determine Widget Action

Based on an event, a callback function is activated. Callbacks are functions which allow a widget to perform an action when prompted. The execute action process produces *widget_action* from *req_client_app_name* or *callback*. *Widget_action* is defined as *header* and *widget_request*.

DFD 1.1.2.1 - Determine Callback Function

Based on the event (*request_buffer_event*, *request_attribute_list_event*, or *request_attribute_item_event*), a callback function is activated.

DFD 1.1.2.2 - Execute Action

Based on the callback function, a specific header is constructed for the data. The header (bubble 1) is concatenated with a list item identifier and/or the responding application name. The resulting information is *request_buffer_from_application2*, *request_attribute_list_from_application2*, or the *attribute_list_choice*. The header created in bubble 3 is used to *request_client_app_name*.

DFD 1.1.3 - Socket Interface

Data_for_widget is stripped of its header and becomes *w_data*. *W_data* is further determined to be either *widget_display_data* or *connection_info*.

DFD 1.1.3.2 - Create and Display Widget

List_update_info tells the process whether to add or delete a selection list item. The selection list is the one which lists clients from which data can be requested. An item added to this list is first converted into a motif string. Other kinds of data converted into a motif string include the owning application's name, initial members of the selection list (shown by *exchange_application_list*), and attribute list items. Motif strings are string definitions recognized by the Motif toolkit which can be displayed by a widget.

DFD 1.1.4.4 - Display in Widget

A Motif list item is evaluated as to whether it should be added to the attribute list or the selection list. The respective lists are created or added to. The attribute list consists of items representing choices that correspond to data attributes as defined by an

application. The selection list consists of client names. In the widget, toggle buttons represent the ability to request buffer data or the attribute list from another client in the integrated system. By combining the selection list choices and the toggle buttons, choice data for the user are produced.

DFD 1.1.5 - Evaluate Connection Info

Connection information could be either the indication that an active socket has been terminated (*socket_closed*) or it could be a request for connection from the owning client application. When a socket is closed, its identifier is deleted from the list of active sockets kept by the GRIM. This allows the GRIM to detect when its owning application has been terminated. As a result, it terminates since the CAD application no longer requires its services. On the other hand, when a new connection is received, the client application's name is requested for use as displayed information in the widget.

DFD 1.1.5.3 - Accept Connection

The occurrence of a connection request creates a new socket dedicated to the client application. This socket descriptor is stored in a socket data structure. The fact that a new connection has been requested generates the request for the client application's name.

DFD 1.1.5.3.2 Request Name

The *new_connection* triggers the construction of a header which will, in effect, request the name of the client application which has just connected to the GRIM.

DFD 1.2 - Socket Interface

The socket interface is found in the AP/SOCK which is used as an intermediary to the integration server by the GRIM and the CAD application. In the data flow representation of the interface, *data_for_application1* consists of a *header* and *client_in_data*. *Client_in_data* is either *widget_in_data* or *application_in_data*.

Widget_in_data consists of a list of exchange clients (for incorporation in the selection list), an attribute list, or a list update. A list update is an instruction, destined for the widget, to add or delete a client name from the selection list. *Widget_in_data* or *owning_application_name* comprise *widget_display_data*, which when orred with *connection_info*, produces *w_data*. When a header is added to *w_data*, it becomes *data_for_widget*.

Application_in_data is *request_application_info* or *new_buffer_data*.

Request_application_info originated from another client in the integrated system or the integration server. *New_buffer_data* has been sent by a client in the integrated system in response to a buffer data request generated by this client. *Request_application_info* is *application_name_request*, *buffer_data_request*, *attribute_list_request*, or *attribute_data_request*. These requests can be filled from *common_buffer*. The responses to these requests take one of three possible forms: *server_destined*, *transfer_data*, or *relay_data*.

Data destined for the integration server include the application's name (for use by the integration server when cross-referencing socket descriptors by client name) or a closed socket signal. When a client socket is terminated, the server realizes that the client is no longer connected to the integrated system. Transfer data are data in the format of

the responding client. These data must be either transformed or translated at the integration server so that they can be sent to the receiving client in the integrated system. Buffer data and attribute data are considered to be transfer data. Relay data is the attribute list from the client which must be transferred to the requesting client in the integrated system.

Response_app_out or *request_app_out* form the data flow defined as *application_out_data*. *Request_app_out* is *widget_request_forward_server* or *initialize_app*. *Widget_request_forward_server* are those requests generated by the user at the GRIM widget which must be relayed by the integration server to another client in the integrated system for response. These requests include *request_buffer_from_application_named*, *request_attribute_list_from_application_named*, or *attribute_choice*. *Initialize_app* is either a request for connection or a request for a list of the clients to be placed in the widget's selection list. These two requests are only generated during the start-up phase of the integration client. When a *header* is added to *application_out_data*, it is transformed into *data_from_client1*.

Widget action is split into *widget_request_forward_server* and *widget_request_client_info*. The latter is defined as any information the GRIM needs to obtain from the client application about the client application. At this point the only information of this type is the client application's name. In response to this request, the client application sends its name as an identifier to the GRIM.

DFD 1.2.1 - Evaluate Data Header

In this process, the data destined for application 1 are stripped of the message header and the data flow *client_in_data* emerges. These data are then determined to be either *widget_in_data* or *application_in_data*.

DFD 1.2.6 - Evaluate Data Header

This process performs the same operation as DFD 1.2.1, except that the input and output data are different. In any case, the process strips the data flow called *widget_action* of its header, thereby forming *widget_request*. The output data are then determined to be either *widget_request_forward_server* or *widget_request_client_info*.

DFD 2 - Integration Server

This diagram contains two identical but inverse processes. We will only treat the process on the left, the right-hand process being implied from the other's description. In the process described, data incoming from client 1 are transformed into data bound for client 2.

DFD 2.1 - Evaluate Data from Client 1

Data from client 1 are evaluated as either *widget_request_forward_server*, *initialize_client*, or *response_app_out*. The data forwarded on to the integration server from client 1's GRIM widget must be transmitted by the server to client 2. These data include buffer requests, attribute list requests, or attribute list choices which need a data response. For the first two requests listed, the integration server uses the responding application's names to locate the socket descriptor of that client at the server. This is possible because the integration server has a data structure containing socket descriptors which are cross-referenced with the client name.

The data which an application sends in response to a request (*response_app_out*) are one of three types: transfer data, server-destined data, or data which need to be relayed in their current state to another client. Once again, transfer data are data that the server may need to modify or manipulate in order to send them on to the client who requested that data. Buffer data and attribute data are of this type. Server-destined data are sent by a client to the integration server where they are received and evaluated. Examples of server-destined data are client names which are placed in a socket data structure to allow cross-referencing with socket descriptors. It is also socket connection information such as a connection request or a client termination notification. If a client has closed communications with the server (in the server's view it has terminated), the socket descriptor and name corresponding to that client are removed from the socket data structure. All clients in the integrated system who previously had the ability to request data from the dead client are informed to delete its name from their selection lists. Data which the client must obtain from the server in order to complete its initialization phase are called *initialize_client*. This data flow is defined as either a connection request or a request for a list of clients from which the client can request data. In response to a connection request, the integration server requests the client application's name. Alternatively, if the request from the client is for a list of exchange clients, the server compiles a list. Either the list of exchange clients, an attribute list from another client in the system (*relay_data*), or a list update message (add or delete client from selection list) forms *widget_in_data*.

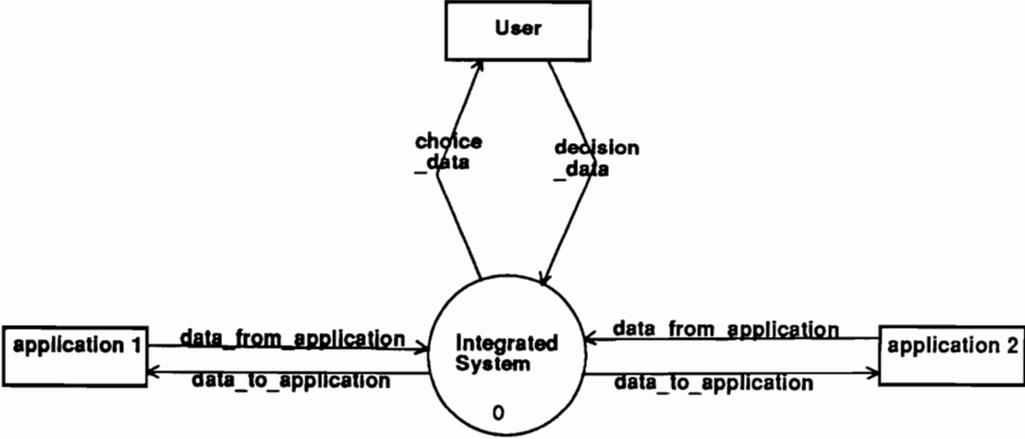
When a header is appended to *widget_in_data*, *new_buffer_data*, *application_name_request*, or *widget_request_forward_server*, the result is *data_for_client2*.

DFD 2.1.1 - Evaluate Data by Header

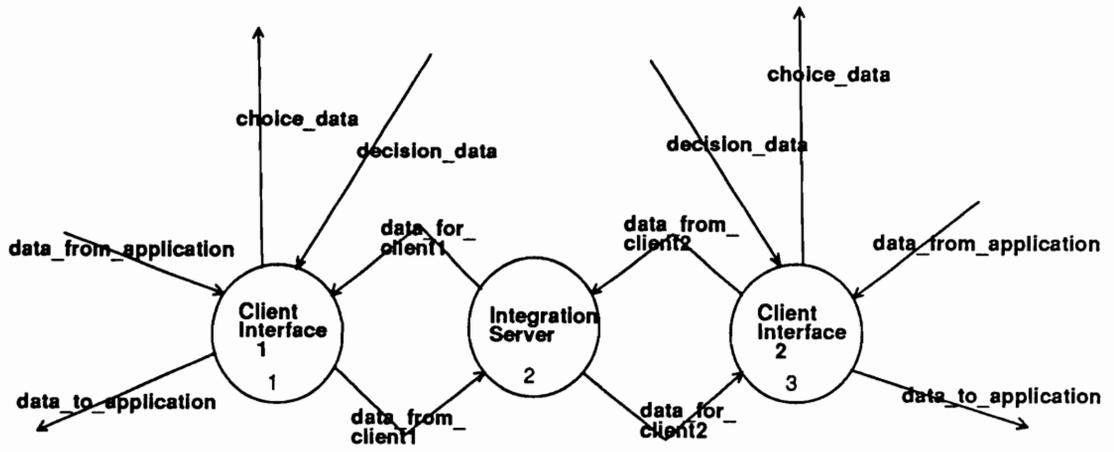
Data incoming from client 1 are stripped of the *header* and *client_out_data results*.

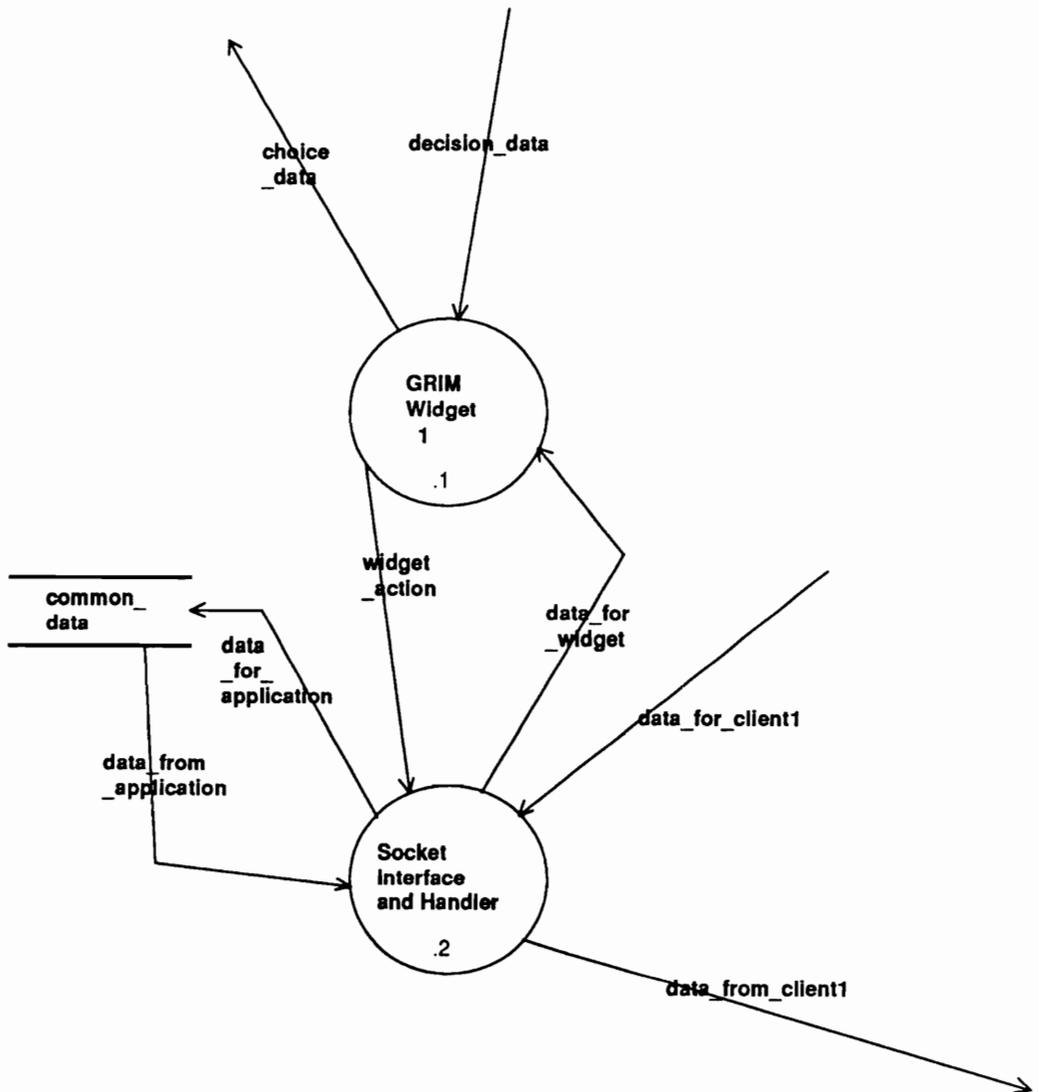
The data flow is then split into *widget_request_forward_server*, *response_app_out*, or *initialize_app*.

Context-Diagram;6
Integration_System

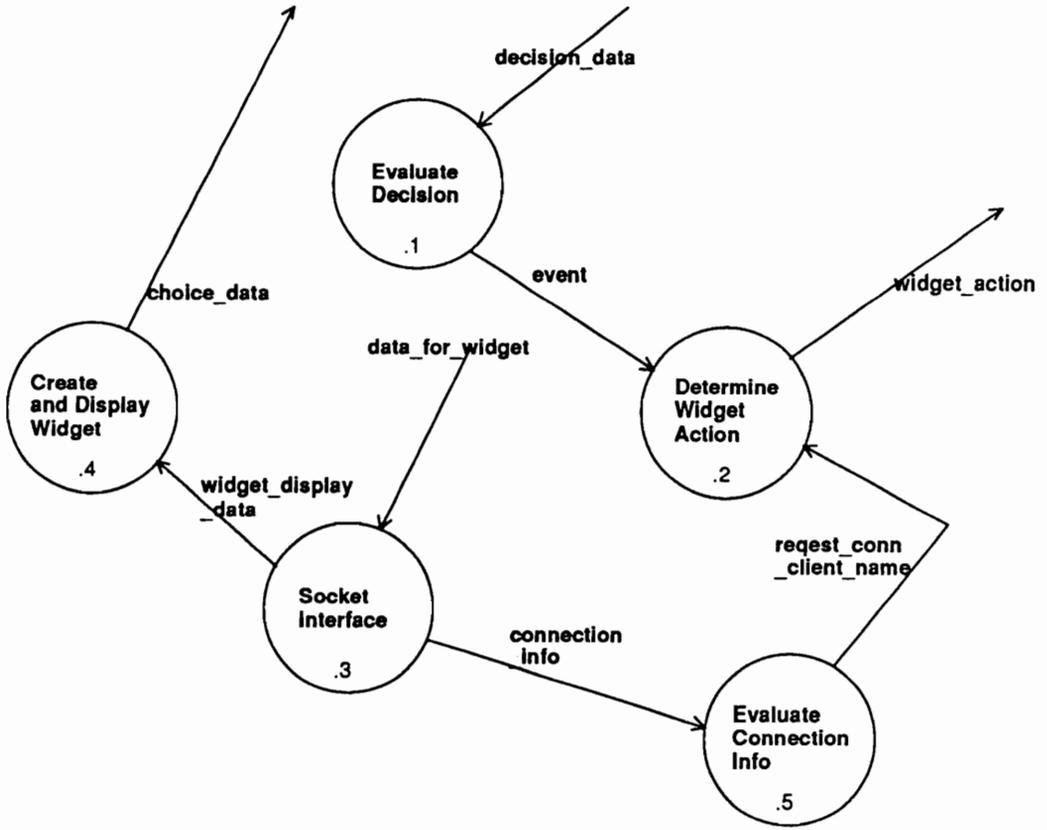


0:3
Integrated System

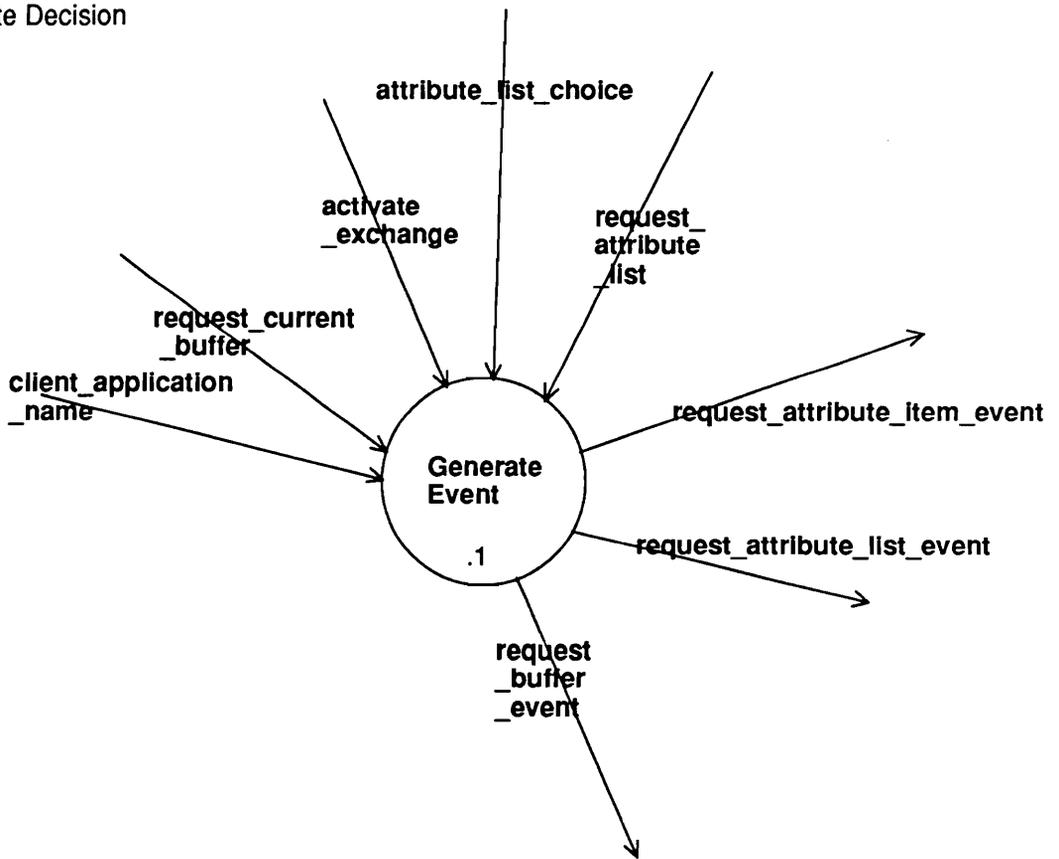




1.1;1
GRIM Widget 1



1.1.1;4
Evaluate Decision



NAME: 1.1.1.1;1

TITLE: Generate Event

INPUT/OUTPUT:

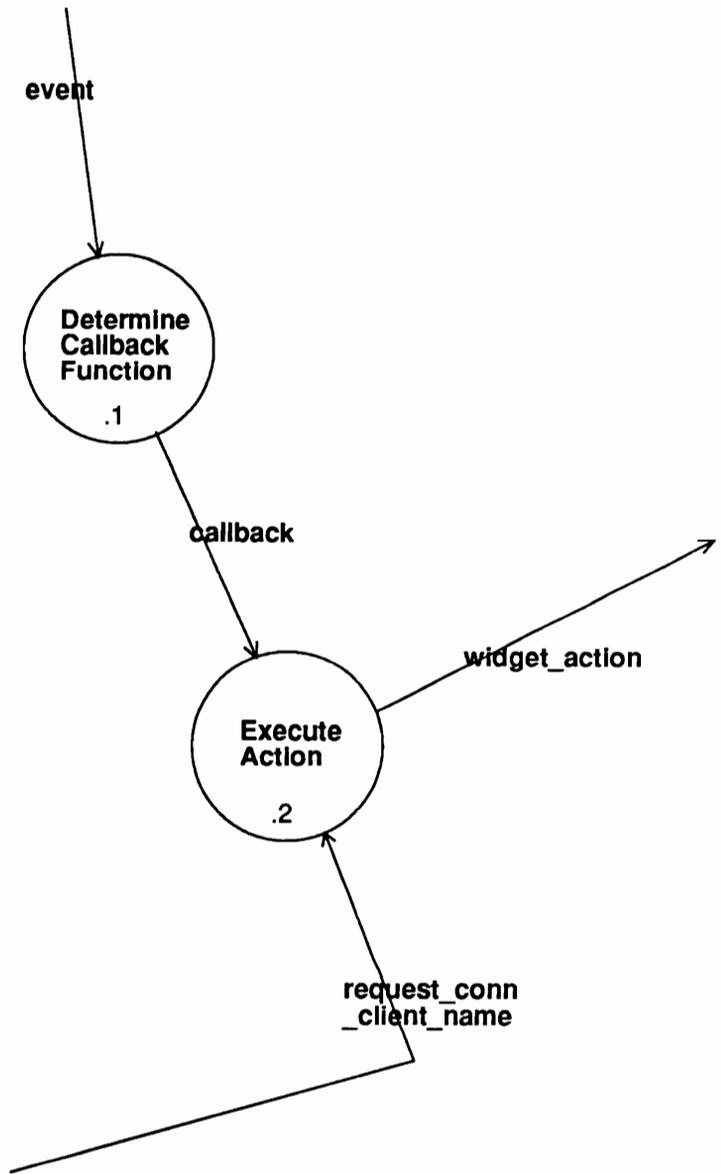
request_current_buffer : data_in
request_attribute_list : data_in
request_buffer_event : data_out
activate_exchange : data_in
request_attribute_list_event : data_out
request_attribute_item_event : data_out
attribute_list_choice : data_in
client_application_name : data_in

BODY:

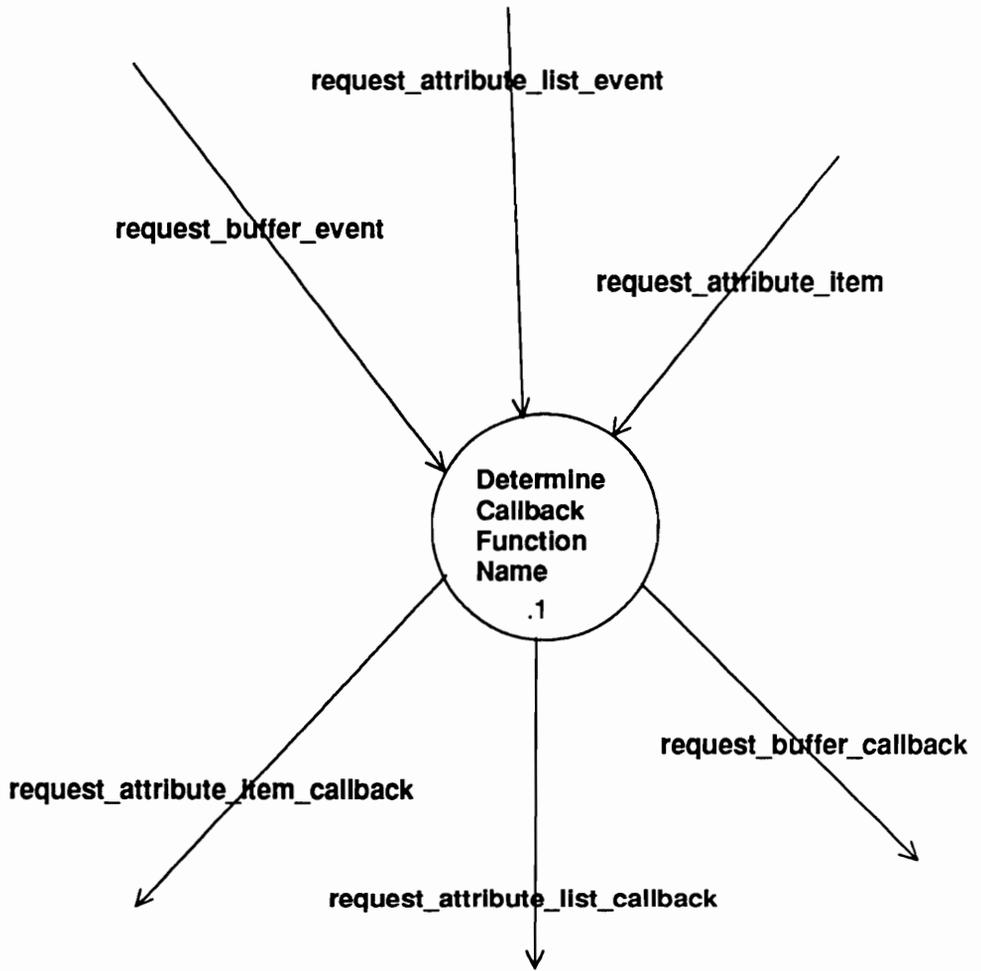
This p-spec is best described using a decision table.

	request buffer event list	request attribute attribute event item	request attribute event
client_application_name	Y	Y	Y
request_buffer_event	Y	N	N
activate_exchange	Y	N	N
attribute_list_choice	N	N	Y
request_attribute_list	N	Y	Y

1.1.2;4
Determine Widget Action



1.1.2.1;2
Determine Callback Function



NAME: 1.1.2.1.1;1

TITLE: Determine Callback Function Name

INPUT/OUTPUT:

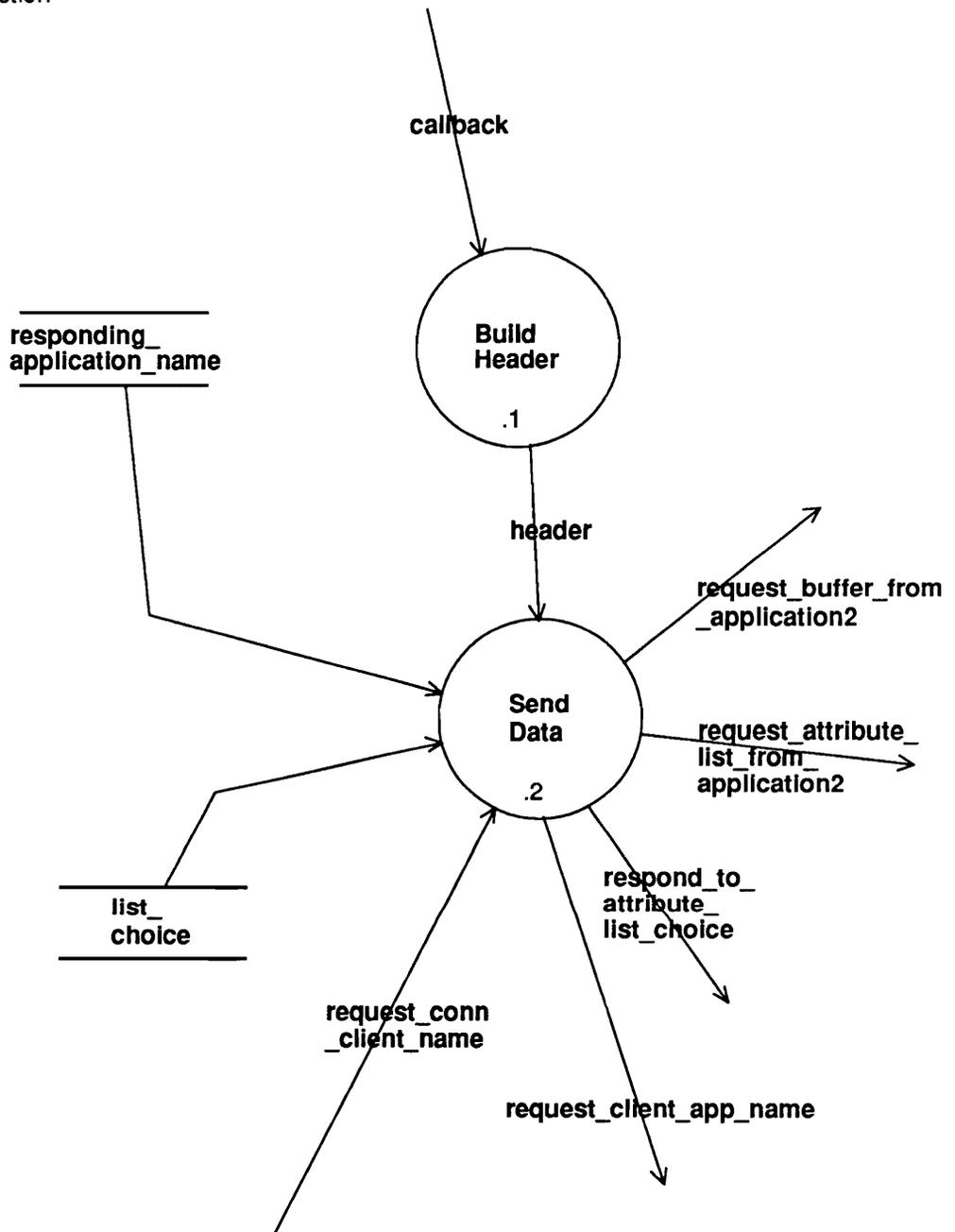
request_buffer_event : data_in
request_attribute_list_event : data_in
request_attribute_item : data_in
request_attribute_item_callback : data_out
request_attribute_list_callback : data_out
request_buffer_callback : data_out

BODY:

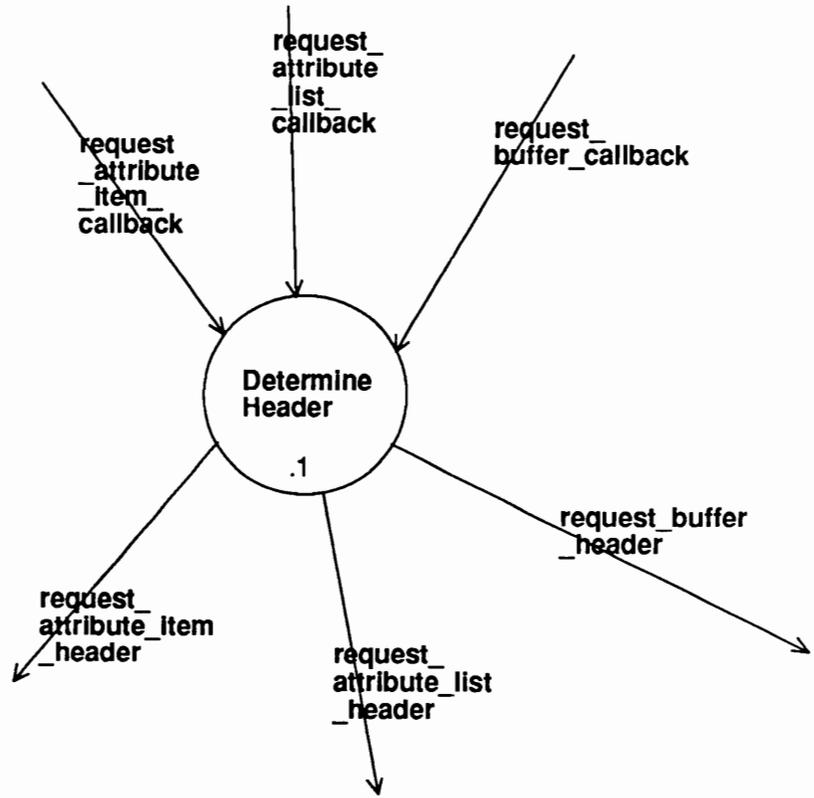
This process is best described by a decision table.

	request_ attribute_ item_ callback	request_ attribute_ list_ callback	request_ attribute_ list_ callback	buffer_ callback
request_buffer_event	N	N	N	Y
request_attribute_list_event	N	Y	Y	N
request_attribute_item	Y	N	N	N

1.1.2.2;7
Execute Action



1.1.2.2.1;1
Build Header



NAME: 1.1.2.2.1.1;1

TITLE: Determine Header

INPUT/OUTPUT:

request_attribute_item_callback : data_in

request_attribute_list_callback : data_in

request_buffer_callback : data_in

request_attribute_item_header : data_out

request_attribute_list_header : data_out

request_buffer_header : data_out

BODY:

This process is best described using a decision table

		request_ attribute_ item_header	request_ attribute_ list_header	request_ attribute_ list_header	buffer_ header
request_attribute_item_callback	Y		N		N
request_attribute_list_callback	N		Y		N
request_buffer_callback	N		N		Y

NAME: 1.1.2.2.2;1

TITLE: Send Data

INPUT/OUTPUT:

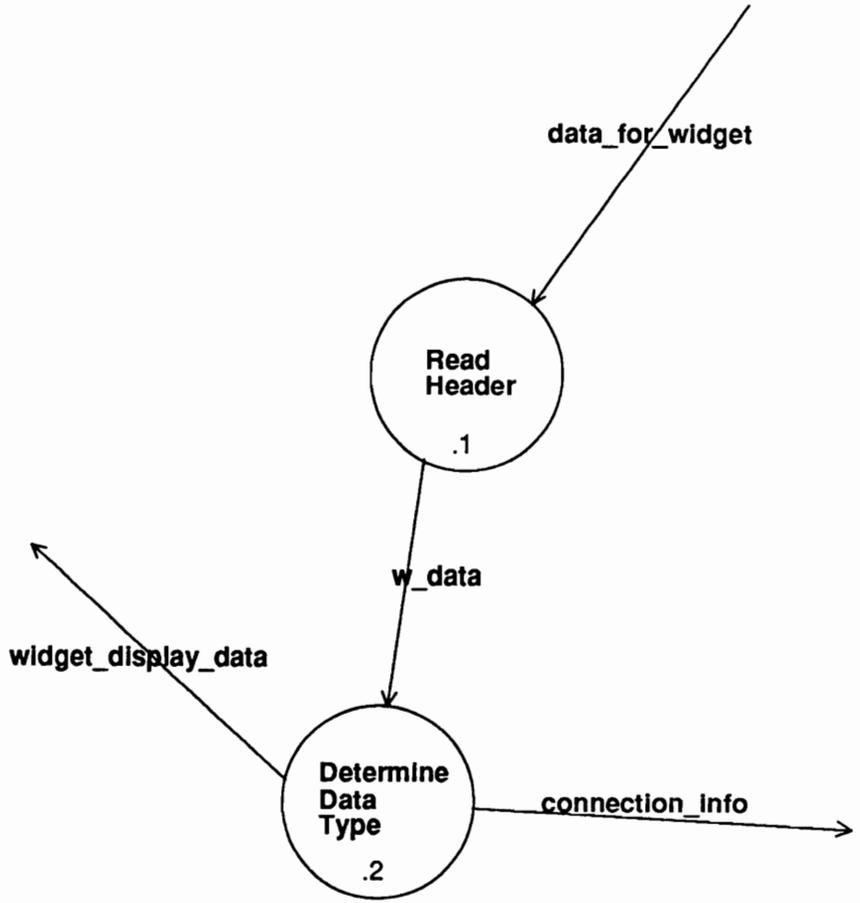
responding_application_name : data_in
header : data_in
request_buffer_from_application2 : data_out
request_attribute_list_from_application2 : data_out
respond_to_attribute_list_choice : data_out
list_choice : data_in
request_conn_client_name : data_in
request_client_app_name : data_out

BODY:

This process is best described using a decision table.

	request_ buffer_ from_ application2	request_ attribute_ list_from_ application2	respond_to attribute_ list_ choice	request_ client_ application _name
responding_application_name	Y	Y	Y	Y
header	Y	Y	Y	Y
list_choice	N	N	Y	N
request_conn_client_name	N	N	N	Y

1.1.3;2
Socket Interface



NAME: 1.1.3.1;1

TITLE: Read Header

INPUT/OUTPUT:

data_for_widget : data_in

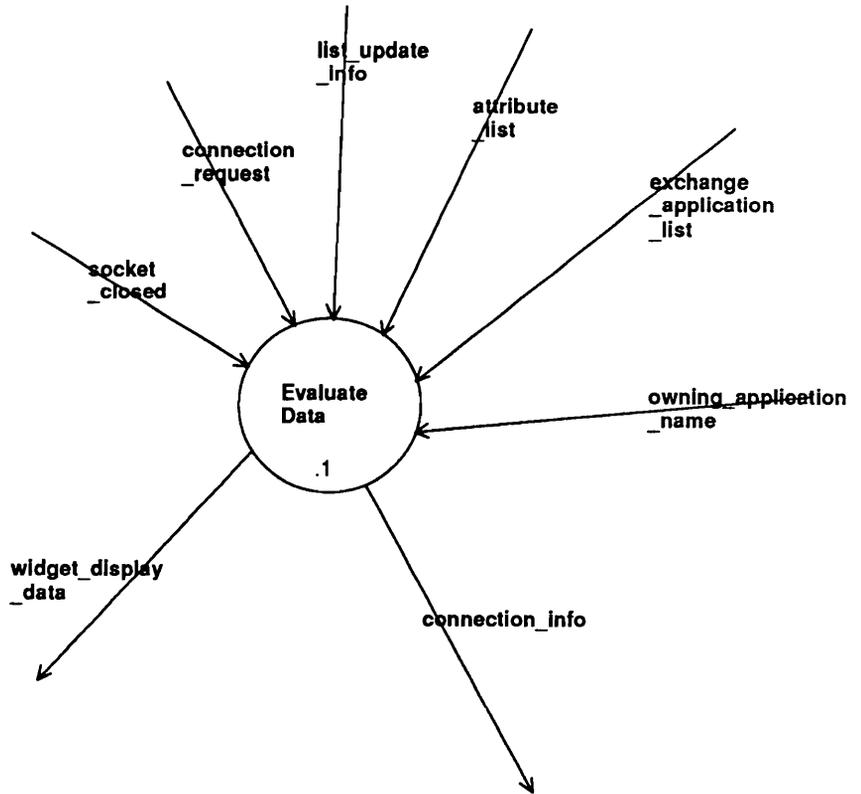
w_data : data_out

BODY:

This process is described in structured English.

READ header.

1.1.3.2;2
Determine Data Type



NAME: 1.1.3.2.1;1

TITLE: Evaluate Data

INPUT/OUTPUT:

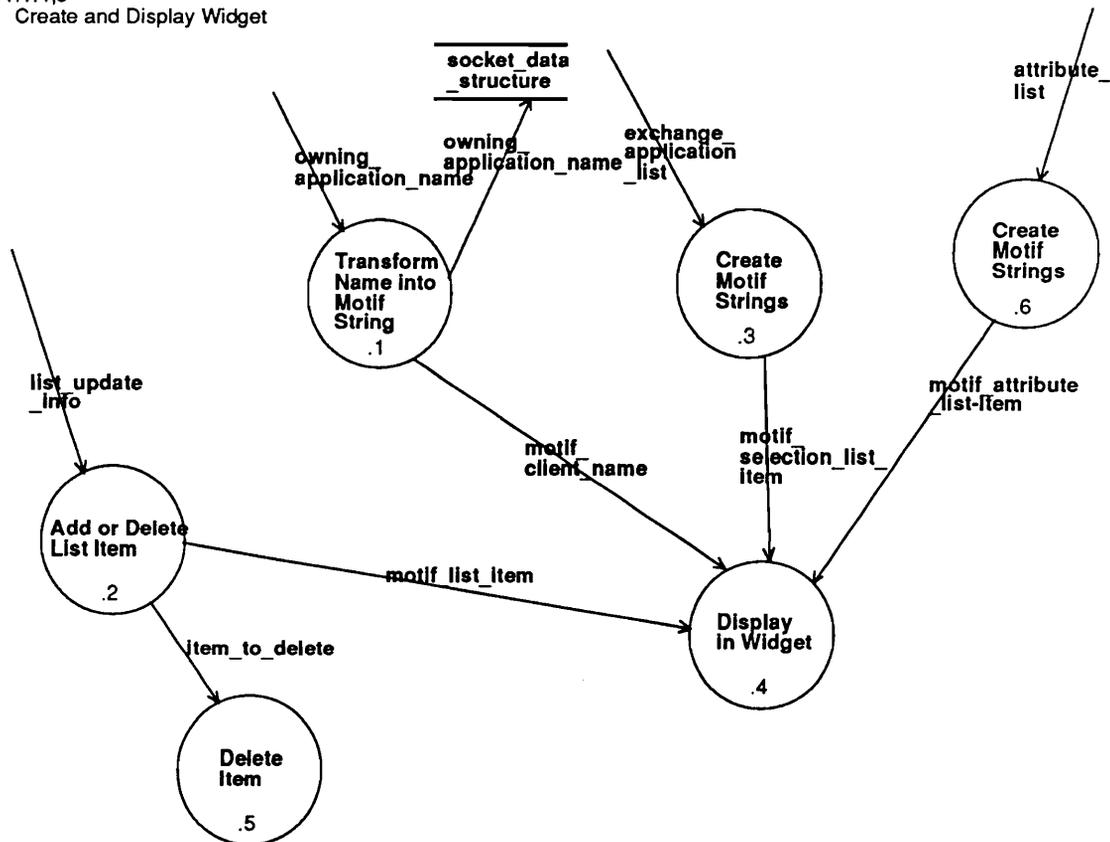
socket_closed : data_in
connection_request : data_in
list_update_info : data_in
attribute_list : data_in
exchange_application_list : data_in
owning_application_name : data_in
widget_display_data : data_out
connection_info : data_out

BODY:

This process is best described using a decision table.

	widget_ display _data	connection_ info
socket_closed	N	Y
connection_request	N	Y
list_update_info	Y	N
attribute_list	Y	N
exchange_application_list	Y	N
owning_application_name	Y	N

1.1.4;5
Create and Display Widget



NAME: 1.1.4.1;1

TITLE: Transform Name into Motif String

INPUT/OUTPUT:

owning_application_name : data_in
owning_application_name : data_out
motif_client_name : data_out

BODY:

This process is explained in structured English

READ owning_application_name

TRANSFORM owning_application_name into motif_client_name

PUT owning_application_name into socket_data_structure

NAME: 1.1.4.2;1

TITLE: Add or Delete List Item

INPUT/OUTPUT:

list_update_info : data_in
item_to_delete : data_out
motif_list_item : data_out

BODY:

This process is described using structured English

IF list_update_info = ADD

 READ list_item

 TRANSFORM list_item into motif_list_item

ELSE

 READ item_to_delete

END IF

NAME: 1.1.4.3;1

TITLE: Create Motif Strings

INPUT/OUTPUT:

exchange_application_list : data_in

motif_selection_list_item : data_out

BODY:

This process is described using pre/post conditions

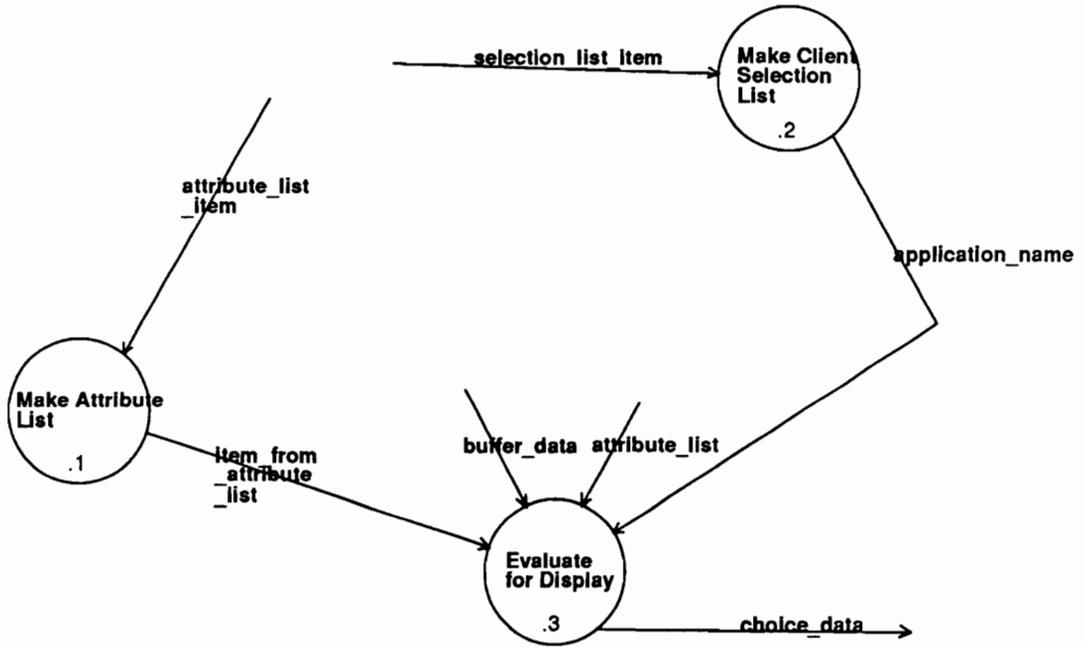
Precondition:

data elements exchange_application_list occur

Postcondition:

produces Motif string motif_selection_list_item for every member of the list.

1.1.4.4;4
Display in Widget



NAME: 1.1.4.4.1;1

TITLE: Make Attribute List

INPUT/OUTPUT:

attribute_list_item : data_in

item_from_attribute_list : data_out

BODY:

This process is described using structured English

DO WHILE attribute_list_item occurs

 ADD attribute_list_item to the attribute list

 DISPLAY attribute_list_item in attribute list and produce item_from_attribute_list

END DO

NAME: 1.1.4.4.2;1

TITLE: Make Client Selection List

INPUT/OUTPUT:

selection_list_item : data_in

application_name : data_out

BODY:

This process is described using structured English

DO WHILE selection_list_item occurs

 ADD selection_list_item to the selection list

 DISPLAY selection_list_item in selection list and produce application name.

END DO

NAME: 1.1.4.4.3;1

TITLE: Evaluate for Display

INPUT/OUTPUT:

item_from_attribute_list : data_in

buffer_data : data_in

attribute_list : data_in

application_name : data_in

choice_data : data_out

BODY:

This process is described using a decision table.

	choice _data	choice _data	choice _data
application_name	Y	Y	N
buffer_data	Y	N	N
attribute_list	N	Y	N
item_from_attribute_list	N	N	Y

NAME: 1.1.4.5;1

TITLE: Delete Item

INPUT/OUTPUT:

item_to_delete : data_in

BODY:

This process is described using structured English

DELETE item_to_delete from the widgetqs selection list.

NAME: 1.1.4.6;1

TITLE: Create Motif Strings

INPUT/OUTPUT:

attribute_list : data_in

motif_attribute_list-Item : data_out

BODY:

This process is described using pre/post conditions.

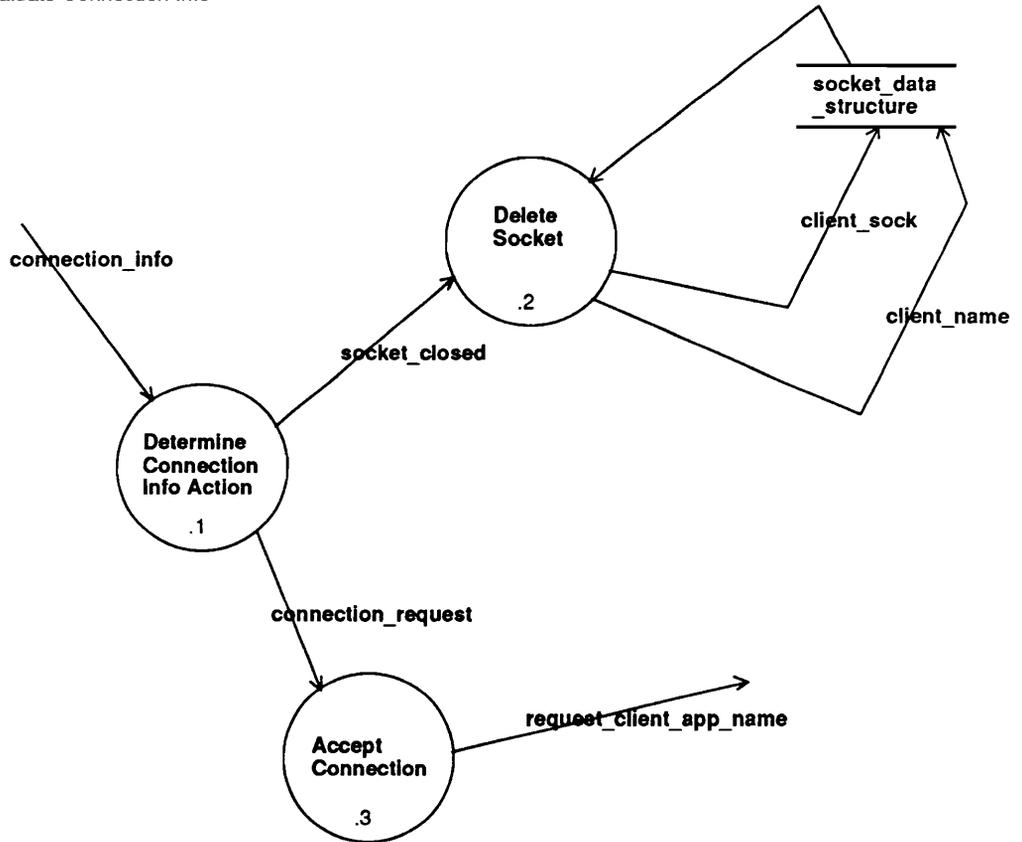
Precondition:

data elements attribute_list occur.

Postcondition:

produce Motif string motif_attribute_list_item for every member of the list.

1.1.5;5
Evaluate Connection Info



NAME: 1.1.5.1;1

TITLE: Determine Connection Info Action

INPUT/OUTPUT:

connection_info : data_in

socket_closed : data_out

connection_request : data_out

BODY:

This process is described using pre/post conditions

Precondition:

occurrence of connection_info

Postcondition:

either socket_closed or connection_request is produced

NAME: 1.1.5.2;1

TITLE: Delete Socket

INPUT/OUTPUT:

socket_closed : data_in

client_sock : data_out

socket_data_structure : data_in

client_name : data_out

BODY:

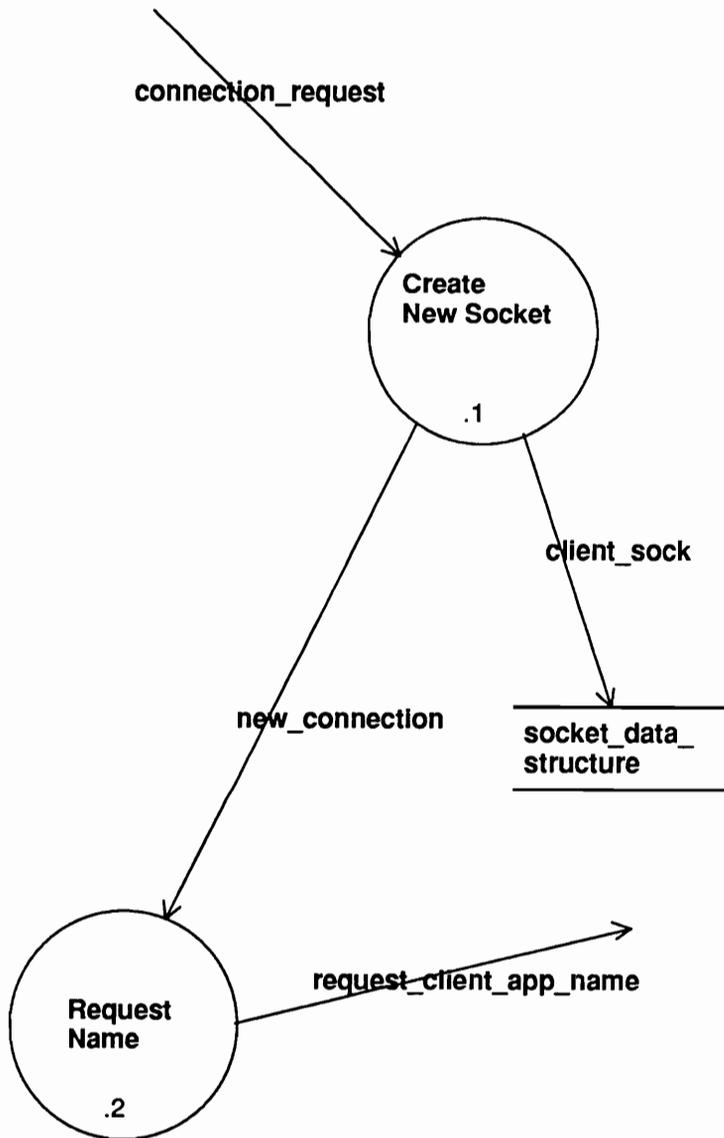
This process is described using structured English

DELETE client_sock from socket_data_structure

DELETE client_name corresponding to client_sock from socket_data_structure

NAME: 1.1.5.3.1;1

1.1.5.3;2
Accept Connection



NAME: 1.1.5.3.1;1

TITLE: Create New Socket

INPUT/OUTPUT:

connection_request : data_in

client_sock : data_out

new_connection : data_out

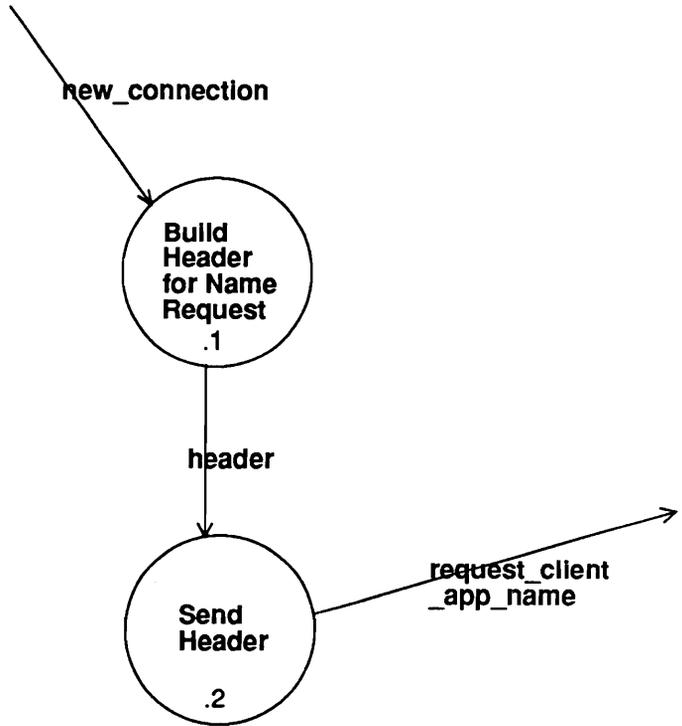
BODY:

This process is described using structured English.

CREATE new socket called client_sock

PUT client_sock into socket_data_structure

1.1.5.3.2;3
Request Name



NAME: 1.1.5.3.2.1;1

TITLE: Build Header for Name Request

INPUT/OUTPUT:

new_connection : data_in

header : data_out

BODY:

This process is described using pre/post conditions

Precondition:

data element new_connection exists

Postcondition:

produce data header corresponding to new connection message

NAME: 1.1.5.3.2.2;1

TITLE: Send Header

INPUT/OUTPUT:

header : data_in

request_client_app_name : data_out

BODY:

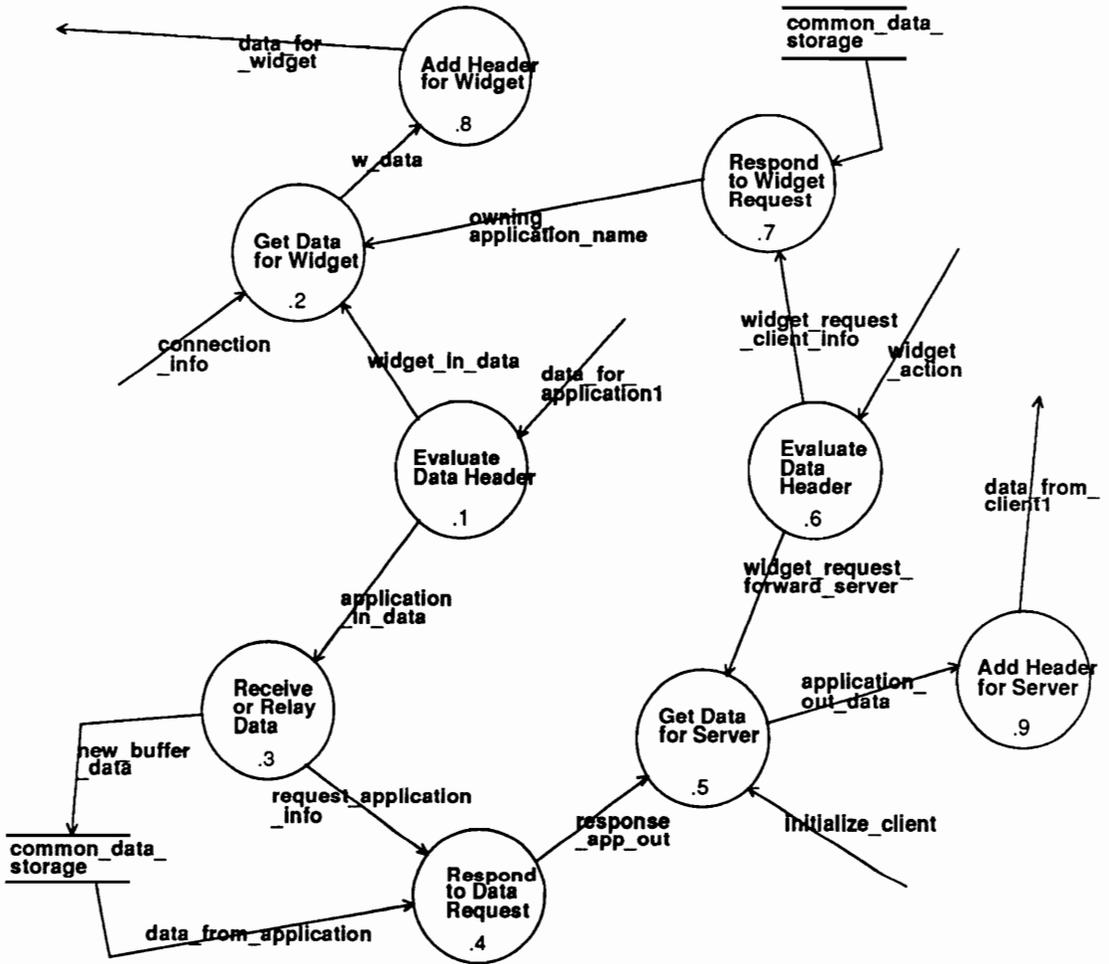
This process is described using pre/post conditions.

Precondition:

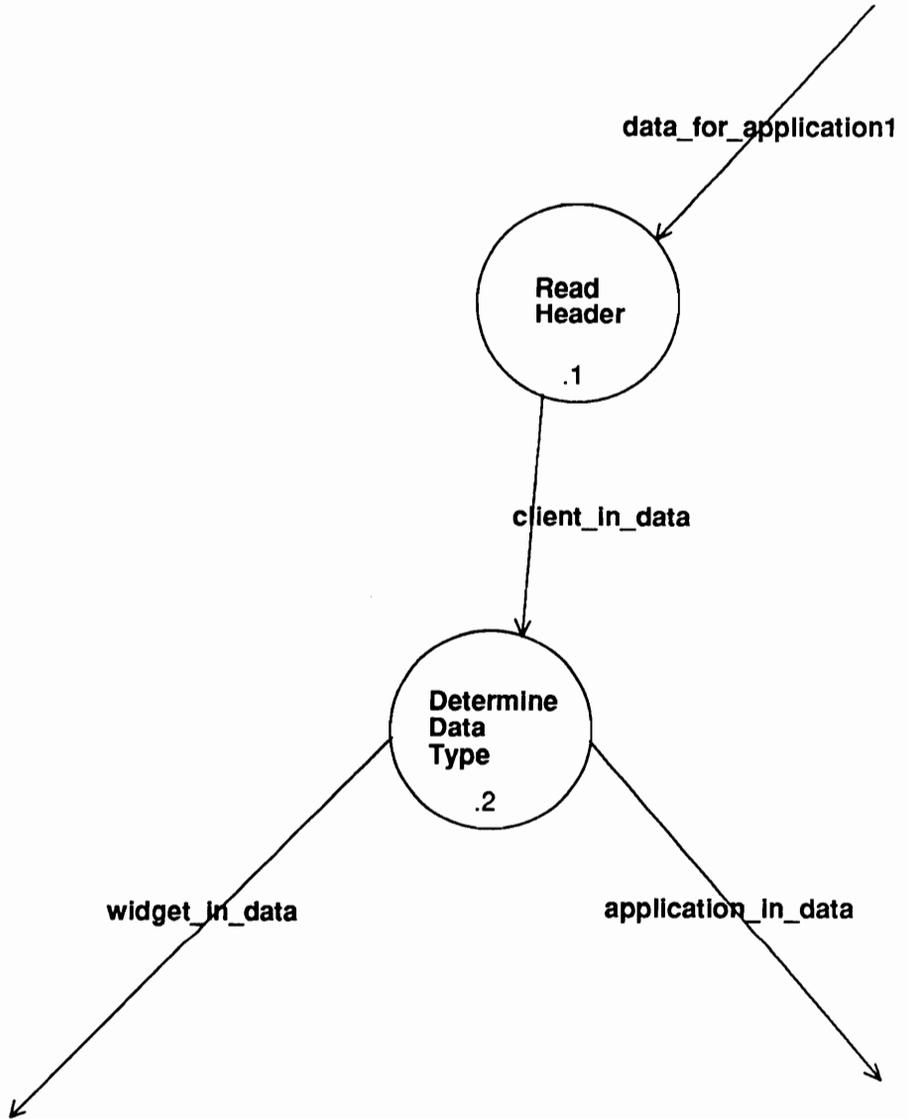
data header occurs

Postcondition:

transmit header which represents request_client_app_name



1.2.1;1
Evaluate Data Header



NAME: 1.2.1.1;1

TITLE: Read Header

INPUT/OUTPUT:

data_for_application1 : data_in

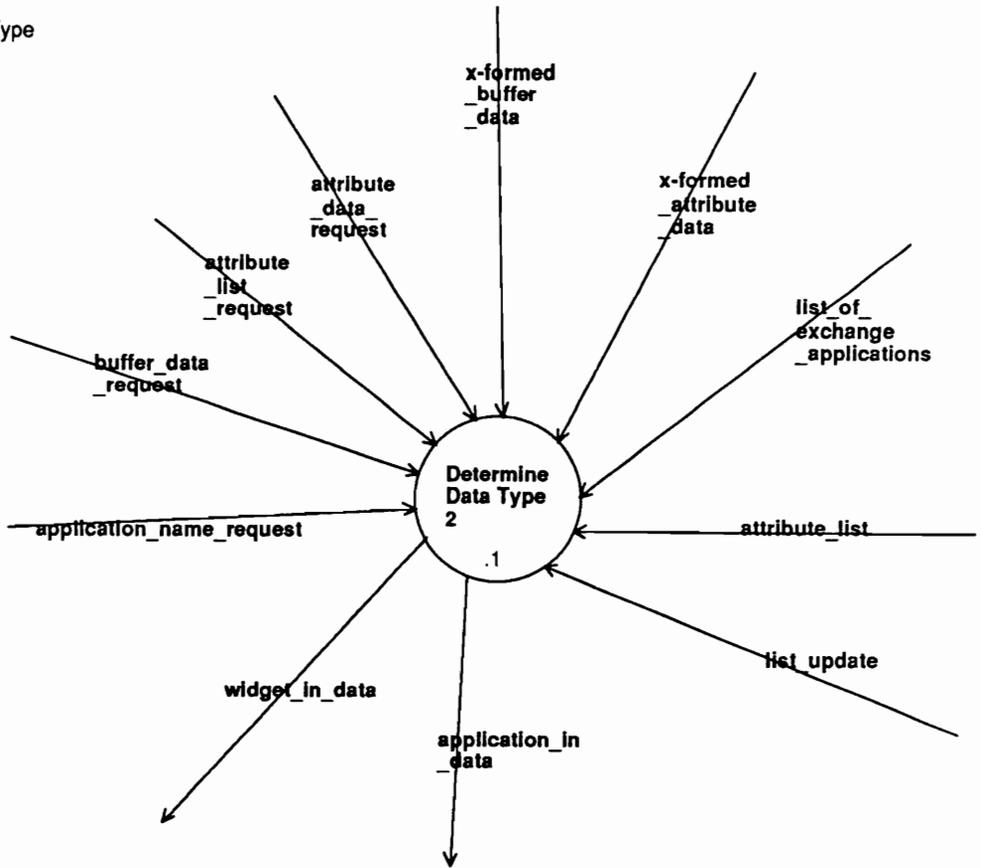
client_in_data : data_out

BODY:

This process is described using structured English

READ header

1.2.1.2;2
Determine Data Type



NAME: 1.2.1.2.1;1

TITLE: Determine Data Type 2

INPUT/OUTPUT:

application_name_request : data_in
buffer_data_request : data_in
attribute_list_request : data_in
attribute_data_request : data_in
x-formed_buffer_data : data_in
x-formed_attribute_data : data_in
list_of_exchange_applications : data_in
attribute_list : data_in
list_update : data_in
widget_in_data : data_out
application_in_data : data_out

BODY:

This process is illustrated using a decision table.

	widget_in _data	application _in_data
application_name_request	N	Y
buffer_data_request	N	Y
attribute_list_request	N	Y
attribute_data_request	N	Y
x-formed_buffer_data	N	Y
x-formed_attribute_data	N	Y
list_of_exchange_applications	Y	N
attribute_list	Y	N
list_update	Y	N

NAME: 1.2.2;1

TITLE: Get Data for Widget

INPUT/OUTPUT:

widget_in_data : data_in
connection_info : data_in
owning_application_name : data_in
w_data : data_out

BODY: This process is described using pre/post conditions.

Precondition:

occurrence of connection_info, widget_in_data, or owning_application_name.

Postcondition:

w_data is produced.

NAME: 1.2.3;1

TITLE: Receive or Relay Data

INPUT/OUTPUT:

application_in_data : data_in
new_buffer_data : data_out
request_application_info : data_out

BODY:

This process is explained through a decision table.

	new _buffer _data	request _application _info
x-formed_attribute_data	Y	N
x-formed_buffer_data	Y	N
application_name_request	N	Y
buffer_data_request	N	Y
attribute_list_request	N	Y
attribute_data_request	N	Y

NAME: 1.2.4;1

TITLE: Respond to Data Request

INPUT/OUTPUT:

data_from_application : data_in

request_application_info : data_in

response_app_out : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

occurrence of request_application_info or data_from_application.

Postcondition:

response_app_out is produced

NAME: 1.2.5;1

TITLE: Get Data for Server

INPUT/OUTPUT:

response_app_out : data_in

application_out_data : data_out

widget_request_forward_server : data_in

initialize_client : data_in

BODY:

This process is described using pre/post conditions.

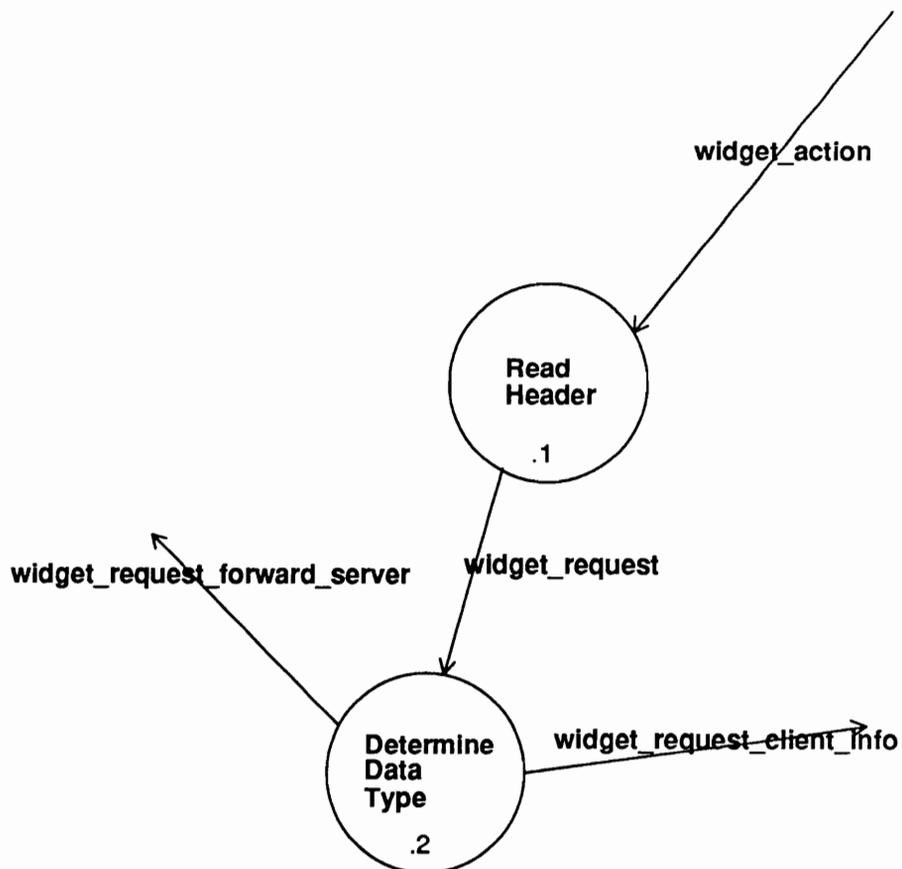
Precondition:

occurrence of response_app_out, widget_request_forward_server, or initialize_client

Postcondition:

application_out_data is produced.

1.2.6;1
Evaluate Data Header



NAME: 1.2.6.1;1

TITLE: Read Header

INPUT/OUTPUT:

widget_action : data_in

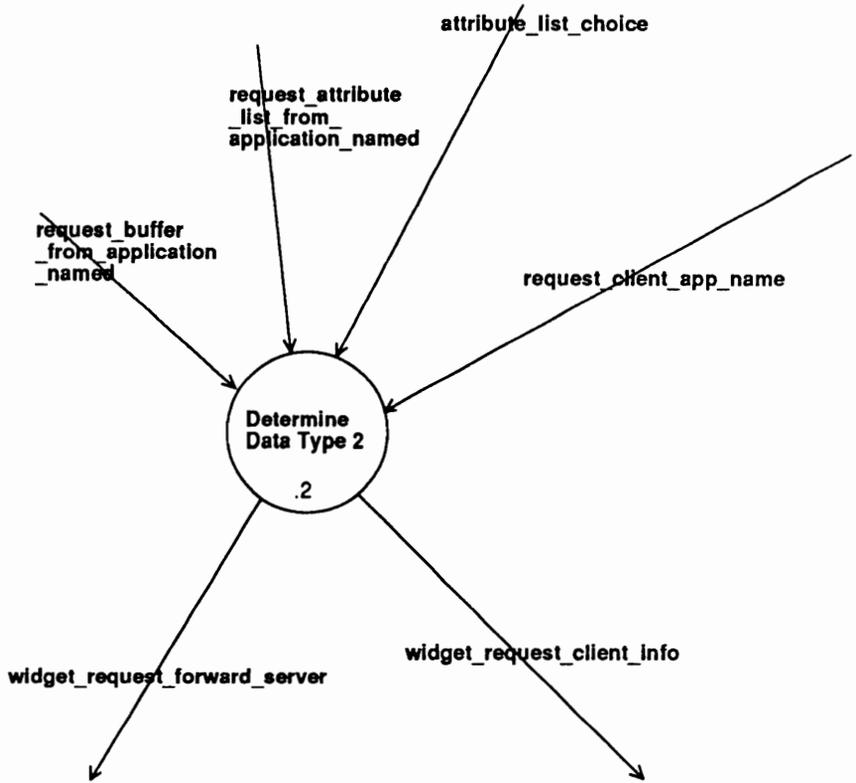
widget_request : data_out

BODY:

This process is described in structured English.

READ header

1.2.6.2;2
Determine Data Type



NAME: 1.2.6.2.1;1

TITLE: Determine Data Type 2

INPUT/OUTPUT:

request_buffer_from_application_named : data_in
request_attribute_list_from_application_named : data_in
attribute_list_choice : data_in
request_client_app_name : data_in
widget_request_forward_server : data_out
widget_request_client_info : data_out

BODY:

This process is described by a decision table.

	widget_request_ forward_server	widget_request _client_info
request_buffer_from_application_named	Y	N
request_attribute_list_from_application_named	Y	N
attribute_list_choice	Y	N
request_client_app_name	N	Y

NAME: 1.2.7;1

TITLE: Respond to Widget Request

INPUT/OUTPUT:

widget_request_client_info : data_in
common_data_storage : data_in
owning_application_name : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

data element widget_request_client_info occurs

Postcondition:

owning_application_name is produced

NAME: 1.2.8;1

TITLE: Add Header for Widget

INPUT/OUTPUT:

w_data : data_in

data_for_widget : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

occurrence of w_data.

Postcondition:

data_for_widget is produced by prepending a header onto w_data

NAME: 1.2.9;1

TITLE: Add Header for Server

INPUT/OUTPUT:

application_out_data : data_in

data_from_client1 : data_out

BODY:

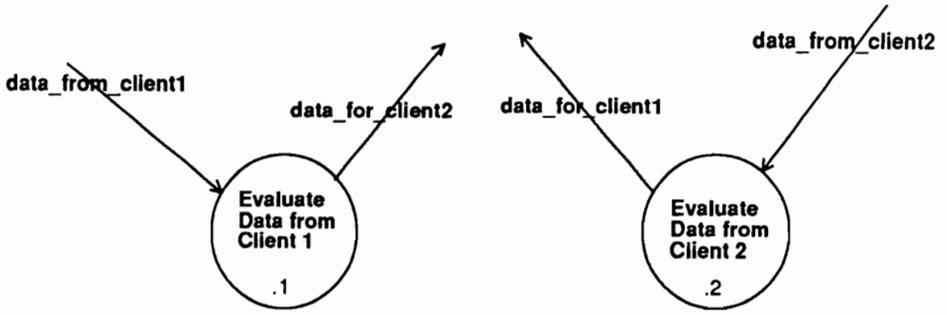
This process is described using pre/post conditions

Precondition:

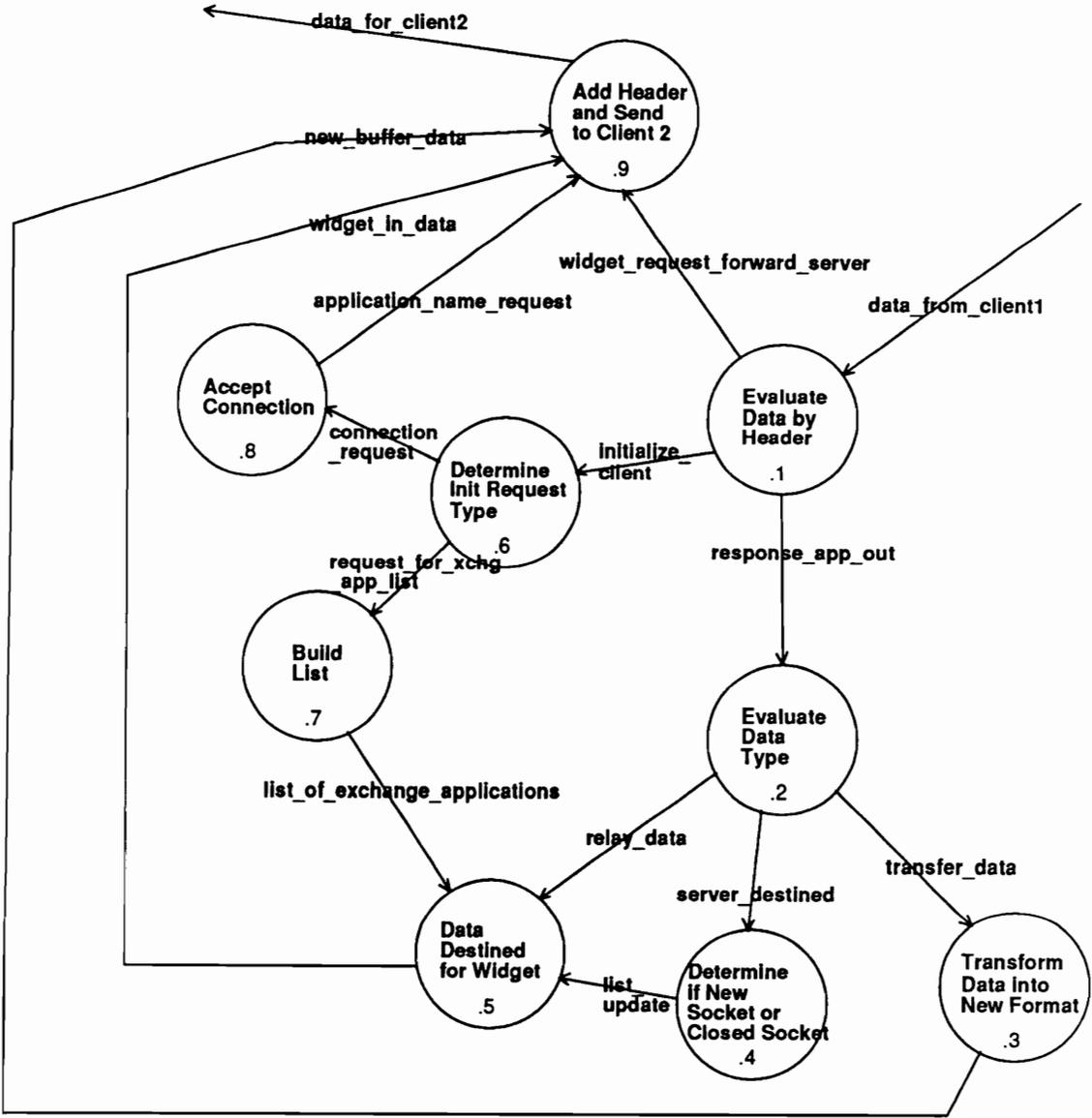
data element application_out_data occurs.

Postcondition:

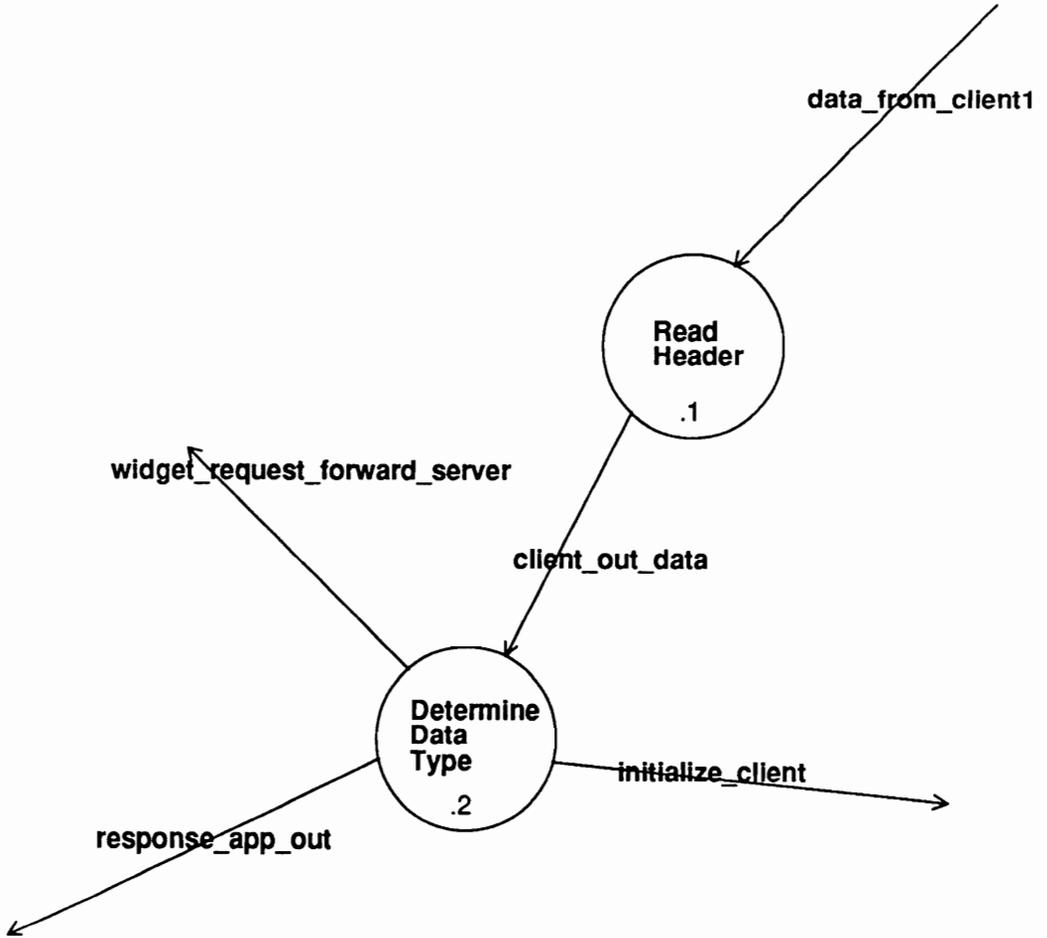
data_from_client1 is produced by prepending a header onto application_out_data.



2.1;4
Evaluate Data from Client 1



2.1.1.2
Evaluate Data by Header



NAME: 2.1.1.1;1

TITLE: Read Header

INPUT/OUTPUT:

data_from_client1 : data_in

client_out_data : data_out

BODY:

This process is described in structured English

READ header

NAME: 2.1.1.2;1

TITLE: Determine Data Type

INPUT/OUTPUT:

client_out_data : data_in

widget_request_forward_server : data_out

initialize_client : data_out

response_app_out : data_out

BODY:

This process is described using a decision table.

	widget_request_ forward_server	initialize _client	response_ app_out
request_buffer_from_application_named	Y	N	N
request_attribute_list_from_application_named	Y	N	N
attribute_list_choice	Y	N	N
server_destined	N	N	Y
transfer_data	N	N	Y
relay_data	N	N	Y
connection_request	N	Y	N
request_for_xchg_app_list	N	Y	N

NAME: 2.1.2;1

TITLE: Evaluate Data Type

INPUT/OUTPUT:

response_app_out : data_in

relay_data : data_out

server_destined : data_out

transfer_data : data_out

BODY:

This process is described by a decision table

	relay_data	server_destined	transfer_data
attribute_list	Y	N	N
application_name	N	Y	N
close_sock	N	Y	N
buffer_data	N	N	Y
attribute_data	N	N	Y

NAME: 2.1.3;1

TITLE: Transform Data into New Format

INPUT/OUTPUT:

transfer_data : data_in

new_buffer_data : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

data element transfer_data occurs.

Postcondition:

transform or translate transfer_data into new_buffer_data.

NAME: 2.1.4;1

TITLE: Determine if New Socket or Closed Socket

INPUT/OUTPUT:

server_destined : data_in

list_update : data_out

BODY:

This process is described in structured English

IF server_destined = application_name

list_update = ADD

ELSE

list_update = DELETE

END IF

NAME: 2.1.5;1

TITLE: Data Destined for Widget

INPUT/OUTPUT:

relay_data : data_in

list_of_exchange_applications : data_in

list_update : data_in

widget_in_data : data_out

BODY:

This process is described by pre/post conditions.

Precondition:

data element relay_data, list_update, or list_of_exchange_applications occurs.

Postcondition:

produce widget_in_data.

NAME: 2.1.6;1

TITLE: Determine Init Request Type

INPUT/OUTPUT:

initialize_client : data_in

request_for_xchg_app_list : data_out

connection_request : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

occurrence of initialize_client.

Postcondition:

production of request_for_xchg_app_list or connection_request based on evaluation of input.

NAME: 2.1.7;1

TITLE: Build List

INPUT/OUTPUT:

request_for_xchg_app_list : data_in

list_of_exchange_applications : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

occurrence of request_for_xchg_app_list.

Postcondition:

production of a list_of_exchange_applications.

NAME: 2.1.8;1

TITLE: Accept Connection

INPUT/OUTPUT:

connection_request : data_in

application_name_request : data_out

BODY:

This process is described using pre/post conditions.

Precondition:

data element connection_request occurs.

Postcondition:

produce application_name_request.

NAME: 2.1.9;1

TITLE: Add Header and Send to Client 2

INPUT/OUTPUT:

widget_request_forward_server : data_in

new_buffer_data : data_in

widget_in_data : data_in

application_name_request : data_in

data_for_client2 : data_out

BODY:

This process is described using pre/post conditions.

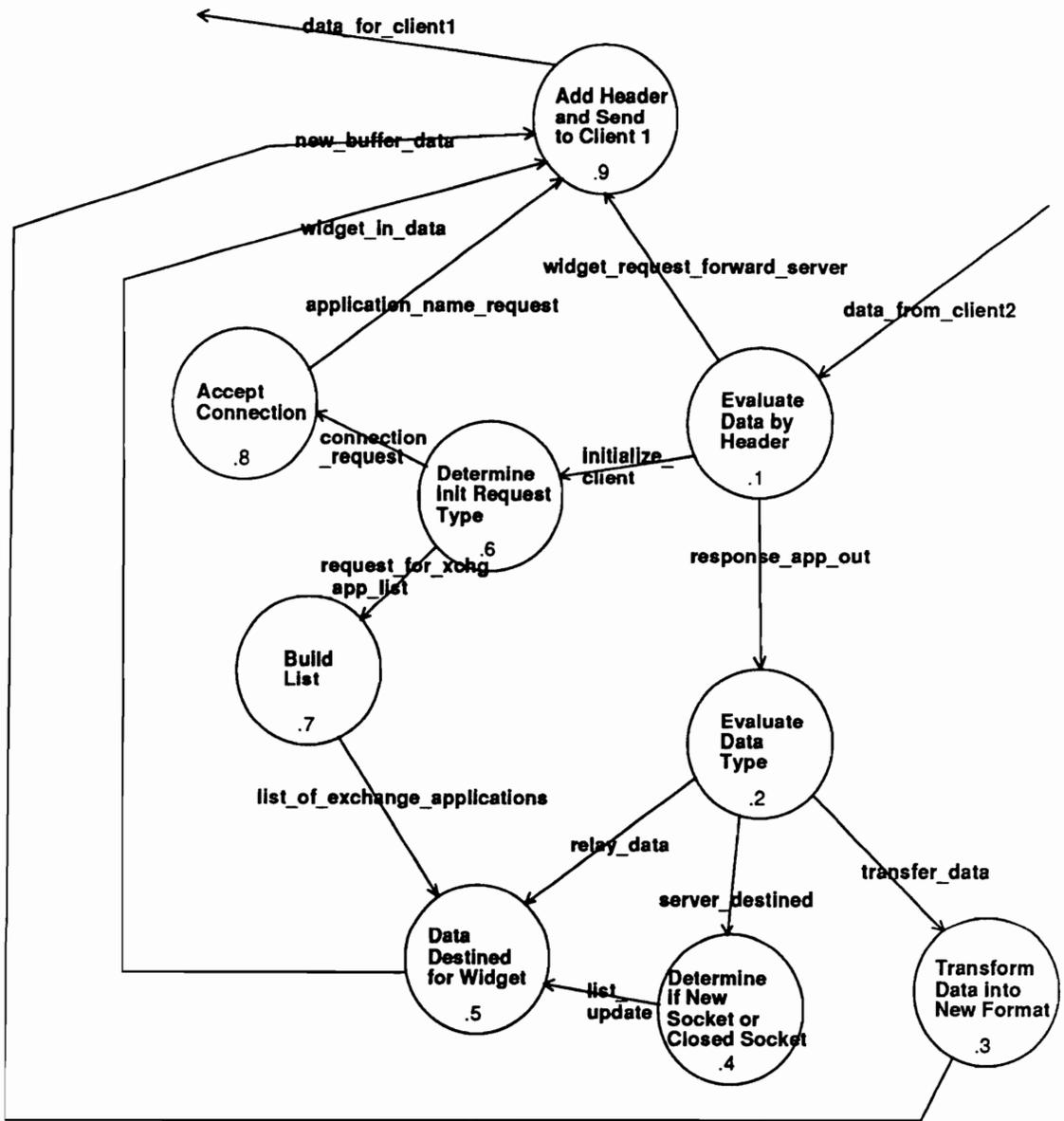
Precondition:

data element new_buffer, widget_in_data, application_name_request,
or widget_request_forward_server occurs.

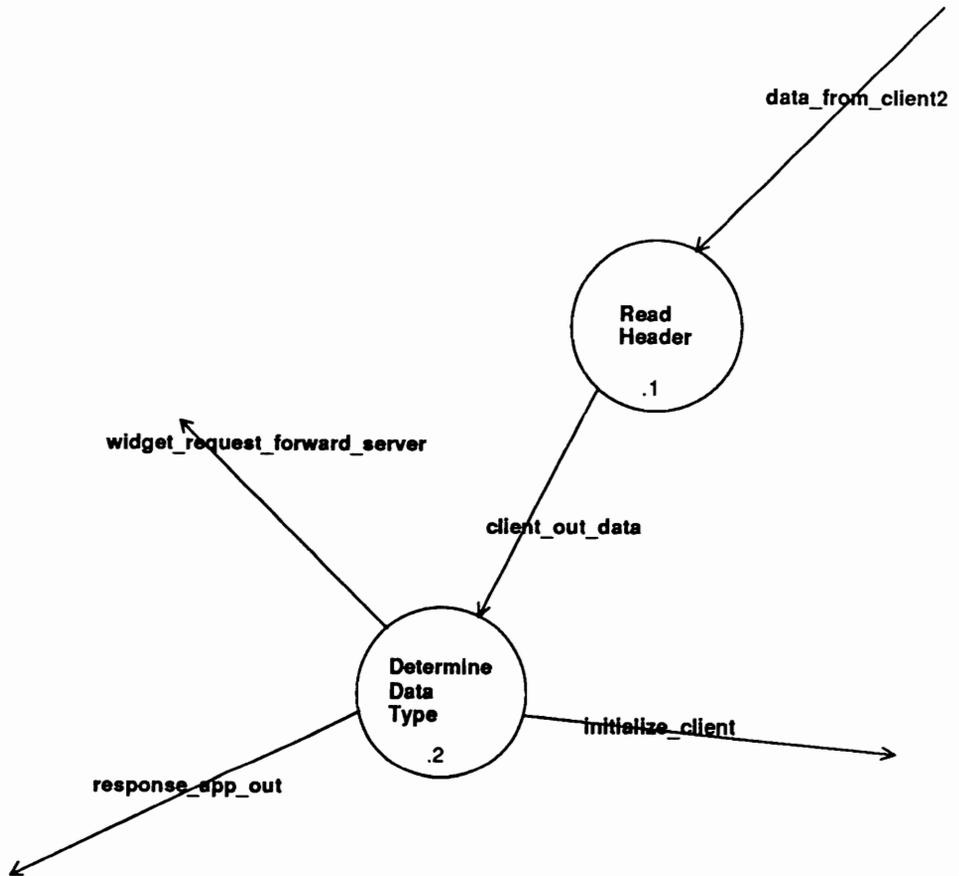
Postcondition:

add header to produce data_for_client2

2.2;5
Evaluate Data from Client 2

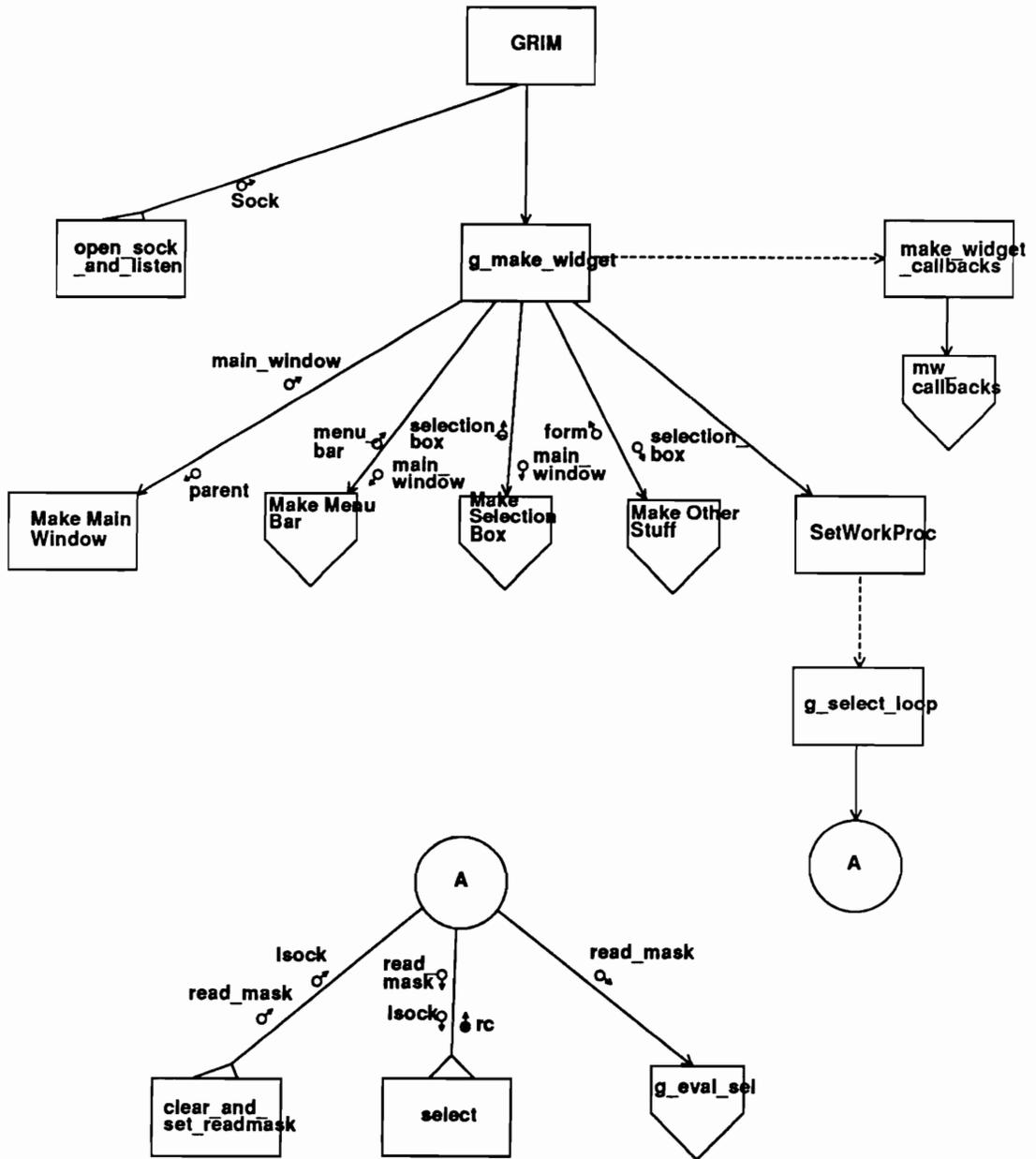


2.2.1;2
Evaluate Data by Header

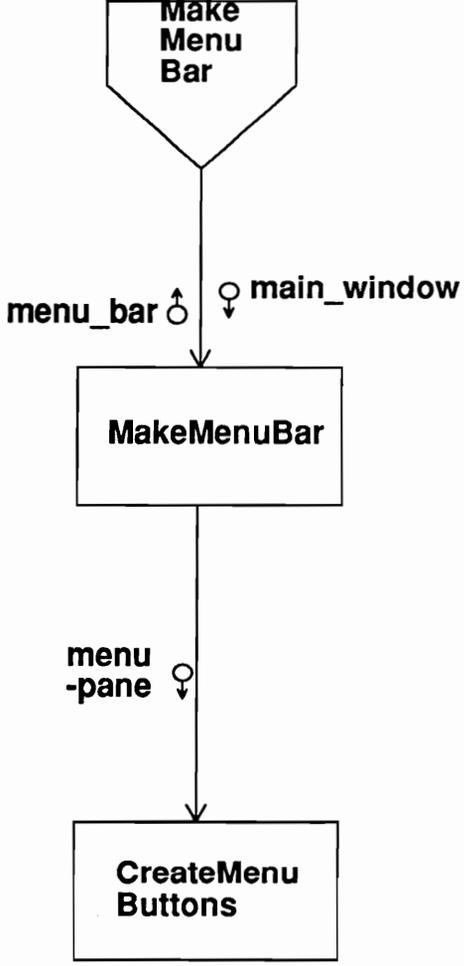


APPENDIX C: GRIM STRUCTURE CHARTS / M-SPECS

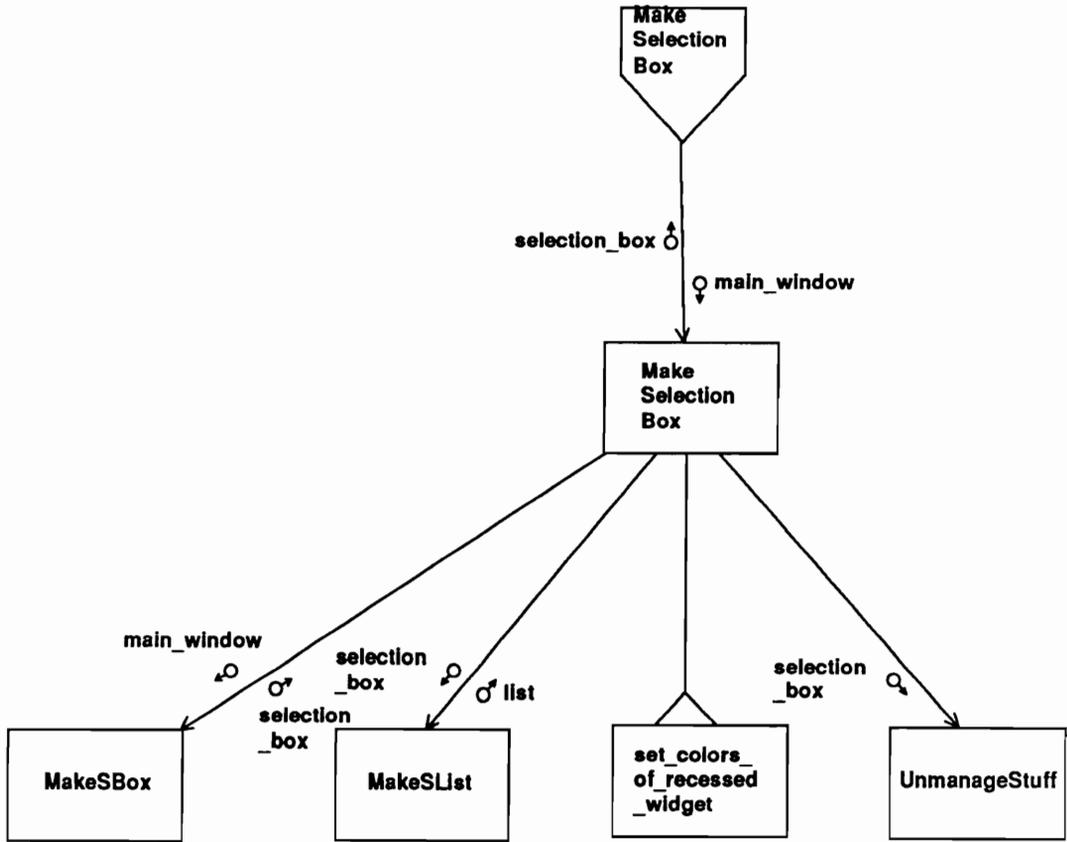
GRIM_Widget;9
GRIM_Widget



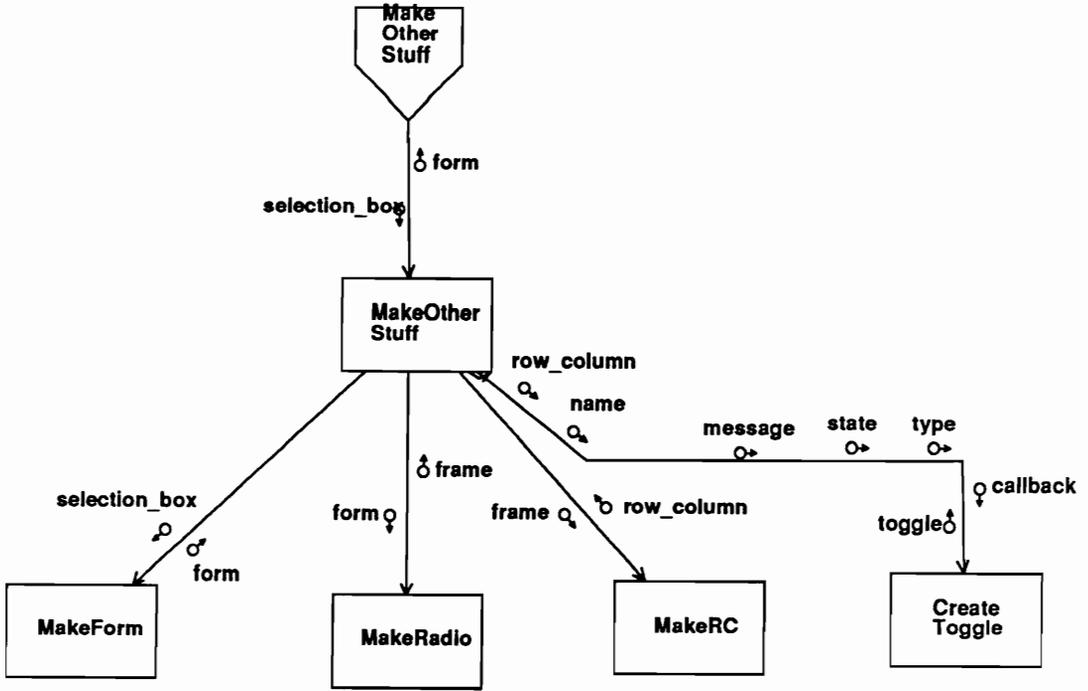
MakeMenuBar;2
No title



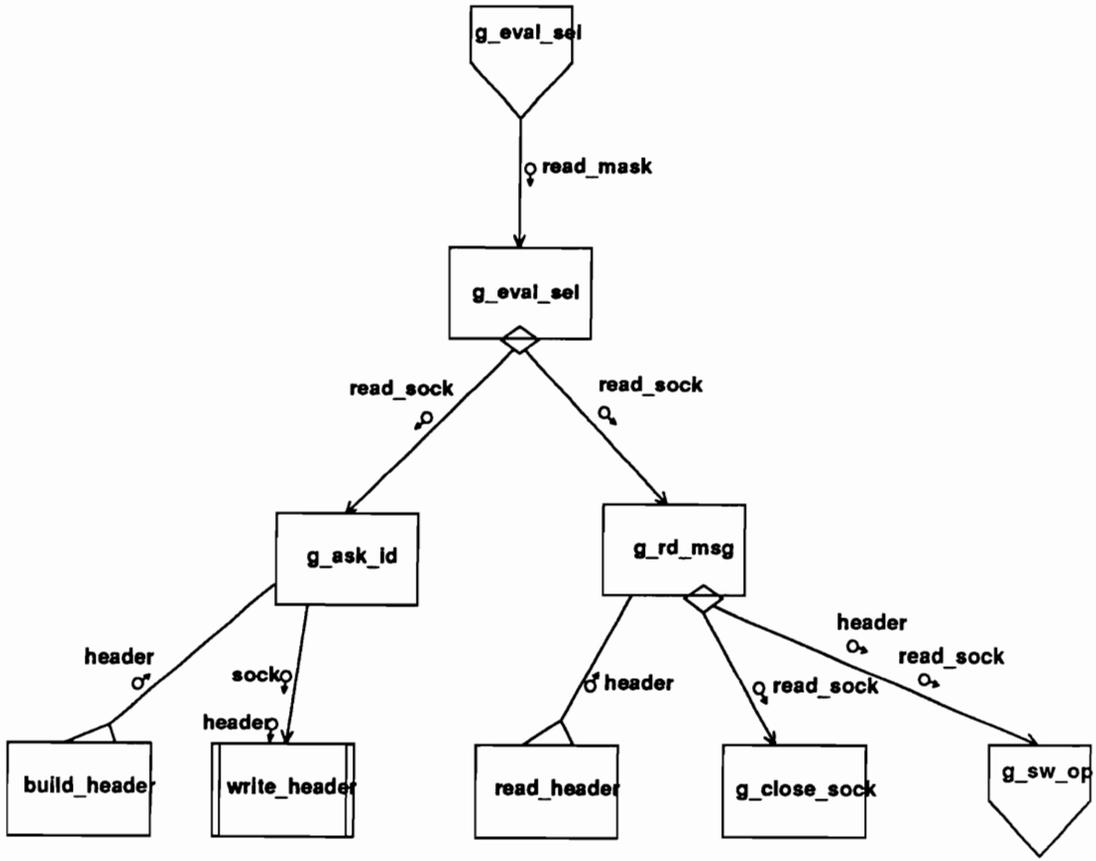
MakeSelectionBox;2
No title



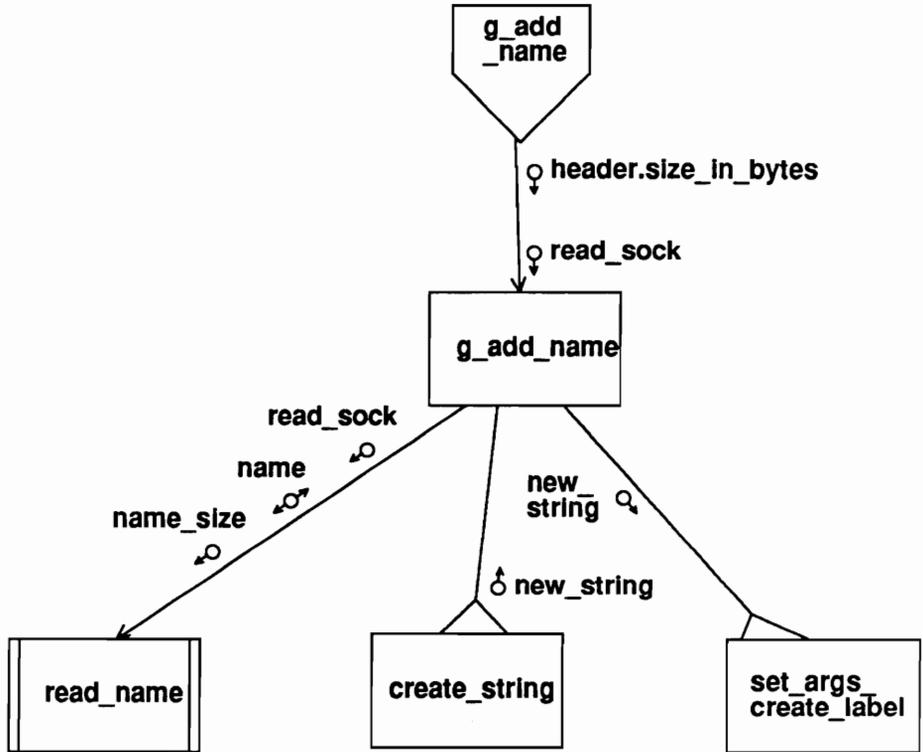
MakeOtherStuff;2
No title



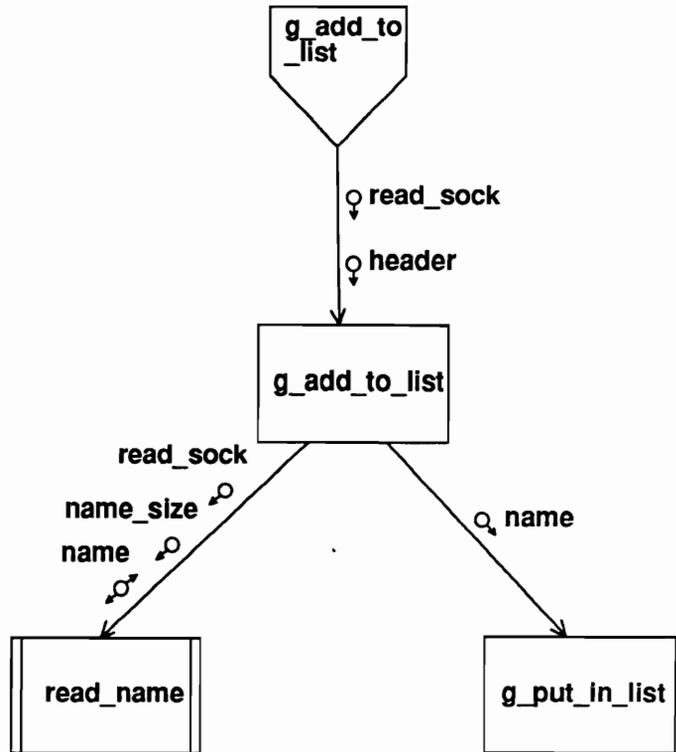
g_eval_sel;3
No title



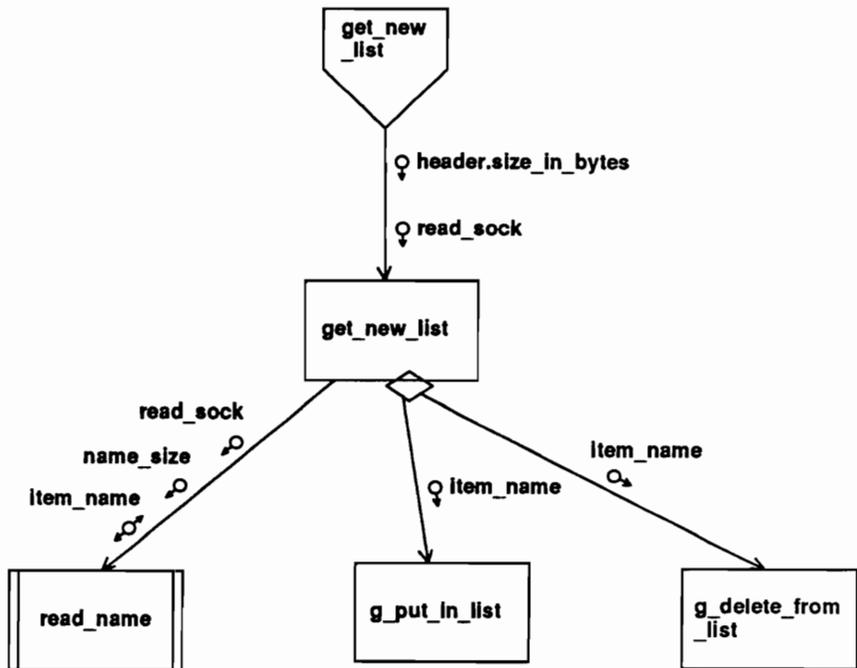
g_add_name;2
No title



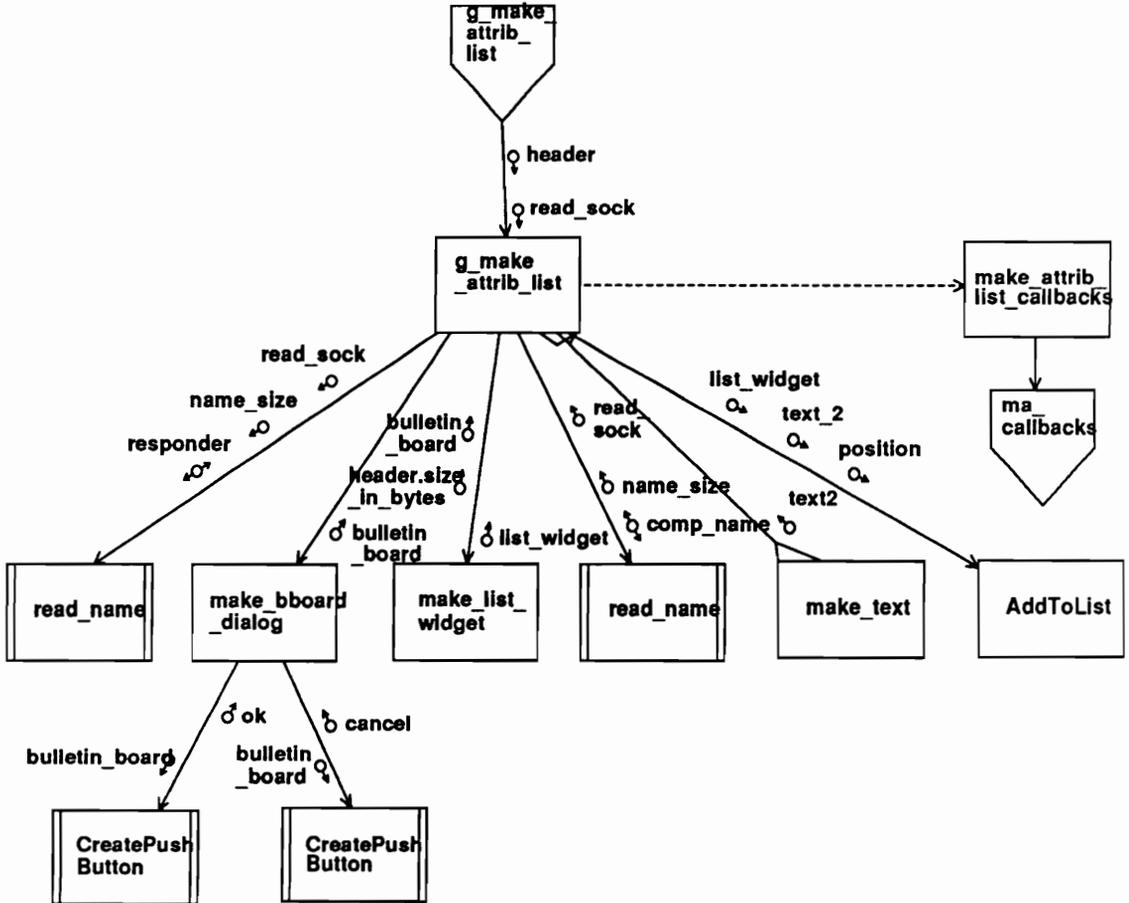
g_add_to_list;3
No title



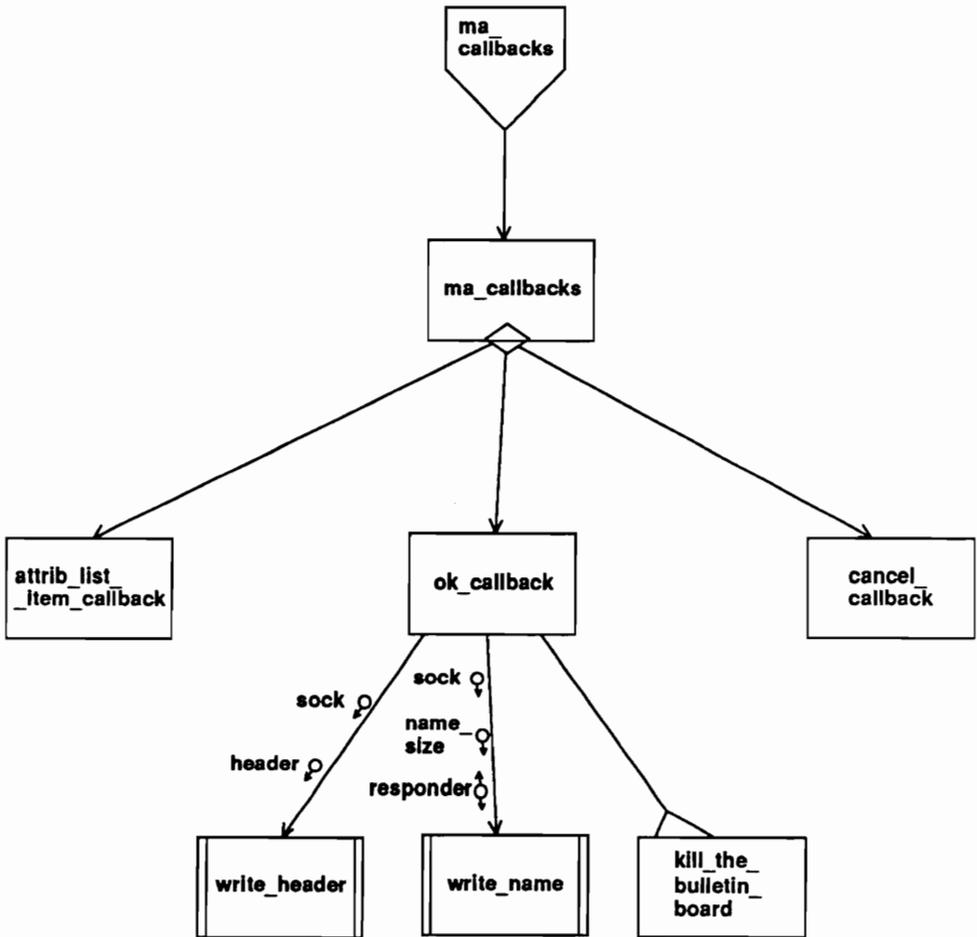
g_get_new_list;2
No title



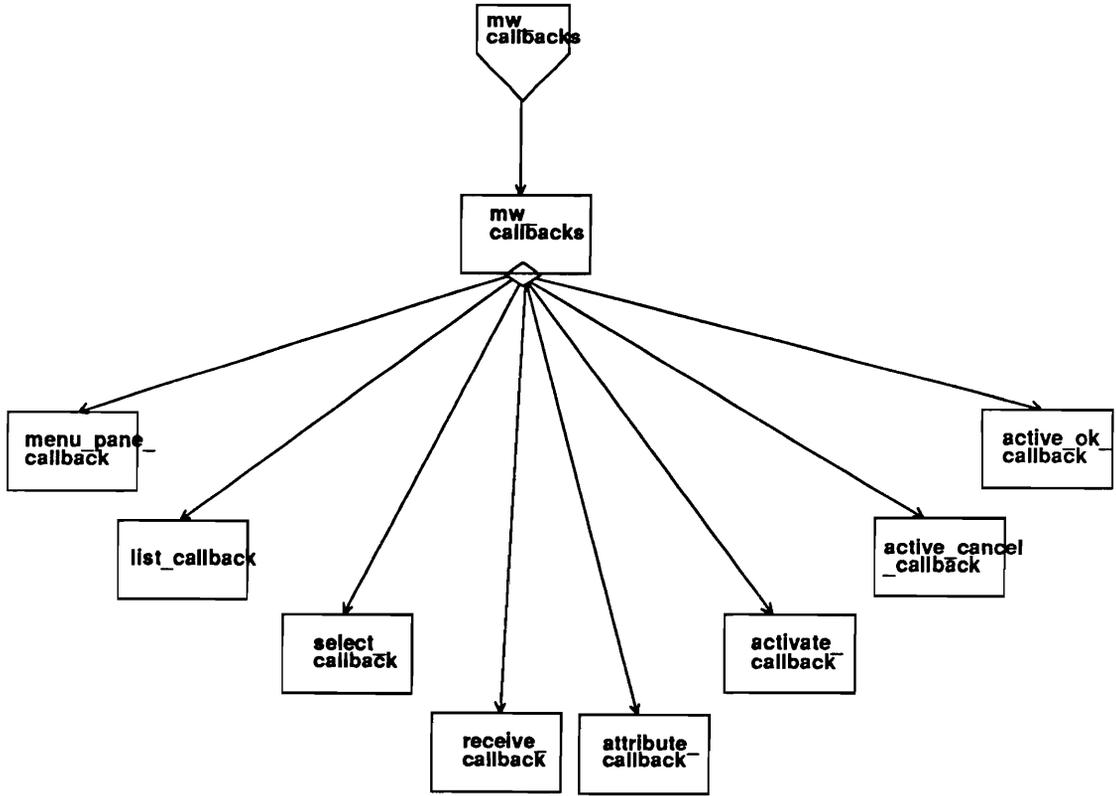
g_make_attr_list;3
No title



ma_callbacks;2
No title



mw_callbacks;1
No title



NAME: GRIM;4

TITLE: GRIM main module

PARAMETERS:

sockets : data_out
num_socks : data_out
listnum : data_out
list_item : data_out
my_client : data_out

LOCALS:

Socket * socket used to listen for connections *
grimmy * server internet information *
one
grim_len

BODY:

```
/* =====  
Source Code Filename: GRIM.c  
Special Considerations: NONE  
Purpose:  
This is the main module of the GRIM widget  
interface. Its purpose is to establish a socket  
to listen for a connection request from the owning  
client and to make a widget for display.  
This module is generic EXCEPT for the pathname  
which is defined at the top of the program. This  
pathname identifies a unique UNIX socket which  
must match the pathname set by the owner client.  
Belongs to GRIM  
===== */  
#define _BSD  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <netdb.h>  
#include <sys/ioctl.h>  
#include <errno.h>  
#include <sys/un.h>  
#include <X11/StringDefs.h>  
#include <Xm/Xm.h>  
#include "../mysock2.h"  
#define pathname "../execs/s.grimsock"  
int sockets[2];  
int num_socks;  
int listnum;  
XmString list_item[50];  
/*char client_list[50][50];*/
```

```

char *my_client;
void make_widget();

void main()
{
int Sock;                /* socket on which listening occurs */
struct sockaddr_un grimmy; /* server internet information */
static int one = 1;      /* set as a constant */
int grim_len;
listnum = 0;

/* --- open socket to listen on and use a stream connection --- */
Sock = socket(AF_UNIX, SOCK_STREAM,0);
if (Sock < 0)
{
perror("server:socket");
exit(-3);
}
sockets[0] = Sock;
num_socks = 1;

/* --- clear the server structure --- */
bzero((char *)&grimmy, sizeof(grimmy));
grimmy.sun_family = AF_UNIX;
strcpy(grimmy.sun_path, pathname);
grim_len = strlen(grimmy.sun_path) + sizeof(grimmy.sun_family);

/* --- bind the Sock to the server --- */
if (bind (Sock, (struct sockaddr *)&grimmy, grim_len) < 0)
{
perror("server:bind");
exit(-3);
}

listen (Sock, 5);
make_widget();
unlink(pathname);
} /* --- end main module --- */

```

NAME: g_make_widget;5

TITLE: GRIM g_make widget

PARAMETERS:

receive_info_toggle : data_out * Widget *

activate_toggle : data_out * Widget *

client_attrib_toggle : data_out * Widget *

```

selection_box : data_out * Widget *
main_window : data_out * Widget *
bulletin : data_out * Widget *
row_column : data_out * Widget *
label_widget : data_out * Widget *
event_generator : data_out * int *
active_switch : data_out * int *
o_active_one : data_out * Widget *
recieve_id : data_out * Widget *
frame : data_out * Widget *
list : data_out * Widget *

```

LOCALS:

```

parent * top level widget *
menu_bar * widget *
form * widget *

```

BODY:

```

/* =====
Source Code Filename: g_make_widget.c
Special Considerations: NONE
Purpose:
To create a widget for displaying choice
data to the user. The widget consists of a
main window with a menu bar at the top containing
choices to reset the list and to exit the widget.
Beneath the menu bar is a selection list containing
the applications in the integrated system with
from which the owning client (client who owns the GRIM
widget) can request data.
Included in this m-spec is the code for make_widget and
all modules beneath it with the exception of those
stemming from the work proc declared in the main
portion of make_widget. Those functions will be
listed separately in other m-specs. The callbacks
for the make_widget routines are also included here.
Belongs to GRIM
===== */
#include <stdio.h>
#define FALSE 0
#define TRUE 1
#define size_of_name 50
#include <sys/un.h>
#define pathname "./execs/s.grimsock"
#include <X11/Intrinsic.h>
#include <X11/Shell.h>
#include "./mysock2.h"
#include <Xm/Xm.h>
#include <Xm/CascadeB.h>
#include <Xm/DialogS.h>

```

```

#include <Xm/BulletinB.h>
#include <Xm/Command.h>
#include <Xm/FileSB.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/MainW.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/SelectioB.h>
#include <Xm/ToggleBG.h>
#include <Xm/ToggleB.h>
#include <X11/MwmUtil.h>

```

```

static Widget MakeMainWindow();
static Widget MakeMenuBar();
static void CreateMenuButtons();
static Widget MakeSelectionBox();
static Widget MakeSBox();
static Widget MakeSList();
static Widget MakeOtherStuff();
static Widget MakeDialogBox();
static Widget CreateToggle();
static Widget MakeForm();
static void UnmanageStuff();
static Widget MakeRC();
static Widget MakeRadio();
static XmString Str2XmString();
static Widget CreateScrolledList();
static Widget create_active_dialog();
static Widget MakeActiveDialog();
static Widget MakeLabel();
static void SetLabel();
static void tell_xchg_client();
static void send_attrib_msg();
void g_select_loop();

```

```

extern int listnum;
extern XmString list_item[50];
extern int sockets[2];
extern int num_socks;
extern char *my_client;

```

```

/* _____ GLOBAL DECLARATIONS _____ */
#define MENU_HELP 200
#define MENU_EXIT 201
#define MENU_RESET 202
#define SEND_EVENT 1
#define RECEIVE_EVENT 2
#define ACTIVATE_EVENT 3

```

```

#define ATTRIBUTE_EVENT 4
#define NONE 0

static Widget receive_info_toggle;
static Widget activate_toggle;
static Widget client_attrib_toggle;
Widget selection_box;
static Widget main_window;
static Widget list;
static Widget bulletin;
Widget row_column;
Widget label_widget;
int event_generator;
int active_switch;
Widget o_active_one;
char *receive_id;
static Widget frame;

/*_____*/
/* — callback for the menu bar selection “actions” — */
void menu_pane_callback(w, client_data, call_data)
Widget w;
caddr_t client_data, call_data;
{
Arg args[10];
int n;
XmAnyCallbackStruct *cbstruc = (XmAnyCallbackStruct *) call_data;

switch((int) client_data)
{
case MENU_EXIT:
printf(“EXITING THE SERVER PROGRAM\n”);
unlink(pathname);
exit(0);
case MENU_RESET:
n = 0;
XtSetArg(args[n], XmNset, FALSE); n++;
XtSetArg(args[n], XmNindicatorOn, TRUE); n++;
XtSetValues(receive_info_toggle, args, n);
XtSetValues(activate_toggle, args, n);
XtSetValues(client_attrib_toggle, args, n);
break;
default:
printf(“unexpected tag in menu_pane_callback\n”);
break;
}
}
/*_____*/
/* — create the callback for list action — */
void list_callback( w, client_data, call_data)
Widget w;

```

```

caddr_t client_data;
caddr_t call_data;
{
extern Boolean activate;
int size_of_client = 50;
char *string;
XmListCallbackStruct *list_data = (XmListCallbackStruct *)call_data;
int n;
Arg args[10];

/* — put chosen list item into the client_id — */
XmStringGetLtoR (list_data->item, XmSTRING_DEFAULT_CHARSET, &string);
if (event_generator == RECEIVE_EVENT)
{
receive_id =(char *) malloc( size_of_client);
receive_id = strcpy(receive_id, string);
} else if (event_generator = ATTRIBUTE_EVENT) {
send_attrib_msg(string);
}
event_generator = NONE;
/* — now determine if it is for the sender or the receiver — */
/* — enable the send and receive and activate radio buttons — */
if (*receive_id != NULL)
{
n = 0;
XtSetArg( args[n], XmNset, FALSE);
XtSetValues(receive_info_toggle, args, n);
XtSetValues(activate_toggle, args, n);
XtSetSensitive(receive_info_toggle, TRUE);
XtSetSensitive(activate_toggle, TRUE);
}
return;
}
/*_____*/
/* — add the callback for the work proc — */
Boolean select_callback(client_data)
caddr_t client_data;
{
/* — call the select_loop module when no event in widget queue — */
select_loop();
return(FALSE);
}
/*_____*/
/*_____*/
void receive_callback( w, client_data, toggle_struct)
Widget w;
XmToggleButtonCallbackStruct *toggle_struct;
{
/* — if the new state of the toggle is true, then show available list— */
if( toggle_struct->set != FALSE)
{

```

```

    event_generator = RECEIVE_EVENT;
}
return;
}
/*-----*/
void attribute_callback( w, client_data, toggle_struct)
Widget w;
XmToggleButtonCallbackStruct *toggle_struct;
{
/* — if the new state of the toggle is true, then show available list—*/
if( toggle_struct->set != FALSE)
{
    event_generator = ATTRIBUTE_EVENT;
}
return;
}
/*-----*/
void activate_callback( w, client_data, toggle_struct)
Widget w;
caddr_t client_data;
XmToggleButtonCallbackStruct *toggle_struct;
{
/* — if the new state of the toggle is true, then show available list—*/
if( toggle_struct->set != FALSE)
{
/* — need to create a dialog widget that lets the user accept choice */
o_active_one = create_active_dialog(row_column, receive_id);
}
return;
}
/*=====*/
void active_cancel_callback(w, client_data, call_data)
Widget w;
caddr_t client_data, call_data;
{
/* — do nothing — */
printf("in active_cancel doing absolutely nothing\n");
}
/*=====*/
void active_ok_callback(w, client_data, call_data)
Widget w;
caddr_t client_data, call_data;
{
    active_switch = TRUE;
    tell_xchg_client(receive_id);
}
/*=====*/
void g_make_widget()
{
Widget parent;

```

```

Widget menu_bar;
Widget form;
event_generator = 0;

/* — initialize the top shell — */
parent = XtInitialize("make_widget.c",
                    "X_GRIM",
                    NULL,
                    0,
                    NULL,
                    0);

/* — make the main window for the widget — */
main_window = MakeMainWindow(parent);

/* — make the menu bar in the main window — */
menu_bar = MakeMenuBar (main_window);

/* — make a selection box — */
selection_box = MakeSelectionBox (main_window);

/* — make other things, like buttons, to put in the box — */
form = MakeOtherStuff (selection_box);

/* — set up the main window — */
XmMainWindowSetAreas (main_window, menu_bar, NULL, NULL, NULL selection_box);

/* — set up the work procedure for branching — */
/* — add a work proc to keep the select polling — */
SetWorkProc();
XtRealizeWidget(parent);
XtMainLoop();
return;
}
/*=====*/
static Widget MakeMainWindow(Widget parent)
{
int n;
Arg args[10];
Widget m_window;

n = 0;
XtSetArg (args[n], XmNscrollingPolicy, XmAPPLICATION_DEFINED); n++;
XtSetArg (args[n], XmNwidth, 275); n++;
XtSetArg (args[n], XmNheight, 375); n++;
m_window = XmCreateMainWindow (parent, "main_window", args, n);
XtManageChild (m_window);
return(m_window);
}
/*=====*/
static Widget MakeMenuBar( Widget widget)
{

```

```

Widget menu_bar;
Widget cascade;
Widget menu_pane;
Arg args[10];
int n;

/* — create the menu bar on the main window(widget) — */
n = 0;
menu_bar = XmCreateMenuBar( widget, "menu_bar", args, n);
XtManageChild(menu_bar);

/* — create pulldown menu off of the menu bar — */
n = 0;
menu_pane = XmCreatePullDownMenu(menu_bar, "menu_pane", args, n);
CreateMenuButtons(menu_pane);
n = 0;
XtSetArg (args[n], XmNsubMenuId, menu_pane); n++;
cascade = XmCreateCascadeButton (menu_bar, "Actions", args, n);
XtManageChild(cascade);
return(menu_bar);
}
/*=====*/
void CreateMenuButtons(Widget menu_pane)
{
Widget button;
int n;

n = 0;
button = XmCreatePushButton (menu_pane, "Reset", args, n);
XtAddCallback (button, XmNactivateCallback, menu_pane_callback, MENU_RESET);
XtManageChild (button);
n = 0;
button = XmCreatePushButton (menu_pane, "Exit", args, n);
XtAddCallback (button, XmNactivateCallback, menu_pane_callback, MENU_EXIT);
XtManageChild (button);
return;
}
/*=====*/
static Widget MakeSelectionBox (Widget widget)
{
Widget text;
Arg      args[10];
int      n;
Widget  hsbar, vsbar;
Widget  s_box;
XrmValue pixel_data;

/* — create the selectionne box — */
s_box = MakeSBox(widget);

/* — register callbacks for selection box list — */

```

```

list = MakeSList(s_box);

/* — set the colors of the recessed widgets — */
if (DefaultDepthOfScreen(XDefaultScreenOfDisplay(XtDisplay(widget))) > 1)
{
    text = XmSelectionBoxGetChild (s_box, XmDIALOG_TEXT);
    XtSetArg (args[0], XmNhorizontalScrollBar, &hsbar);
    XtSetArg (args[1], XmNverticalScrollBar, &vsbar);
    XtGetValues (XtParent(list), args, 2);
    _XmSelectColorDefault (s_box, NULL, &pixel_data);
    XtSetArg (args[0], XmNbackground, *((Pixel *) pixel_data.addr));
    XtSetValues (list, args, 1);
    XtSetValues (text, args, 1);
    XtSetValues (hsbar, args, 1);
    XtSetValues (vsbar, args, 1);
}

/* — unmanage children that weren't needed — */
UnmanageStuff(s_box);
XtManageChild(s_box);
return(s_box);
}
/* ===== */
static Widget MakeSBox(Widget widget)
{
    int i, n;
    Arg args[10];
    Widget s_box;
    XmString charset = (XmStringCharSet) XmSTRING_DEFAULT_CHARSET;
    XmString new_string;

    /* — clear out the list item array — */
    listnum = 0;
    for (i = 0; i < 50; i++)
    {
        list_item[i] = XmStringCreateLtoR (NULL, charset);
    }

    /* — set list header text — */
    new_string = XmStringCreateLtoR("Exchange Selections for ", charset);

    /* — create the selection box — */
    n = 0;
    XtSetArg (args[n], XmNshadowThickness, 1); n++;
    XtSetArg (args[n], XmNshadowType, XmSHADOW_OUT); n++;
    XtSetArg (args[n], XmNtextString, list_item[0]); n++;
    XtSetArg (args[n], XmNlistItems, list_item); n++;
    XtSetArg (args[n], XmNlistItemCount, listnum); n++;
    XtSetArg (args[n], XmNlistLabelString, new_string); n++;
    XtSetArg (args[n], XmNselectionLabelString,
        XmStringCreateLtoR("Current Exchange Selection", charset)); n++;
}

```

```

s_box = XmCreateSelectionBox(widget, "selection_box", args, n);
return(s_box);
}
/* ===== */
static Widget MakeSList(Widget selection_box)
{
Widget s_list;

/* — add a list to the field of the selection box — */
s_list = XmSelectionBoxGetChild (selection_box, XmDIALOG_LIST);

/* — add callbacks for the list — */
XtAddCallback (s_list, XmNbrowseSelectionCallback, list_callback, NULL);
XtAddCallback (s_list, XmNdefaultActionCallback, list_callback, NULL);
return(s_list);
}
/* ===== */
static void UnmanageStuff(Widget selection_box)
{
int i;
Widget kid[5];

/* — unmanage children not needed by this selection box — */
i = 0;
kid[i++] = XmSelectionBoxGetChild (selection_box, XmDIALOG_SEPARATOR);
kid[i++] = XmSelectionBoxGetChild (selection_box, XmDIALOG_OK_BUTTON);
kid[i++] = XmSelectionBoxGetChild (selection_box, XmDIALOG_CANCEL_BUTTON);
kid[i++] = XmSelectionBoxGetChild (selection_box, XmDIALOG_APPLY_BUTTON);
kid[i++] = XmSelectionBoxGetChild (selection_box, XmDIALOG_HELP_BUTTON);
XtUnmanageChildren (kid, i);
return;
}
/* ===== */
static Widget MakeOtherStuff (Widget widget)
{
Widget box;
Arg args[10];
int n;
XmString label_string = NULL;

/* — create outer form box — */
box = MakeForm(widget);

/* — create radio box and dialog style toggles */
frame = MakeRadio(box);
row_column = MakeRC(frame);
receive_info_toggle = CreateToggle(row_column,
                                   "receive data",
                                   "CLIENT FROM WHICH TO RECEIVE DATA",
                                   FALSE,
                                   XmN_OF_MANY,

```

```

        receive_callback);

client_attrib_toggle = CreateToggle(row_column,
        "client attribute",
        "CLIENT ATTRIBUTE LISTING",
        FALSE,
        XmN_OF_MANY,
        attribute_callback);

activate_toggle = CreateToggle(row_column,
        "activate",
        "ACTIVATE EXCHANGE",
        FALSE,
        XmN_OF_MANY,
        activate_callback);

XtSetSensitive(activate_toggle, FALSE);
return(box);
}
/
*=====*/
static Widget MakeForm(Widget widget)
{
int n;
Arg args[10];
Widget box_form;

/* — create outer form — */
n = 0;
XtSetArg(args[n], XmNy, 300); n++;
XtSetArg(args[n], XmNx, 0); n++;
XtSetArg(args[n], XmNwidth, 200); n++;
XtSetArg(args[n], XmNheight, 100); n++;
box_form = XmCreateForm (widget, "outer_form", args, n);
XtManageChild(box_form);
return(box_form);
}
/
*=====*/
static Widget MakeRadio(Widget box)
{
int n;
Arg args[10];
Widget radio_frame;

/* — create radio box */
n = 0;
XtSetArg (args[n], XmNshadowType, XmSHADOW_ETCHED_IN); n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM); n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM); n++;

```

```

XtSetArg (args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
XtSetArg (args[n], XmNbottomPosition, 75); n++;
radio_frame = XmCreateFrame (box, "frame", args, n);
XtManageChild (radio_frame);
return(radio_frame);
}
/
*=====*/
static Widget MakeRC(Widget frame)
{
int n;
Arg args[10];
Widget rc;

/* — make row column widget for toggles — */
n = 0;
XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
rc = XmCreateRowColumn (frame, "row_column", args, n);
XtManageChild(rc);
return(rc);
}
/
*=====*/
Widget CreateToggle(parent, name, message, state, type, callback_func)
Widget parent;
char name[];
char message[];
Boolean state;
int type;
void (*callback_func)();
{
Widget toggle_widget;
XmString motif_string;
XmString Str2XmString();
Arg args[10];
int n;

/* — change message into motif string — */
motif_string = Str2XmString(message);
n = 0;
XtSetArg (args[n], XmNlabelString, motif_string); n++;
XtSetArg (args[n], XmNindicatorType, type); n++;

/* — try XmNindicatorOn with True and False — */
XtSetArg(args[n], XmNindicatorOn, TRUE); n++;

/* XtSetArg(args[n], XmNindicatorOn, FALSE); n++;*/
XtSetArg(args[n], XmNset, state); n++;
toggle_widget = XmCreateToggleButton(parent, name, args, n);
XtManageChild(toggle_widget);
}

```

```

/* — add a callback for when the value is changed — */
XtAddCallback(toggle_widget, XmNvalueChangedCallback, callback_func, NULL);
XmStringFree(motif_string);
return(toggle_widget);
}
/* ===== */
/* ===== */
SetWorkProc()
{
Boolean work_state = FALSE;

if(work_state == FALSE)
{
work_state = XtAddWorkProc(select_callback, NULL);
printf("starting select callback \n");
}
return;
}
/* ===== */
XmString Str2XmString (string)
char *string;
{
XmString motif_string;

/* — create motif string — */
motif_string = XmStringCreateLtoR(string,
XmSTRING_DEFAULT_CHARSET);
return(motif_string);
}
/* ===== */
/* ===== */
Widget create_active_dialog(Widget parent, char receiver[])
{
Widget active_widge, dead_widget;
active_widge = MakeActiveDialog(parent, receiver);

/* — get rid of the help button — */
dead_widget = XmMessageBoxGetChild(active_widge, XmDIALOG_HELP_BUTTON);
XtUnmanageChild(dead_widget);

/* — set up callback on cancel button — */
XtAddCallback(active_widge, XmNcancelCallback, active_cancel_callback, NULL);

/* — set up callback on ok button — */
XtAddCallback(active_widge, XmNokCallback, active_ok_callback, NULL);
return(active_widge);
}
/* _____ */
static Widget MakeActiveDialog(Widget parent, char receiver[])
{

```

```

char *new_receive_string;
XmString motif_string, Str2XmString();
int n;
Arg args[10];
Widget a_widge;

/* — concatenate the sender and receiver with messages — */
new_receive_string = strcat("client from whom data received is ", receiver);
printf("c_a_d: strcat new_receive_string %s\n", new_receive_string);
motif_string = Str2XmString(new_receive_string);

/* — create widget — */
n = 0;
XtSetArg(args[n], XmNmessageString, motif_string); n++;
a_widge = XmCreateInformationDialog(parent,
                                   "active_widget",
                                   args,
                                   n);

XtManageChild(a_widge);

/* — reset string message — */
new_receive_string = strcpy(new_receive_string, "client from whom data is requested ");
free(new_receive_string);
XmStringFree(motif_string);
return(a_widge);
}
/*_____*/
void tell_xchg_client(char *name)
{
HEADER header;

/* — send a message to the client to tell server which exchg — */
header.size_in_bytes = size_of_name;
header.maj_opcode = 1;
header.min_opcode = 1;

/* — send it — */
if (write(sockets[1], &header, sizeof(HEADER)) < 0)
{
perror("tell_xchg_client: write header");
exit(1);
}
printf("tell_client: the name requested is %s\n", name);
if( write(sockets[1], name, size_of_name) < 0)
{
perror("tell_xchg_client: write name");
exit(1);
}
return;
} /* — end tell_xchg_client — */
/*_____*/

```

```

void send_attrib_msg(char *name)
{
HEADER header;

header.size_in_bytes = 0;
header.maj_opcode = 1;
header.min_opcode = 3;
printf("maek_wiget: sending message to client for attributes \n");

/* — send it — */
if (write(sockets[1], &header, sizeof(HEADER)) < 0)
{
perror("sam: write header");
exit(1);
}
printf("sam: the name requested is %s\n", name);
if( write(sockets[1], name, size_of_name) < 0)
{
perror("sam: write name");
exit(1);
}
return;
} /* — end send_attrib_msg.c — */

```

NAME: g_select_loop;5

TITLE: GRIM g_select_loop

PARAMETERS:

LOCALS:

BODY:

/* ===== */

Source Code Filename: g_select_loop.c

Special Considerations: NONE

Purpose:

To check whether or not a signal has been detected
on a socket.

Belongs to GRIM

===== */

/

*===== */

/*

g_select_loop.c

Function: sets the select mode on for the server to screen incoming connections

Variables: sock - socket info

Coded by: Michele Grieshaber

Date : 08/28/91

```
*/
/
*=====*/
#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include "./mysock2.h"

/* — supporting routines — */
void g_eval_sel();
/* — end supporting routines — */

extern int sockets[2];
extern int num_socks;

void g_select_loop()
{
fd_set read_mask;           /* mask which filters sockets for reading */
struct timeval to;         /* time structure for select timeout */
int i, rc;                 /* rc is the return code variable */
int lsock;                 /* product of a socket sort — largest sock*/
extern int errno;         /* error number for debug purposes */

/* — clear the read mask for the select — */
FD_ZERO(&read_mask);

/* — compare the mask against all available sockets — */
/* — also keep track of the largest socket value for later use — */
/* — but to do that, set sock initially to zero — */
lsock = 0;
for (i = 0; i < num_socks; i++)
{
    FD_SET(sockets[i], &read_mask);
    if(sockets[i] > lsock)
    {
        lsock = sockets[i]; /* sorting lsock for nfds arg */
    }
}
}/* — end for nsocks — */

/* — set the timeout values for the select — */
bzero((char *)&to, sizeof(to));
to.tv_sec = 0;
```

```

/* — hang out in the select — */
rc = select(lsock+1, &read_mask, (fd_set *)0, (fd_set *)0, &to);
if(rc < 0)
{
    perror("select");
    exit(1);
} else if (rc > 0) {
    /* — evaluate the response to select if any — */
    g_eval_sel(read_mask);
} /* — end if rc — */

return;
} /* — end g_select_loop.c — */

```

NAME: g_eval_sel;6

TITLE: GRIM g_eval_sel

PARAMETERS:

read_mask : data+control_in

LOCALS:

BODY:

/* =====

Source Code Filename: g_eval_sel.c

Special Considerations: NONE

Purpose:

To evaluate the signal which occurred on a socket.

Belongs to GRIM

===== */

/

=====/

/*

g_eval_sel.c

Function: evaluates the value of the read mask returned from the
select call in set_sel.

If the signal comes in on the listening socket, the
client is requesting to be accepted for connection by
the GRIM server.

If the signal comes on a socket that has already been
established (accepted), the header is read by cl_rdmsg
and appropriate action is taken.

Variables: sockets - array containing socket info
read_mask - indicates which sockets have info on them

Coded by: Michele Grieshaber

Date : 06/10/91

*/

```

/
*=====*/
#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/file.h>
#include <signal.h>
#include <sys/select.h>
#include <errno.h>
#include "../mysock2.h"

/* — supporting routines — */
void g_rd_msg();
void g_ask_id();
/* — end supporting routines — */

extern int sockets[2];
extern int num_socks;

g_eval_sel(fd_set read_mask)
{
int new_sock;          /* new socket accepted by the server */
int i;                /* just your ordinary everyday integer */
struct sockaddr_in sin; /* structure containing client ip stuff */
int length = sizeof(sin); /* length of above structure */

/* — check to see if the read_mask matches any of the available sockets —*/
if(FD_ISSET(sockets[0], &read_mask))
{
/* — accept the new connection — */
if((sockets[num_socks] = accept(sockets[0],&sin,&length)) < 0)
{
perror("Server:accept");
exit(-3);
}
/* — send a message to the newly connected client to get his name */
g_ask_id(sockets[num_socks]);
num_socks += 1;
} else {
/* — check the other connected sockets one at a time for info — */
for(i = 1; i < num_socks; i++)
{
if (FD_ISSET(sockets[i], &read_mask))
{

```

```

    /* — read message on socket — */
    g_rd_msg(sockets[i]);
    }/* — end if — */
}/* — end for — */
}/* — end if — */

return;
} /* — end g_eval_sel.c — */

```

NAME: g_ask_id;5

TITLE: GRIM g_ask_id

PARAMETERS:

LOCALS:

BODY:

```

/* =====
Source Code Filename: g_ask_id.c
Special Considerations: NONE
Purpose:
To inquire the name of the client application
who owns the GRIM widget
Belongs to GRIM
===== */

```

```

#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include “./mysock2.h”

```

```
void write_header ();
```

```
void g_ask_id(int sock)
{
HEADER header;
```

```

/* — send msg to client asking for an identifying string — */
header.size_in_bytes = 0;
header.maj_opcode = 0;
header.min_opcode = 0;
write_header(sock, header);

```

```
return;
} /* — end g_ask_id — */
```

NAME: g_rd_msg;5

TITLE: GRIM g_rd_msg

PARAMETERS:

LOCALS:

BODY:

```
/* =====
```

Source Code Filename: g_rd_msg.c

Special Considerations: NONE

Purpose:

To read the header off of the socket to send
to be evaluated.

Belongs to GRIM

```
===== */
```

```
/
```

```
*=====*/
```

```
/*
```

g_rd_msg.c

Function: reads the header from the information coming in on a socket.

Header info then sent to a routine which does a switch
on the major and minor opcodes contained in the header.

Variables: read_sock - socket on which info is waiting
sock_struc - structure containing socket info

Coded by: Michele Grieshaber

Date : 06/10/91

changed for the grim interface by mmg on 8/27/91

```
*/
```

```
/
```

```
*=====*/
```

```
#define _BSD
```

```
#define TRUE 1
```

```
#define FALSE 0
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <sys/socketvar.h>
```

```
#include <sys/uio.h>
```

```
#include <errno.h>
```

```
#include “./mysock2.h”
```

```
/* — supporting routines — */
```

```
void g_sw_op();
```

```

void g_close_sock();
/* — end supporting routines — */

g_rd_msg(int read_sock)
{
int nval;           /* return code from read */
HEADER header;    /* header read from the socket. Contains info */
                  /* such as size of info on socket, major opcode */
                  /* and minor opcode _____ */

/* — read the header from the information sitting on the socket — */
nval = read(read_sock, &header, sizeof(HEADER));
if(nval == -1)
{
    perror("rd_msg: read");
    exit(1);
} else if(nval == 0) {
    /* — go to routine to close connection and take socket out of list - */
    g_close_sock(read_sock);
} else {
    /* — send to sw-op to determine action associated with opcode — */
    g_sw_op(header, read_sock);
} /* — end if — */

return;
} /* — end g_rd_msg.c — */

```

NAME: g_close_sock;6

TITLE: GRIM g_close_sock

PARAMETERS:

LOCALS:

BODY:

/* ===== */

Source Code Filename: g_close_sock.c

Special Considerations: NONE

Purpose:

To close a socket which is no longer active

Belongs to GRIM

===== */

/* ===== */

/* g_close_sock.c

Function: deletes a socket from the socket list when a client is closed.

Arguments: int dead_sock — socket that has been closed

SOCK_INFO *sock_struct — structure containing number of

```

sockets and the socket list
Coded by: Michele Grieshaber
Date : 06/05/91
changed for grim interface by mmg on 8/27/91
*/
/*=====*/
#include <stdio.h>
#include <sys/un.h>
#include "../mysock2.h"
#define pathname "./execs/s.grimsock"

extern int sockets[2];
extern int num_socks;

g_close_sock(int dead_sock)
{
int i,j;
static int size_of_name = 50;

/* — loop thru socket list to find entry which matches dead socket — */
for (i = 0 ; i < num_socks; i++)
{
if(sockets[i] == dead_sock)
{
/* — if the socket is dead, close the widget — */
unlink(pathname); /* gets rid of socket file used for communication */
exit(0);
} /* — end if dead_sock — */
} /* — end for i — */

return;
} /* — end g_close_sock.c — */

*****
NAME: g_sw_op;5

TITLE: GRIM g_sw_op

PARAMETERS:
LOCALS:
BODY:
/*=====
Source Code Filename: g_sw_op.c
Special Considerations: NONE
Purpose:
To determine the module which will handle
the message which has just come in on the socket.
Belongs to GRIM

```

```

===== */
/
*===== */
/*
g_sw_op.c
Function: based on the major and minor opcodes contained in the
          header structure passed in from the rd_msg routine,
          this routine (using switch statements) will determine
          the appropriate action to take

Variables:  header   - contains size and maj and minor opcodes
            sock_struc - structure containing socket info
            read_sock - socket on which information resides

Coded by: Michele Grieshaber
Date : 06/10/91
changed for grim interface by mmg on 8/27/91
*/
/
*===== */

#define _BSD
#define TRUE 1
#define FALSE 0
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"
#define ADD 1
#define DELETE 0

/* — supporting routines — */
void g_get_new_list();
void g_add_name();
void g_add_to_list();
void g_make_attrib_list();
/* — end supporting routines — */

g_sw_op(HEADER header, int read_sock)
{
int i, j;          /* an i for an i */

/* — begin major opcode switch — */
switch(header.maj_opcode)
{
case 0:
/* — begin minor opcode switch for major case 0 — */
switch(header.min_opcode)
{
case 0:

```

```

        g_add_name(read_sock, header.size_in_bytes);
    break;

    case 1:
    break;

    case 2:
        g_add_to_list(read_sock, header);
    break;

    case 3:
    break;

    default:
        printf("g_sw_op: not a valid minor opcode\n");
    break;
} /* — end switch(min_opcode) — */
/* — end minor opcode switch for major case 0 ————— */
break;

case 1:
break;

case 2:
    switch(header.min_opcode)
    {
        case 0:
            /* — get updated list info which originated from server — */
            /* — note that header.size_in_bytes is actual an
                action code for the widget to add or delete list — */
            g_get_new_list(read_sock, header.size_in_bytes);
        break;

        case 3:
            /* — get and display attribute list — */
            g_make_attr_list(read_sock, header);
        break;

        default:
            printf("not a valid minor opcode for major opcode = 2\n");
        break;
    } /* — end switch minor opcode for major opcode = 2 — */
    break;

    default:
        printf("g_sw_op: not a valid major opcode\n");
    break;
} /* — end switch(maj_opcode) — */
/* — end major opcode switch ————— */
return;
} /* — end g_sw_op.c — */

```

NAME: g_add_name;7

TITLE: GRIM g_add_name

PARAMETERS:

LOCALS:

BODY:

```
/* =====  
Source Code Filename: g_add_name.c  
Special Considerations: NONE  
Purpose:  
To receive the owning client applicationqs  
name and to place that in the widget for identification  
purposes.  
Belongs to GRIM  
===== */
```

```
#include <stdio.h>  
#include <string.h>  
#include <Xm/Xm.h>  
#include <X11/Intrinsic.h>  
#include <X11/StringDefs.h>  
#include <Xm/SelectioB.h>  
  
extern char *my_client;  
extern Widget selection_box;  
char *read_name();  
  
void g_add_name(int read_sock, int name_size)  
{  
    XmString client_name, new_string;  
    XmString charset = (XmStringCharSet) XmSTRING_DEFAULT_CHARSET;  
    int n;  
    Arg args[10];  
    char *name;  
  
    if(name_size > 50)  
    {  
        return;  
    }  
  
    /* read name to add to list off of socket */  
    name = read_name(read_sock, name, name_size);  
    my_client = (char *)malloc(name_size);  
    my_client = strcpy(my_client, name);  
  
    /* create a Motif string out of the name — */  
    client_name = XmStringCreateLtoR (my_client, charset);  
    new_string = XmStringCreateLtoR("Exchange Selections for ", charset);  
    new_string = XmStringConcat(new_string, client_name);
```

```

/* — set the arguments for the selection box to include new name — */
n = 0;
XtSetArg (args[n], XmNlistLabelString, new_string); n++;
XtSetValues (selection_box, args, n);
free(name);
XmStringFree(client_name);
XmStringFree(new_string);

return;
} /* — end g_add_name — */

```

NAME: g_add_to_list;7

TITLE: GRIM g_add_to_list

PARAMETERS:

LOCALS:

BODY:

```

/* =====
Source Code Filename: g_add_to_list.c
Special Considerations: NONE
Purpose:
To receive a list of clients to put in the
exchange selections list of the widget
Belongs to GRIM
===== */

```

```

#define _BSD
#define size_of_name 50
#define ADD 1
#define DELETE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/un.h>
#include “./mysock2.h”

```

```

/* — declaration of external functions — */

```

```

void g_put_in_list();
char *read_name();

```

```

void g_add_to_list(int read_sock, HEADER header)
{
int i;

```

```

char *name;

/* — get the incoming info from client on xchg list — */
for ( i = 0; i < header.size_in_bytes; i++)
{
    name = read_name(read_sock, name, size_of_name);
    if (header.size_in_bytes == ADD)
    {
        g_put_in_list(name);
    }
} /* — end for header.size_in_bytes (number in list) — */

free(name);
return;
} /* — end g_add_to_list.c — */

```

NAME: g_get_new_list;6

TITLE: GRIM g_get_new_list

PARAMETERS:

LOCALS:

BODY:

```

/* =====
Source Code Filename: g_get_new_list.c
Special Considerations: NONE
Purpose:
To add or delete a client name from the
exchange selections list.
Belongs to GRIM
===== */

```

```

#define _BSD
#define size_of_name 50
#define ADD 1
#define DELETE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/un.h>
#include “./mysock2.h”
/* — declaration of external functions — */
void g_put_in_list();

```

```

void g_delete_from_list();
char * read_name();
/* ----- end declaration of e.funcs ----- */

void g_get_new_list (int sock, int action)
{
int i;
char *new_item;

/* -- read in the list items and put them in a linked list -- */
new_item = read_name(sock, new_item, size_of_name);

/* -- determine if action is add or delete -- */
if (action == ADD)
{
g_put_in_list(new_item);
} else {
g_delete_from_list (new_item);
}

return;
} /* -- end g_get_new_list.c -- */

```

NAME: g_put_in_list;5

TITLE: GRIM g_put_in_list

PARAMETERS:

name : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: g_put_in_list.c

Special Considerations: NONE

Purpose:

Places a name in the selection list of
the GRIM widget

Belongs to GRIM

===== */

#include <stdio.h>

#include "../mysock2.h"

#include <Xm/Xm.h>

#include <X11/Intrinsic.h>

#include <Xm/SelectioB.h>

extern Widget selection_box;

extern int listnum;

```

extern XmString list_item[50];

void g_put_in_list(char *new_item)
{
int n;
XmString charset = (XmStringCharSet) XmSTRING_DEFAULT_CHARSET;
Arg args[10];

/* — go thru linked list and put items in the widget — */
/* — add items to list for each client that is attached — */
list_item[listnum] = XmStringCreateLtoR(new_item, charset);
listnum += 1;
n = 0;
XtSetArg (args[n], XmNtextString, list_item[0]); n++;
XtSetArg (args[n], XmNlistItems, list_item); n++;
XtSetArg (args[n], XmNlistItemCount, listnum); n++;
XtSetValues (selection_box, args, n);

return;
} /*— end g_put_in_list.c — */

```

NAME: g_delete_from_list;5

TITLE: GRIM delete_from_list

PARAMETERS:

item_name : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: g_delete_from_list

Special Considerations: NONE

Purpose:

To delete the name of a client who has disconnected from the integrated system. This client name is in the selection list used to request data exchanges.

Belongs to GRIM

===== */

```

#define size_of_name 50
#include <stdio.h>
#include <string.h>
#include “./mysock2.h”
#include <Xm/Xm.h>
#include <X11/Intrinsic.h>
#include <Xm/SelectioB.h>

```

```

extern Widget selection_box;
extern int listnum;
extern XmString list_item[50];

void g_delete_from_list (char *new_item)
{
int i, j, n;
XmString charset = (XmStringCharSet) XmSTRING_DEFAULT_CHARSET;
Arg args[10];
XmString item;

/* — change the new_item to an XmString, then compare — */
item = XmStringCreateLtoR(new_item, charset);

/* — cycle thru the list and determine which one matches — */
for (i = 0; i <= listnum; i++)
{
if(XmStringCompare (list_item[i], item) == 0)
{
/* — delete that item from the list — */
for (j = i; j <= listnum; j++)
{
list_item[j] = XmStringCopy(list_item[j+1]);
}
listnum -= 1;
}
}

n = 0;
XtSetArg (args[n], XmNtextString, list_item[0]); n++;
XtSetArg (args[n], XmNlistItems, list_item); n++;
XtSetArg (args[n], XmNlistItemCount, listnum); n++;
XtSetValues (selection_box, args, n);

return;
} /*— end g_delete_from_list.c — */

```

NAME: g_make_attrib_list;5

TITLE: GRIM g_make_attrib_list

PARAMETERS:

LOCALS:

BODY:

```
/* =====  
Source Code Filename: g_make_attrib_list.c  
Special Considerations: NONE  
Purpose:  
To create a list of attributes based on  
items sent to the widget from another application  
in the integrated system.  
Belongs to GRIM  
===== */  
#define _BSD  
#define size_of_name 50  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socketvar.h>  
#include <sys/socket.h>  
#include <sys/uio.h>  
#include <netinet/in.h>  
#include <netdb.h>  
#include <sys/un.h>  
#include "../mysock2.h"  
#include <Xm/List.h>  
#include <Xm/BulletinB.h>  
#include <Xm/PushB.h>  
#include <X11/StringDefs.h>  
#include <X11/Intrinsic.h>  
#include <Xm/Xm.h>  
#include <X11/Shell.h>  
#include <X11/MwmUtil.h>  
  
/* — function declarations — */  
char *read_name();  
void write_header();  
static Widget make_bboard_dialog();  
static Widget make_list_widget();  
static XmString Str2XmString();  
static Widget CreatePushButton();  
extern Widget row_column;  
Widget bulletin_board; /* — bulletin board widget for attribs — */  
int comp_num = 0; /* — component number of item chosen — */  
extern int sockets[2]; /* — contains listening and client sock — */  
extern int num_socks; /* — number of socks in sockets array — */  
char *responder; /* — name of client to whom CAL belongs — */  
/* ===== */
```

```

void attrib_list_item_callback(w, client_data, call_data)
Widget w;
caddr_t client_data;
caddr_t call_data;
{
char *string;
XmListCallbackStruct *list_data = (XmListCallbackStruct *)call_data;
int n;
Arg args[10];

/* — put chosen list item into the client_id — */
XmStringGetLtoR (list_data->item, XmSTRING_DEFAULT_CHARSET, &string);

/* — need to extract the component number from the string — */
sscanf(string, "%d", &comp_num);
return;
}
/* ===== */
/* ok_callback:
this callback for the ok button will send the list choice to
the client which then transfers it to the server and so on. */
/* ===== */
void ok_callback(w, client_data, call_data)
Widget w;
caddr_t client_data;
caddr_t call_data;
{
HEADER header; /* — header for protocol msg to client — */

header.size_in_bytes = 0;
header.maj_opcode = 1;
header.min_opcode = 4;
write_header(sockets[1], header);
/* — write the source name to the client — */
/* — source name is the name of the client to whom the client attribute
list belongs _____ */
write_name(sockets[1], responder, size_of_name);
printf("ok_callback: the source name is %s\n", responder);

/* — send comp num to client for transmission to server etc — */
if (write(sockets[1], &comp_num, sizeof(int)) < 0)
{
perror("ok_callback: writing comp num");
exit(1);
}

/* — close the widget — */
XtUnmanageChild(bulletin_board);
free(responder);
return;
}

```

```

/* ===== */
/* cancel_callback:
this callback for the cancel button will destroy the client attribute
list without transmitting any information to the client */
/* ===== */
void cancel_callback(w, client_data, call_data)
Widget w;
caddr_t client_data;
caddr_t call_data;
{
/* — kill the client attribute widget — */
XtUnmanageChild(bulletin_board);
return;
}
/* ===== */
/* ===== */
void g_make_attr_list(int read_sock, HEADER header)
{
int i;                /* — just an integer — */
char *comp_name;     /* — name to go in attribute list — */
int size_of_compname = 21; /* — size of name to go in attrib list — */
int comp_num;        /* — component number — */
Widget list_widget; /* — list widget identifier — */
XmString motif_string, Str2XmString; /* — strings for motif — */
char text[4];        /* — text buffer to put comp_num in — */
char blank[] = “ ”; /* — need i say more? — */
char *text2;         /* — text string for composite list item */

/* — read the source name of the attribute list — */
responder = read_name(read_sock, responder, size_of_name);

/* — create a bulletin board dialog widget — */
bulletin_board = make_bboard_dialog();

/* — create a list widget to go in bulletin board — */
list_widget = make_list_widget(bulletin_board, header.size_in_bytes);

/* — read the list off of the socket and put it in a file selec widget */
for (i = 0; i < header.size_in_bytes; i++)
{
/* — read component number and component name — */
if(read(read_sock, &comp_num, sizeof(int)) < 0)
{
perror(“make_att_list: reading comp_num”);
exit(1);
}
comp_name = read_name(read_sock, comp_name, size_of_compname);

/* — clear out text — */
bzero((char *)text, sizeof(text));
}
}

```

```

/* — write comp number into text buffer — */
sprintf(text, "%d", comp_num);

/* — put text string into text2 string — */
text2 = (char *)malloc(size_of_compname+4+1);
text2 = strcat(text2, text);
/* — add a blank after the comp num in the text string — */
text2 = strcat(text2, blank);
/* — add the name of the component to the string — */
text2 = strcat(text2, comp_name);
/* — add the string as an entry in the list — */
AddToList(list_widget, text2, i+1);

/* — clear and free — */
bzero((char *)text2, size_of_compname+4+1);
free(text2);
free(comp_name);
}
XtManageChild(bulletin_board);
return;
} /* — end make_attrib_list.c — */
/*=====*/
Widget make_bboard_dialog()
{
Widget bulletin;
Widget ok, cancel;
Arg args[10];
int n;
XmString motif_string, Str2XmString();
char text[] = "client attribute list";
motif_string = Str2XmString(text);

/* — create a bulletin board dialog, but donqt manage it. A callback
will manage it later —————*/
n = 0;
XtSetArg(args[n], XmNautoUnmanage, False); n++;
XtSetArg(args[n], XmNnoResize, False); n++;
XtSetArg(args[n], XmNdialogTitle, motif_string); n++;
bulletin = XmCreateBulletinBoardDialog(row_column,
                                     "bulletin",
                                     args,
                                     n);

/* — create push buttons in dialog, the ok is default — */
n = 0;
XtSetArg(args[n], XmNshowAsDefault, 1); n++;
ok = CreatePushButton (bulletin, "OK", args, n, ok_callback);
n = 0;
XtSetArg(args[n], XmNdefaultButton, ok); n++;
XtSetValues(bulletin, args, n);
n = 0;

```

```

XtSetArg(args[n], XmNx, 50); n++;
cancel = CreatePushButton(bulletin, "Cancel", args, n, cancel_callback);
XmStringFree(motif_string);
return(bulletin);
} /* — make_bboard_dialog.c — */
/* ===== */
Widget make_list_widget(Widget parent, int list_size)
{
Widget list;
Arg args[10];
int n;

/* — set up empty list — */
n = 0;
XtSetArg(args[n], XmNitemCount, 0); n++;
XtSetArg(args[n], XmNselectionPolicy, XmSINGLE_SELECT); n++;
XtSetArg(args[n], XmNvisibleItemCount, list_size); n++;
XtSetArg(args[n], XmNy, 50); n++;
list = XmCreateScrolledList(parent,
                             "list",
                             args,
                             n);

XtManageChild(list);
XtAddCallback(list,
              XmNsingleSelectionCallback,
              attrib_list_item_callback,
              NULL);

return(list);
} /* — make_list_widget.c — */
/* ===== */
AddToList(Widget widget, char string[], int position)
{
XmString motif_string, Str2XmString();
motif_string = Str2XmString(string);

XmListAddItemUnselected (widget,
                          motif_string,
                          position);

XmStringFree(motif_string);
} /* — AddToList — */
/* ===== */
XmString Str2XmString (string)
char *string;
{
XmString motif_string;

motif_string = XmStringCreateLtoR(string,
                                  XmSTRING_DEFAULT_CHARSET);

return(motif_string);
}

```

```

/*=====*/
Widget CreatePushButton(parent, name, args, n, callback_func)
Widget parent;
char name[];
Arg *args;
int n;
void (*callback_func)();
{
Widget push_widget;

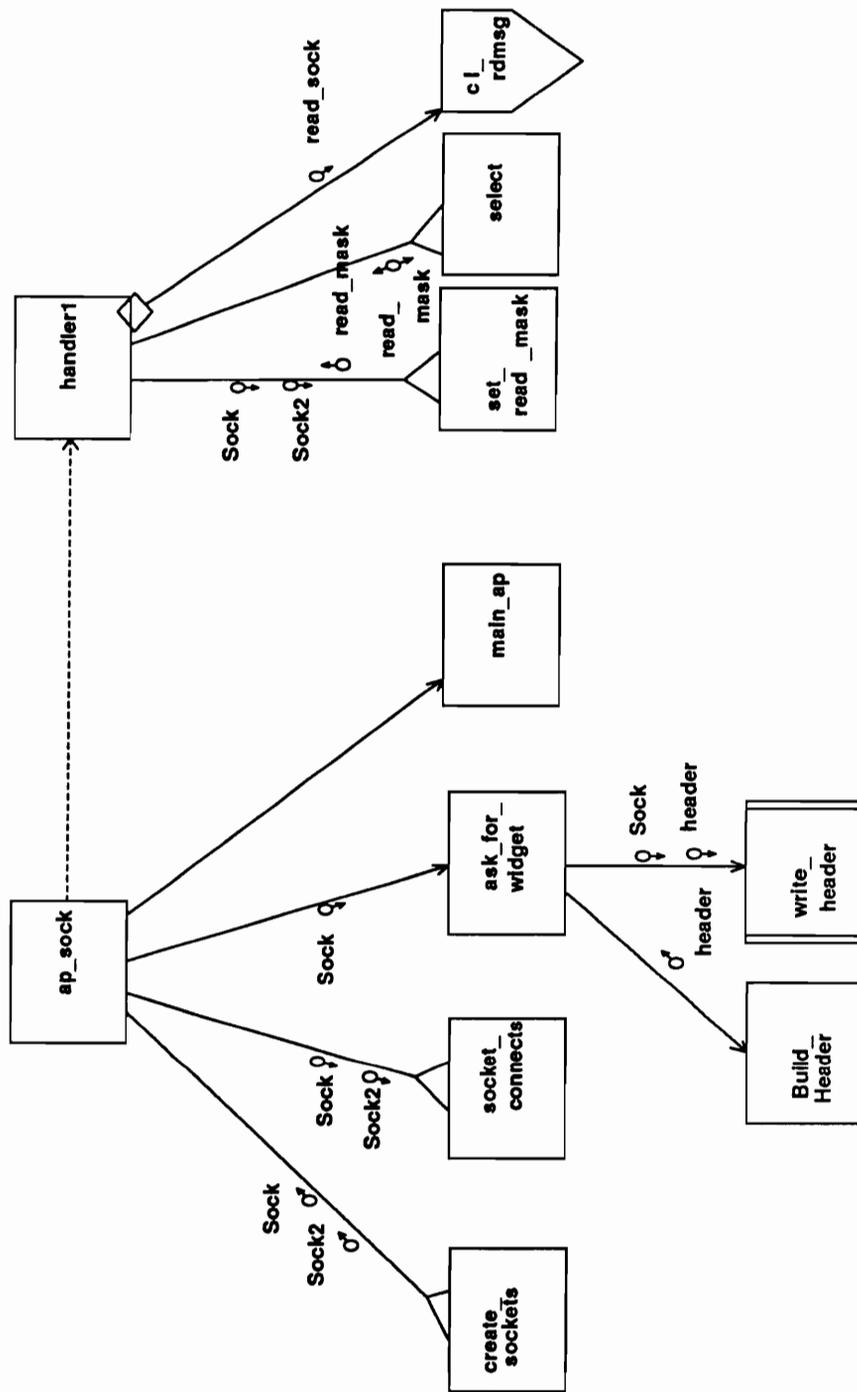
push_widget = XtCreateManagedWidget(name,
                                     xmPushButtonWidgetClass,
                                     parent,
                                     args,
                                     n);

XtAddCallback(push_widget,
              XmNactivateCallback,
              callback_func,
              NULL);

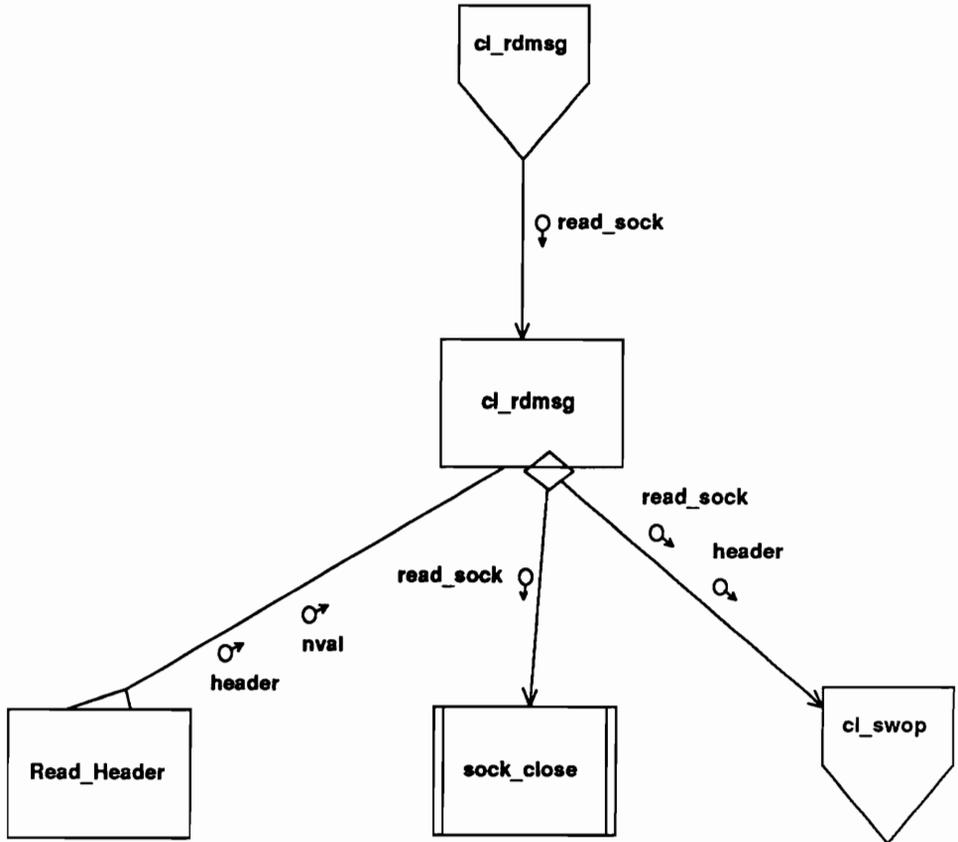
return(push_widget);
} /* — createpushbutton — */

```

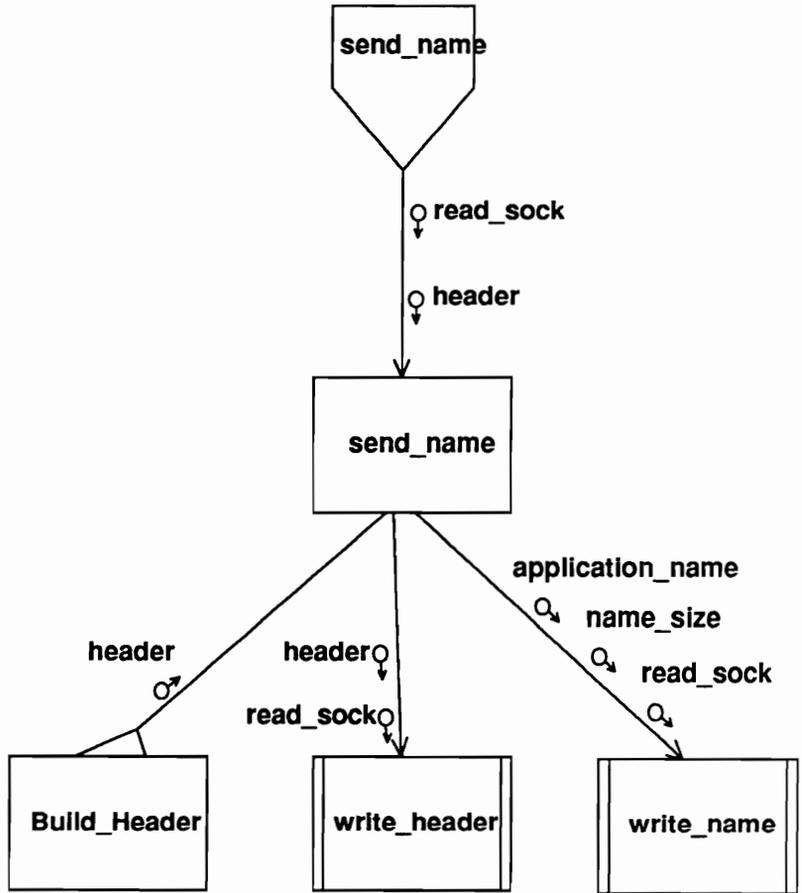
APPENDIX D: CLIENT APPLICATION STRUCTURE CHARTS / M-SPECS



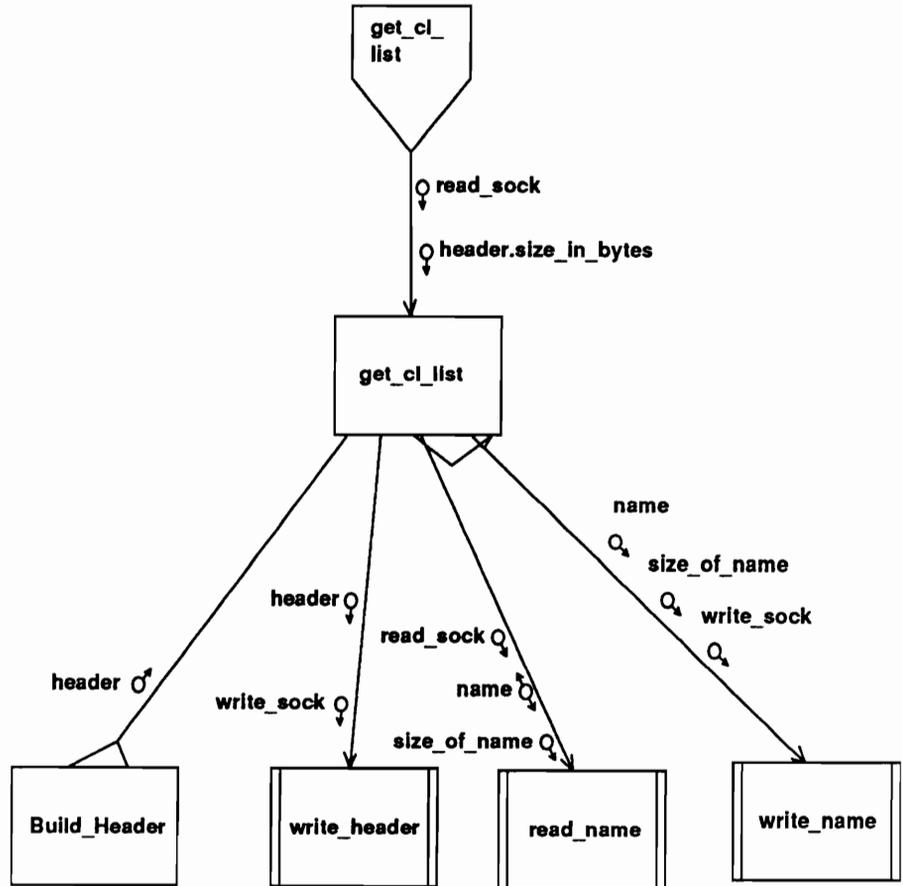
cl_rdmsg;1
No title



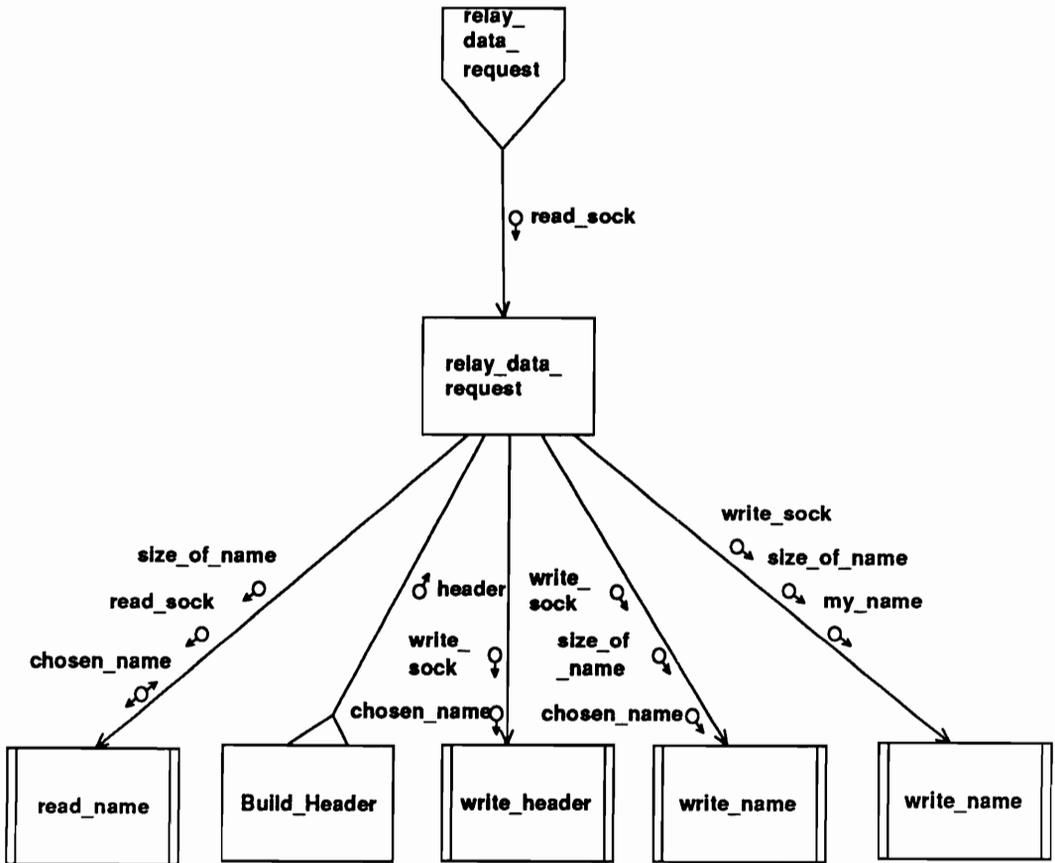
send_name;1
No title



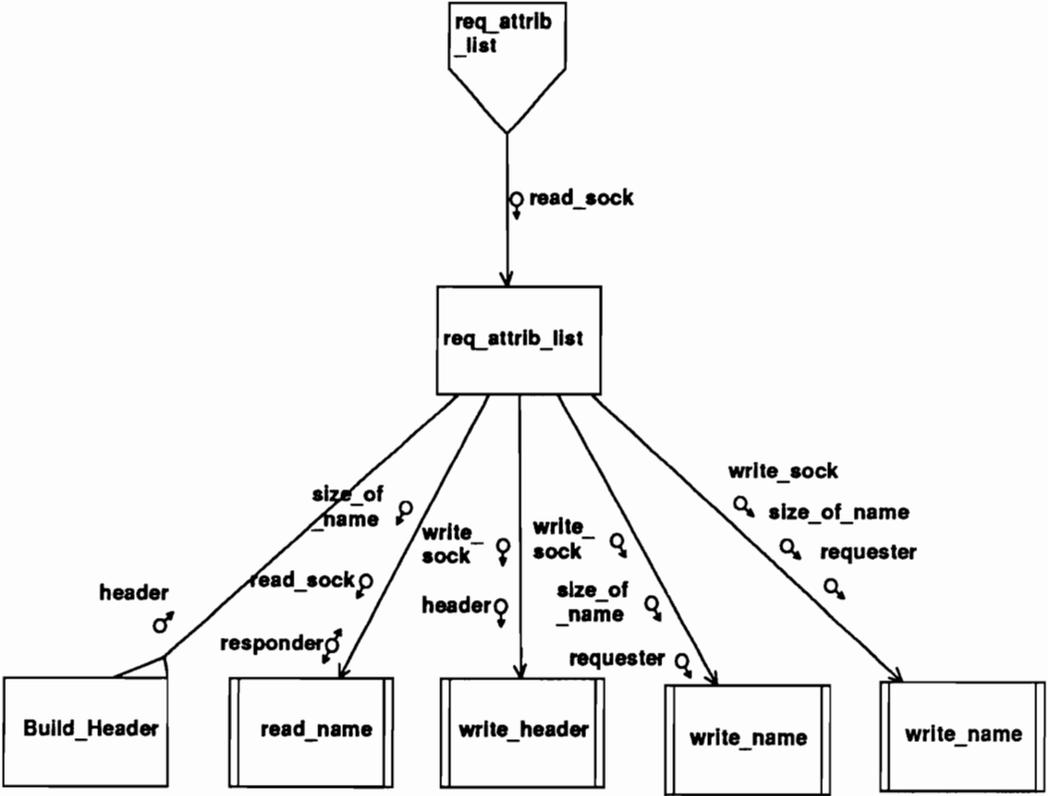
get_cl_list;1
No title



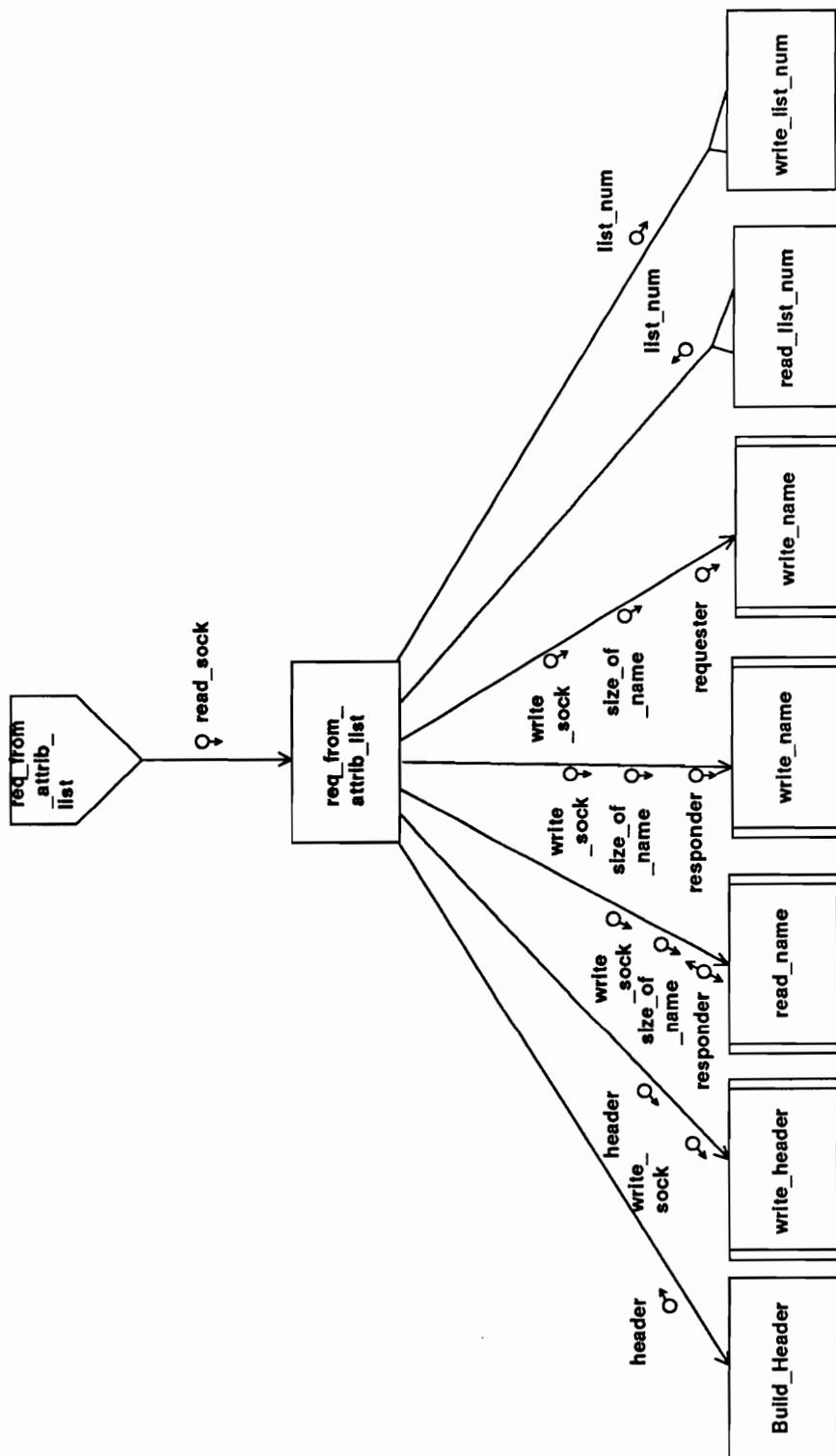
relay_data_request;2
No title



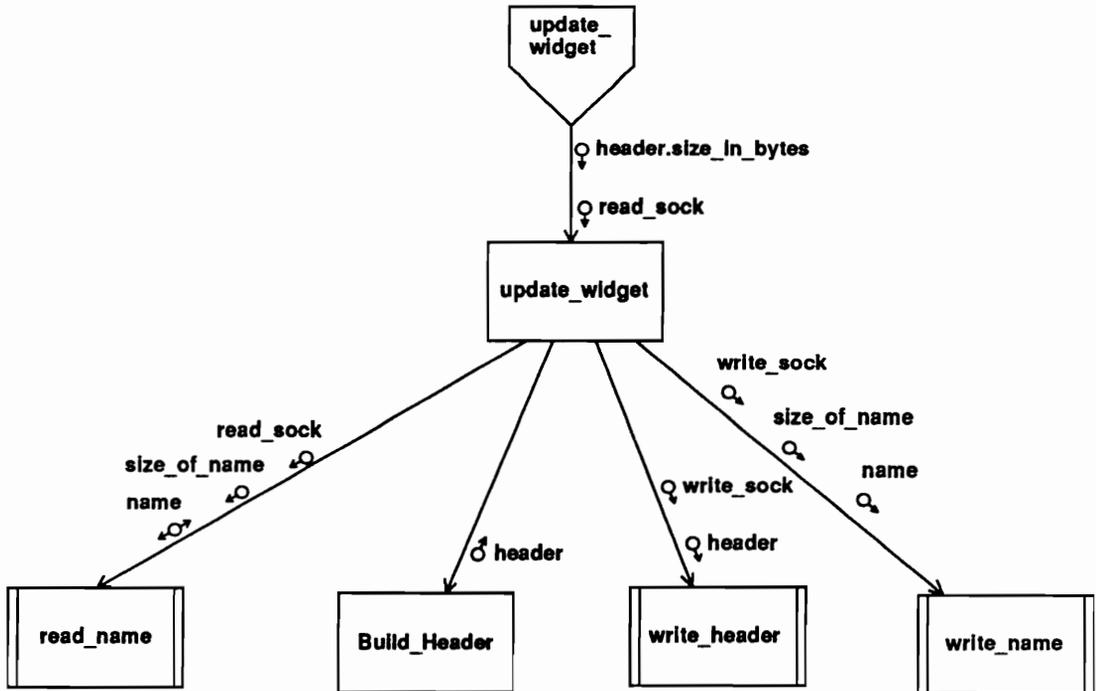
req_attrib_list;1
No title



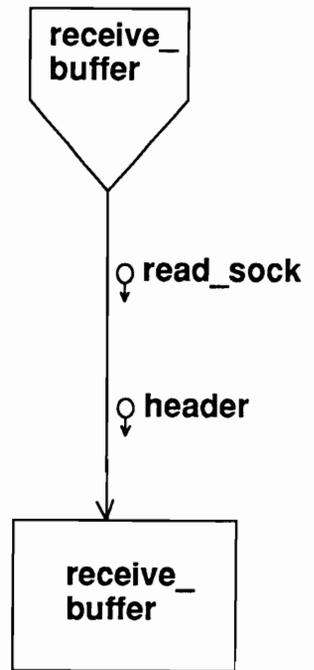
req_from_attrb_list:1
 No title



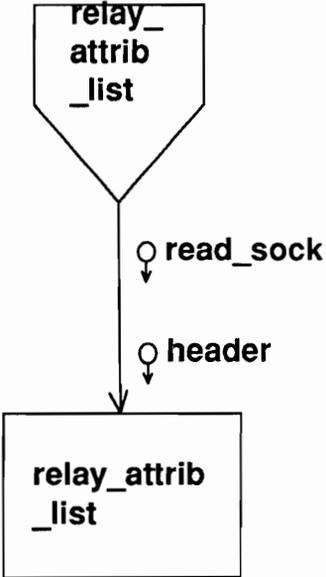
update_widget;1
No title



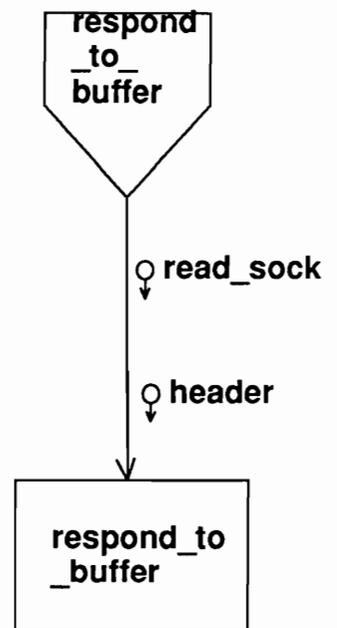
receive_buffer;1
No title



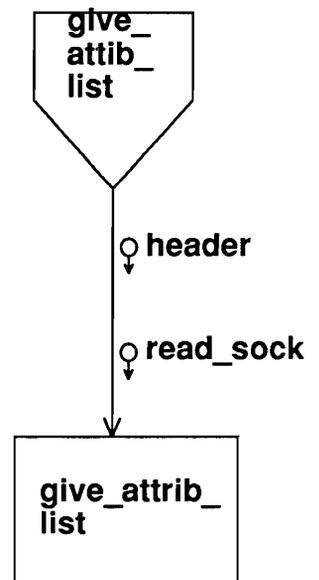
relay_attrib_list;1
No title



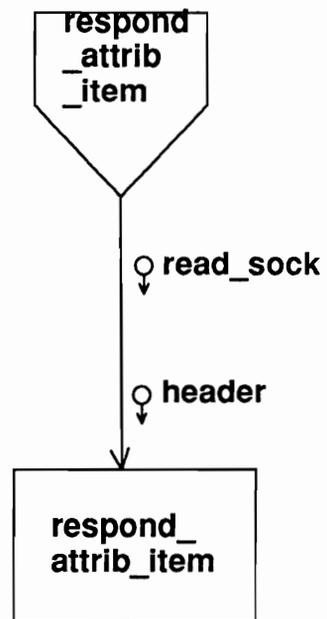
respond_to_buffer_request;1
No title



give_attrib_list;1
No title



respond_attrib_item;1
No title



NAME: ap_sock;6

TITLE: Integration client main module

PARAMETERS:

ACTIVE_SERVER : data_out
ACTIVE_WIDGET : data_out
LOCALS: server
gethostbyname
Host
Pid
c
len
i
one
errno
handle_me
end_handle
grimmy
grim_len

BODY:

```
/* =====  
Source Code Filename: ap_sock.c  
Special Considerations:  
Sets up global definitions  
for data which the client interface and the application  
will have in common.  
Purpose:  
To open asynchronous sockets with the server and the  
GRIM widget and set up an event handler which will  
act when a signal comes in on one of those sockets.  
Belongs to client application  
===== */  
/  
*=====*/  
/*  
ap_sock.c  
Function: to act as the AP/SOCK interface between the B-Spline  
Toolkit and the integration server located on Port 2000  
Variables: none at the moment  
Coded by: Michele Grieshaber  
Date : 06/10/91  
edited the 25th of August for use with client-based GRIM  
*/  
/  
*=====*/  
#define _BSD  
#define TRUE 1  
#define FALSE 0  
#include <stdio.h>
```

```

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <fcntl.h>
#include <sys/file.h>
#include <sys/un.h>
#include <signal.h>
#include <errno.h>
#include "../mysock2.h"
#include "/u/knight/show/execs/showtime.h"
#define HostName "cadrt9"
#define Port 2000
#define Port2 2002
#define HostName2 "cadrt9"

void main_ap();
void cl_rdmmsg();
void ask_for_widget();

/* — global declarations so the handler can understand variables used — */
int Sock;
int Sock2;
MODEL *Model = (MODEL *)NULL;
char pathname[] = "../execs/s.grimsock";
int ACTIVE_WIDGET = TRUE;
int ACTIVE_SERVER = TRUE;
/* _____ end global decs _____ */
/*****
/*
    handler1.c
    Function: to handle events that come across on the asynchronous
             socket which communicates with the server
    Variables: none can be passed in but it needs to know the socket
             descriptor for communication with the server and the
             location of the Model data structure so that information
             contained therein can be used by the b-spline toolkit.
    Coded by: Michele Grieshaber
    Date : 06/10/91
*/
/*****
void handler1(Signal, Code, SCP)
int Signal, Code;
struct sigcontext *SCP;
{
fd_set read_mask;      /* mask which tells select what to look for */
struct timeval to;     /* time value structure for the select call */
int rc;                /* return code */

```

```

int nsock;

/* do a select to see if there is indeed information on the socket to read*/
FD_ZERO(&read_mask);

/* set mask to include socket to server and to the GRIM */
FD_SET(Sock,&read_mask);
FD_SET(Sock2, &read_mask);

/* — clear out the timeval structure — */
bzero((char *)&to, sizeof(to));
to.tv_sec = 0;

/* determine if sock or sock2 greater */
if (Sock > Sock2)
{
    nsock = Sock;
} else {
    nsock = Sock2;
}

/* use select to check for socket activity */
rc = select(nsock+1, &read_mask, (fd_set *)0, (fd_set *)0, &to);
if (rc < 0)
{
    perror("select:");
} else if (rc > 0) {
    if(FD_ISSET(Sock, &read_mask))
    {
        cl_rdmsg(Sock);
    } else {
        cl_rdmsg(Sock2);
    }
}
return;
} /* — end handler1 — */
/*****
/*****
main()
{
struct sockaddr_in server;          /* server internet info */
struct hostent *Host, *gethostbyname(); /* host information */
int Pid;                            /* process ident */
int c, len, i ,one =1;
extern int errno;                   /* error number */
struct sigaction handle_me, end_handle; /* signal structures */
struct sockaddr_un grimmy;
int grim_len;

do
{

```

```

/* — create a socket for the client — */
Sock = socket(AF_INET,SOCK_STREAM,0);

if (Sock == -1)
{
    perror("Inet_Client:socket");
    exit(-3);
}
Sock2 = socket(AF_UNIX,SOCK_STREAM,0);
if (Sock2 == -1)
{
    perror("Inet_Client:socket");
    exit(-3);
}

/* — clear out the handle_me structure — */
bzero((char *)&handle_me, sizeof(handle_me));
handle_me.sa_handler =handler1;
sigaction(SIGIO, &handle_me, &end_handle);

/* — set up the socket to be nonblocking???????? — */
/* — set up the async event handler ————— */
/* — set the process receiving SIGIO signal to us — */
Pid = getpid();
if (ioctl(Sock,SIOCSPGRP,&Pid) == -1)
{
    perror("ioctl FIOSETOWN");
    exit(1);
}

/* — allow receipt of async i/o signals — */
if (ioctl(Sock, FIOASYNC, &one) < 0)
{
    perror("ioctl FIOASYNC:");
    exit(1);
}
if (ioctl(Sock2,SIOCSPGRP,&Pid) == -1)
{
    perror("ioctl FIOSETOWN");
    exit(1);
}

/* — allow receipt of async i/o signals — */
if (ioctl(Sock2, FIOASYNC, &one) < 0)
{
    perror("ioctl FIOASYNC:");
    exit(1);
}
Host = gethostbyname(HostName); /* resolves strg to internet address*/
if (Host == NULL)
{

```

```

    perror("Inet_Client;host");
    exit(-1);
}

bzero((char *)&server,sizeof(server)); /*binds socket to port 0 */
server.sin_family = AF_INET; /* set domain */
server.sin_port = Port; /* set port to connect */
bcopy(Host->h_addr, (char *)&server.sin_addr.s_addr,Host->h_length);

/* — copies the address of what he wants
to connect to into sin structure — */
bzero((char *)&grimmy, sizeof(grimmy));
grimmy.sun_family = AF_UNIX;
strcpy(grimmy.sun_path, pathname);
grim_len = strlen(grimmy.sun_path) + sizeof(grimmy.sun_family);

/* — try to connect to the server — */
c = connect(Sock, (char *)&server, sizeof(server));
if (c < 0)
{
    perror("connecting stream socket ");
} else {
    printf("ap_sock: connected to server\n");
}

/* connect the socket to the GRIM */
c = connect(Sock2,(struct sockaddr *)&grimmy,sizeof(struct sockaddr_un));
if (c < 0)
{
    perror("connecting stream socket 2");
    exit(1);
} else {
    printf("ap_sock: connected to grimmy\n");
}

} while (c < 0 && errno == ECONNREFUSED);

/* now that the connection has been established, must proceed to the
main portion of the application */
/* main_ap will call the main program of B-Spline Toolkit and start exec */
ask_for_widget(Sock);
main_ap();

}/* — end ap_sock.c — */

```

NAME: ask_for_widget;4

TITLE: IC ask_for_widget

PARAMETERS:

Sock : data_in

ACTIVE_SERVER : data_in

LOCALS: header

BODY:

/* =====

Source Code Filename: ask_for_widget.c

Special Considerations: NONE

Purpose:

Sends a request to the server for a list of clients connected to the integrated system from whom it can request data. This information when received will be relayed to the GRIM widget for display.

Belongs to client application

===== */

```
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include "/u/michele/sock/mysock.h"
```

```
extern int ACTIVE_SERVER; /* used to make sure server active */
void write_header();
```

```
void ask_for_widget(int sock)
{
HEADER header;
```

```
/* — ask the server to send info to client to pass to widget — */
header.size_in_bytes = 0;
header.maj_opcode = 0;
header.min_opcode = 1;
if (ACTIVE_SERVER)
{
write_header(sock, header);
} /* — end if active server — */
return;
} /* — end ask_for_widget.c — */
```

NAME: main_ap;3

TITLE: IC main_ap

PARAMETERS:

LOCALS:

BODY:

/* ===== */

Source Code Filename:

Special Considerations: NONE

Purpose:

===== */

/*=====*/

/*

 Name: main_ap.c

Author: Michele Grieshaber

 Date: 06/10/91

 Description: this subroutine is called by the AP/SOCK Interface
 to start the main routine of the B-Spline Toolkit.
 The data structure Model is initialized here and
 contains data pertinent to the geometry used in the
 Toolkit.

*/

/*=====*/

#include <stdio.h>

#include "/u/knight/show/execs/showtime.h"

/* — supporting routines — */

void showtime();

/* — end supporting routines — */

void main_ap()

{

/* — start the b-spline toolkit — */

showtime();

return;

} /* — end main_ap.c — */

NAME: handler1;2

TITLE: IC event handler

PARAMETERS:

LOCALS:

BODY:

/* =====

Source Code Filename: part of ap_sock.c

Special Considerations: NONE

Purpose:

The handler allows the reception of asynchronous signals. It determines upon which socket the signal occurred using a read_mask. When a signal occurs, the main application is suspended, the handler is enabled, the signal is read and its message evaluated for handling. When the event has been properly handled, control is restored to the main application.

===== */

NAME: cl_rdmsg;4

TITLE: IC cl_rdmsg

PARAMETERS:

read_sock : data_in

LOCALS: header

nval

BODY:

/* =====

Source Code Filename: cl_rdmsg.c

Special Considerations: NONE

Purpose:

First read performed on a signal. Tries to read a "headers" worth of data from the socket. If there is no data on the socket, then it is perceived as a disconnection signal. When there is data, the header is read and sent for evaluation to a module which performs select_ops based on the header.maj_opcode and header.min_opcode portions of the header.

Belongs to client application

===== */

/

*===== */

```

/*
cl_rdmsg.c
Function: reads the incoming header of the socket connected to
          the server and sends it to cl_swop for resolution
Variables:   read_sock      - socket on which information is waitin
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
*=====*/
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include "/u/michele/sock/mysock.h"
#include "/u/knight/show/execs/showtime.h"

/* — function declarations — */
void cl_swop();
void sock_close();
/* — end function declarations — */

cl_rdmsg(int read_sock)
{
HEADER header;      /* header containing info size and maj/min opcodes */
int nval;           /* return code */

/* — read the header from the information sitting on the socket — */
nval = read(read_sock, &header, sizeof(HEADER));
if(nval == -1)
{
perror("cl_rdmsg: read");
exit(1);
} else if(nval == 0) {
/* — go to routine to close connection and take socket out of list - */
sock_close(read_sock);
} else {
/* — send the header to a routine which will switch based on opcodes — */
cl_swop(read_sock, header);
}
return;
} /* — end cl_rdmsg.c — */

```

NAME: sock_close;5

TITLE: IC sock_close

PARAMETERS:

LOCALS:

BODY:

```
/* =====  
Source Code Filename: sock_close.c  
Special Considerations: NONE  
Purpose:  
To close socket which is no longer active.  
Belongs to client application  
===== */
```

```
#include <stdio.h>  
#define FALSE 0  
#define TRUE 1  
  
extern int ACTIVE_WIDGET;  
extern int ACTIVE_SERVER;  
extern int Sock2;  
  
void sock_close(int dead_sock)  
{  
int i;  
  
/* — determine if the closed socket was to the server or widget — */  
if(dead_sock == Sock2) /* — widget is the dead socket — */  
{  
ACTIVE_WIDGET = FALSE;  
} else { /* — server is the dead socket — */  
ACTIVE_SERVER = FALSE;  
}  
return;  
} /* — end sock_close.c — */
```

```
*****
NAME: cl_swop;5
```

```
TITLE: IC specific cl_swop
```

```
PARAMETERS:
```

```
read_sock : data_in
```

```
header : data_in
```

```
ACTIVE_SERVER : data_in
```

```
Sock : data_in
```

```
LOCALS:
```

```
BODY:
```

```
/* =====
```

```
Source Code Filename: cl_swop.c
```

```
Special Considerations: NONE
```

```
Purpose:
```

```
To determine which module should handle
the data incoming on the socket. This determination
is based on a select_op (switches) performed on
the header.maj_opcode and header.min_opcode
portions of the header.
```

```
Belongs to client application
```

```
===== */
```

```
/
```

```
*=====*/
```

```
/*
```

```
cl_swop.c
```

```
Function: resolves the messages passed in from the server by using
opcodes to determine the required response
```

```
Variables:   read_sock      - socket on which to read and write
              header        - struct containing opcodes
```

```
Coded by: Michele Grieshaber
```

```
Date : 06/10/91
```

```
*/
```

```
/
```

```
*=====*/
```

```
#define _BSD
```

```
#define my_name "B-SPLINE TOOLKIT"
```

```
#define size_of_name 50
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socketvar.h>
```

```
#include <sys/socket.h>
```

```
#include <sys/uio.h>
```

```
#include <netinet/in.h>
```

```
#include <netdb.h>
```

```
#include "../mysock2.h"
```

```
#include "/u/knight/show/execs/showtime.h"
```

```
/* — function declarations — */
```

```
void get_cl_list();
```

```

void send_name();
void relay_data_request();
void rcv_acsynt();
void req_attrib_list();
void relay_attrib_list();
void req_from_attrib_list();
void message(); /* message routine belonging to B-Spline toolkit */
/* — end function declarations — */

extern int Sock;
extern int ACTIVE_SERVER;

void cl_swop(int read_sock, HEADER header)
{
/* char my_name[] = "B-SPLINE TOOLKIT";*/ /* name of the client */
int nval; /* return code */
int nu = 3, nw = 3; /* parametric u and w values */

/* — starting switch on major opcode ————— */
switch(header.maj_opcode)
{
case 0:
switch(header.min_opcode)
{
case 0:
/* the server is asking for information on the client to put— */
/* in a client_information structure ————— */
send_name(read_sock, header);
break; /* — break min case 0 — */

case 1:
break; /* — break for case 1 — */

case 2:
/* — get info from server and send to widget — */
get_cl_list(read_sock, header.size_in_bytes);
break; /* — break for case 2 — */

} /* — end switch — */
break; /* — end major case 0 — */

case 1:
/* — starting switch on minor opcode for major case 1 ————— */
switch(header.min_opcode)
{
case 0:
break;

case 1: /* case 1 under minor opcode is for the sender */
relay_data_request(read_sock);
break;
}
}
}

```

```

case 2:      /* case 2 under minor opcode is for the receiver */
break; /* — end maj case 1 minor case 2 — */

case 3:
/* — request client_attribute listing from server — */
req_attrib_list(read_sock);
break;

case 4:
req_from_attrib_list(read_sock);
break;
default:
message("BAD INFORMATION FROM THE SERVER...", 1);
break; /* — end maj case 1 minor default — */
} /* end switch(min_opcode) */
/* — end minor opcode switch for major case 1 ————— */
break; /* — end major case 1 — */

case 2:
switch(header.min_opcode)
{
case 0:
/* — update the widget with current info from server — */
update_widget(read_sock, header.size_in_bytes);
break;

case 1:      /* case 2 under minor opcode is for the receiver */
rcv_acsynt(read_sock, header);
break; /* — end maj case 1 minor case 2 — */

case 3:
relay_attrib_list(read_sock, header);
break;

} /* — end switch minor for major = 2 — */
break;
default:
message("BAD INFORMATION FROM THE SERVER...", 1);
break; /* end default for major opcode — */
} /* end switch(maj_opcode) */
/* — end major opcode switch ————— */
return;
} /* — end cl_swop.c — */

```

NAME: send_name;4

TITLE: IC send_name

PARAMETERS:

read_sock : data_in

header : data_in

LOCALS:

BODY:

```
/* =====  
Source Code Filename: send_name.c  
Special Considerations: Need to define client name  
at the top of this program.  
Separate modules of this  
program are necessary for  
each client for this reason.  
Purpose:  
To send the client application name to  
the requesting socket (the server or the  
GRIM widget)  
Belongs to client application  
===== */
```

```
#define _BSD  
#define my_name "B-SPLINE TOOLKIT"  
#define size_of_name 50  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socketvar.h>  
#include <sys/socket.h>  
#include <sys/uio.h>  
#include <netinet/in.h>  
#include <netdb.h>  
#include "./mysock2.h"  
  
/* — function declarations — */  
void write_header();  
void write_name();  
extern int ACTIVE_SERVER;  
  
void send_name(int read_sock, HEADER header)  
{  
header.size_in_bytes = sizeof(my_name);  
  
header.maj_opcode = 0;  
header.min_opcode = 0;  
if(ACTIVE_SERVER)  
{  
write_header(read_sock, header);  
write_name(read_sock, my_name, sizeof(my_name));  
} /* — end ACTIVE_SERVER — */
```

```
return;
} /* — end send_name.c — */
```

```
*****
```

```
NAME: get_cl_list;4
```

```
TITLE: IS get_cl_list
```

```
PARAMETERS:
```

```
read_sock : data_in
```

```
header.size_in_bytes : data_in
```

```
ACTIVE_WIDGET : data_in
```

```
Sock2 : data_in
```

```
LOCALS: name
```

```
i
```

```
header
```

```
BODY:
```

```
/* =====
```

```
Source Code Filename: get_cl_list.c
```

```
Special Considerations: NONE
```

```
Purpose:
```

```
To receive the members of a list containing application names from the server, and to send the list members on to the GRIM widget which belongs to the client who has just received the information.
```

```
In the get_cl_list structure chart, write_sock represents the GRIM widget's socket, which is called Sock2 in the source code.
```

```
Belongs to GRIM
```

```
===== */
```

```
#define _BSD
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socketvar.h>
```

```
#include <sys/socket.h>
```

```
#include <sys/uio.h>
```

```
#include <netinet/in.h>
```

```
#include <netdb.h>
```

```
#include <sys/un.h>
```

```
#include "../mysock2.h"
```

```
#include "/u/knight/show/execs/showtime.h"
```

```
#define size_of_name 50
```

```
extern int ACTIVE_WIDGET;
```

```
extern int Sock2;
```

```
void write_header();
```

```
void write_name();
```

```
char *read_name();
```

```

void get_cl_list(int read_sock, int size)
{
char *name;
int i;
HEADER header;

/* — only if the widget is active will this be performed — */
if (ACTIVE_WIDGET)
{
/* — send msg to widget telling it of incoming client_list — */
header.size_in_bytes = size;
header.maj_opcode = 0;
header.min_opcode = 2;
write_header(Sock2, header);
for (i = 0; i < size; i++)
{
/*name = (char *)malloc(size_of_name);*/
/* — read off info from the server — */
name = read_name(read_sock, name, size_of_name);
write_name = (Sock2, name, size_of_name);
} /* — end for size — */
} /* — end if ACTIVE_WIDGET — */

return;
} /* — end get_cl_list.c — */

```

NAME: relay_data_request;5

TITLE: IC relay_data_request

PARAMETERS:

read_sock : data_in

ACTIVE_SERVER : data_in

Sock : data_in

LOCALS: header

responder

requester

BODY:

/* =====

Source Code Filename: relay_data_request.c

Special Considerations:

The variable requester must be defined for every client in the integrated system, therefore, each client must contain a version of this module with its name defined (please note that this name must be identical to the one given in the application exchange relation file.)

Purpose:

To notify the application whose name was chosen to be the supplier of buffer data for the client of the impending request. In other words, the client sending the request wants to receive the buffer of the application chosen by the user. The chosen application is called responder in this function, since it is he who will respond to the request by providing the data.

Belongs to client application

```
===== *
#define _BSD
#define requester "B-SPLINE TOOLKIT"
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include "../mysock2.h"

/* — function declarations — */
char *read_name();
void write_header();
void write_name();
extern int ACTIVE_SERVER;
extern int Sock;

void relay_data_request(int read_sock)
{
    HEADER header;
    char *responder;

    /* — read the name from the widget to send to server — */
    responder = read_name(read_sock, responder, size_of_name);

    /* — make sure server is active — */
    if (ACTIVE_SERVER)
    {
        header.size_in_bytes = size_of_name;
        header.maj_opcode = 1;
        header.min_opcode = 1;
        write_header(Sock, header);
        write_name(Sock, responder, size_of_name);
        write_name(Sock, requester, size_of_name);
    } /* — end if ACTIVE SERVER — */

    return;
} /* — end relay_data_request.c — */
```

NAME: req_attrib_list;5

TITLE: IC req_attrib_list

PARAMETERS:

read_sock : data_in

ACTIVE_SERVER : data_in

Sock : data_in

LOCALS: header

responder

requester

BODY:

/* =====

Source Code Filename: req_attrib_list.c

Special Considerations:

Requester must be defined as the name of the owning application in a define statement at the top of this module, thus a unique module is needed per client in the integrated system.

Purpose:

To relay the request registered at the clientqs widget for the attribute list of the client called responder. Responder is one of the applications listed in the widgets exchange selection list.

Belongs to client application

===== */

#define _BSD

#define requester "B-SPLINE TOOLKIT"

#define size_of_name 50

#include <stdio.h>

#include <sys/types.h>

#include <sys/socketvar.h>

#include <sys/socket.h>

#include <sys/uio.h>

#include <netinet/in.h>

#include <netdb.h>

#include "../mysock2.h"

/* — function declarations — */

void write_header();

void write_name();

char *read_name();

extern int Sock;

extern int ACTIVE_SERVER;

void req_attrib_list(int read_sock)

{

HEADER header;

char *responder;

```

/* — send request for attribute list to server — */
header.size_in_bytes = 0;
header.maj_opcode = 1;
header.min_opcode = 3;

/* — read name of client from the widget signal — */
responder = read_name(read_sock, responder, size_of_name);

/* — relay this info to the server — */
write_header(Socket, header);
write_name(Socket, requester, size_of_name);
write_name(Socket, responder, size_of_name);

return;
} /* — end of req_attrib_list.c — */

```

NAME: req_from_attrib_list;4

TITLE: IC req_from_attrib_list

PARAMETERS:

read_sock : data_in

Socket : data_in

LOCALS: responder

requester

list_num

header

BODY:

/* =====

Source Code Filename: req_from_attrib_list.c

Special Considerations:

Requester must be defined as the name of the owning application, therefore each client in the integrated system needs to modify this file to contain his name in requester.

Purpose:

To request data as a function of an item from the attributes list. The attribute list was supplied in a previous request from a client in the integrated system, and this request is directed to that client, known in this function as responder.

Belongs to client application

===== */

#define _BSD

#define requester "B-SPLINE TOOLKIT"

```

#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include "../mysock2.h"

/* — external functions — */
char *read_name();
void write_header();
void write_name();
extern int Sock;

void req_from_attrib_list(int read_sock)
{
char *responder;
int list_num;
HEADER header;

/* — define and write header to server — */
header.size_in_bytes = 0;
header.maj_opcode = 1;
header.min_opcode = 4;
write_header(Sock, header);

/* — read the source name from widget and send to server — */
responder = read_name(read_sock, responder, size_of_name);
write_name(Sock, responder, size_of_name);

/* — send your name to the server — */
write_name(Sock, requester, size_of_name);

/* — read the component number and send to server — */
if(read(read_sock, &list_num, sizeof(int)) < 0)
{
perror("rfal: reading component number");
exit(1);
}
if (write(Sock, &list_num, sizeof(int)) < 0)
{
perror("rfal: write component number");
exit(1);
}
return;
} /* — end req_from_attrib_list.c — */

```

NAME: update_widget;4

TITLE: IC update_widget

PARAMETERS:

header.size_in_bytes : data_in

read_sock : data_in

ACTIVE_WIDGET : data_in

Sock2 : data_in

LOCALS: i

header

name

BODY:

/* =====

Source Code Filename: update_widget.c

Special Considerations: NONE

Purpose:

To send the name of a client who has just connected to the server, or disconnected, to the widget. The widget updates his selection list of exchange clients by either adding the name (in the case of a connect) or deleting it (disconnect). The name is accompanied by an ADD or DELETE flag. Belongs to client application

===== */

#define _BSD

#include <stdio.h>

#include <sys/types.h>

#include <sys/socketvar.h>

#include <sys/socket.h>

#include <sys/uio.h>

#include <netinet/in.h>

#include <netdb.h>

#include <sys/un.h>

#include "../mysock2.h"

#define size_of_name 50

void write_header();

void write_name();

char *read_name();

extern int Sock2;

extern int ACTIVE_WIDGET;

void update_widget(int read_sock, int action)

{

int i;

char *name;

HEADER header;

/* — perform only if the widget is active — */

```

if (ACTIVE_WIDGET)
{
/* — read off the new client list and send to widget — */
name = read_name( read_sock, name, size_of_name);
header.size_in_bytes = action;
header.maj_opcode = 2;
header.min_opcode = 0;
write_header(Sock2, header);
write_name(Sock2, name);
} /* — end if ACTIVE WIDGET — */

return;
} /* — end update_widget.c — */

```

NAME: receive_buffer;11

TITLE: IC specific read_buffer

PARAMETERS:

read_sock : data_in

header : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This module is one which is entirely client dependent. Each client in the integrated system must provide a module to receive buffer data sent by other clients in the system to the server - where the data is transformed into the format read into this module by the client. Included in this m-spec is a sample module called rcv_acsynt.c which is used by the B-Spline Toolkit to receive buffer data from ACSYNT.

Belongs to client application

===== */

#define _BSD

#define size_of_name 50

#include <stdio.h>

#include <sys/types.h>

#include <sys/socketvar.h>

#include <sys/socket.h>

#include <sys/uio.h>

#include <netinet/in.h>

```

#include <netdb.h>
#include "../mysock2.h"

/* — function declarations — */
void model_read();
void after_read();

void rcv_acsynt(int read_sock, HEADER header)
{
int nu = 3, nw = 3; /* parametric u and w values */

/* — the data sent across from the server in this case
is in the form of a linked list of Model data structures.
This portion of the program must clean up an old Model
if it exists, then proceed to place the information on
the socket into the Model linked list which is used to
contain the geometry in the B-Spline Toolkit ————— */
/* — read the info from the socket — */
model_read(read_sock,header.size_in_bytes);

/* — compute tangents and draw geometry — */
after_read(nu,nw);

return;
} /* — end rcv_acsynt.c — */
/
*=====*/
/*
model_read.c
Function: receives model elements from the server and reads them
into the Model structure for the B-Spline Toolkit .
Variables: read_sock - socket on which to read from server
number_of_models- number of components server sends
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
*=====*/
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include "../mysock2.h"
#include "/u/knight/show/execs/showtime.h"
#include "/u/knight/show/subdivide/intersect.h"

void clean_up();
extern MODEL *Model;

```

```

void model_read(int read_sock, int number_of_models)
{
int rc;                                /* return code */
int ii, i, j, k;
int nu = 3, nw = 3;                    /* rendering parameters */
static int name_size = 21;            /* size of component name fields */
comp_data *component, *newcomp;      /* ptrs to component structures */
float pt1, pt2, pt3;                 /* point info from server */

/* — the server will be sending a stream of information containing
a linked list of model structures ... the size represents
the number of model structures contained in the list _____ */
clean_up();

/* ——— Initialize Structure IDs to Zero ——— */
Model->acs_root = -1;
Model->nubs_root = -1;
Model->fillet_root = -1;
Model->int_root = -1;
Model->intlst = (intersection *)NULL;
Model->num_comp = number_of_models;
for(ii = 1; ii <= number_of_models; ii++)
{
/*———— allocate space for new component —————*/
newcomp = (comp_data *)malloc(sizeof(comp_data));
if(ii ==1)
{
Model->comp = newcomp;
} else {
component->next = newcomp;
}
newcomp->existence = 1; /* set existence to yes */
newcomp->nu = nu; /* initialize rendering */
newcomp->nw = nw;

/* ——— Read in Component Information ——— */
rc=read(read_sock,newcomp->comp_name, name_size);
if(rc < 0)
{
perror("read_model: read comp name ");
exit(1);
}
rc=read(read_sock,&(newcomp->comp_number),sizeof(int));
if(rc < 0)
{
perror("read_model: read comp number");
exit(1);
}
rc=read(read_sock,&(newcomp->color),sizeof(int));
if(rc < 0)
{

```

```

    perror("read_model: read comp color");
    exit(1);
}
rc=read(read_sock,&(newcomp->acs_ncross),sizeof(int));
if(rc < 0)
{
    perror("read_model: read acs_ncross");
    exit(1);
}
rc=read(read_sock,&(newcomp->acs_npts),sizeof(int));
if(rc < 0)
{
    perror("read_model: read acs_npts");
    exit(1);
}

/* ----- Allocate Array for hermite points ----- */
newcomp->acs_pts = (float ***)calloc(newcomp->acs_ncross,
sizeof(float **));
newcomp->acs_utan = (float ***)calloc(newcomp->acs_ncross,
sizeof(float **));
newcomp->acs_wtan = (float ***)calloc(newcomp->acs_ncross,
sizeof(float **));
for ( i = 0 ; i < newcomp->acs_ncross ; i++ )
{
    newcomp->acs_pts[i] = (float **)calloc(newcomp->acs_npts,
sizeof(float *));
    newcomp->acs_utan[i] = (float **)calloc(newcomp->acs_npts,
sizeof(float *));
    newcomp->acs_wtan[i] = (float **)calloc(newcomp->acs_npts,
sizeof(float *));
    for ( j = 0 ; j < newcomp->acs_npts ; j++ )
    {
        newcomp->acs_pts[i][j] = (float *)calloc(3,sizeof(float));
        newcomp->acs_utan[i][j] = (float *)calloc(3,sizeof(float));
        newcomp->acs_wtan[i][j] = (float *)calloc(3,sizeof(float));
    } /* — end for j — */
} /* — end for i — */

/* —— Read in point data —— */
for ( i = 0 ; i < newcomp->acs_ncross ; i++ )
{
    for ( j = 0 ; j < newcomp->acs_npts ; j++ )
    {
        rc = read(read_sock, &(newcomp->acs_pts[i][j][0]),
                sizeof(float));

        if (rc < 0)
        {
            perror("model_read: reading pts");
        }
        rc = read(read_sock,&(newcomp->acs_pts[i][j][1]),

```

```

                                sizeof(float));
    if (rc < 0)
    {
        perror("model_read: reading pts");
    }
    rc = read(read_sock,&(newcomp->acs_pts[i][j][2]),
                                sizeof(float));
    if (rc < 0)
    {
        perror("model_read: reading pts");
    }
}/* — end for j — */
}/* — end for i — */
component = newcomp; /* reset pointers */
} /* — end for ii — */

component->next = (comp_data *)NULL; /* set last pointer to NULL */
return;
} /* — end model_read — */
/
*=====*/
/*
after_read.c
Function: after Model information is read from the server, the
          tangents are computed and the hermite geometry is
          drawn
Variables:  nu      - parametric variable in u direction
            nw      - parametric variable in w direction
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
*=====*/
#include <stdio.h>
#include "/u/knight/show/execs/showtime.h"

/* — function declarations — */
void message(); /* — all of these external funcs are in B-spline module. Not included in this code — */
void draw_hermite();
void acs_tangents();
/* — end function declarations — */

extern MODEL *Model;

void after_read (int nu, int nw)
{
/* — with the model structure full oth things must be done — */
acs_tangents(Model); /* calculate hermite tangents */

/* — draw the wireframe geometry — */
message("DATA TRANSFER FROM SERVER SUCCESSFUL", 1);

```

```

message("CREATING WIRE FRAME GEOMETRY...", 1);
draw_hermite(Model,nu,nw);

return;
} /* — end after_read — */
/
=====*/
/*
clean_up.c
Function: checks to see if a current model exists and cleans it
         out if one does
         Memory allocation is performed for the Model .
Variables: none at the moment
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
=====*/
#include <stdio.h>
#include "/u/knight/show/execs/showtime.h"

/* — function declarations — */
void clean_model();
void message();
/* — end function declarations — */

extern MODEL *Model;

void clean_up()
{
/* — check to see if current model is full — */
if(Model != NULL)
{
message("CLEANING UP OLD MODEL",1);
clean_model(Model); /* clear out old Model */
} /* — end if Model — */

/* — allocate memory for the model — */
Model = (MODEL *)malloc(sizeof(MODEL));

return;
} /* — end clean_up — */

```

NAME: relay_attrib_list;4

TITLE: IC specific relay_attrib_list

PARAMETERS:

read_sock : data_in

header : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This module is responsible for passing members of an attribute list sent by another client in the system to his GRIM widget. The structure of the module is entirely dependent on the corresponding module in the server, which in turn is dependent on the way in which the client sending the list outputs its data. As an example, a module used to by the B-Spline Toolkit to relay the attribute list sent by ACSYNT is included in this m-spec. Belongs to client application

===== */

```
#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/un.h>
#include "../mysock2.h"
```

/* — function declarations — */

```
void write_header();
void write_name();
char *read_name();
extern int Sock2;

void relay_attrib_list(int read_sock, HEADER header)
{
int i;
int comp_num;
char *comp_name;
char *responder;
char *requester;
```

```

int size_of_compname = 21;
/* — relay this information to the widget — */
/* header is the same as it was from the server ... size_in_bytes = ncomps
    ... maj_opcode = 2
    ... min_opcode = 3
    _____ */

write_header(Sock2, header);
requester = read_name(read_sock, requester, size_of_name);

/* — read responding clients name and send it to the GRIM — */
responder = read_name(read_sock, responder, size_of_name);
write_name(Sock2, responder, size_of_name);

for(i = 0; i < header.size_in_bytes; i++)
{
    if (read (read_sock, &comp_num, sizeof(int)) < 0)
    {
        perror(" make_att_l: reading comp_num");
        exit(1);
    }

    /* — write comp num to grim — */
    if (write(Sock2, &comp_num, sizeof(int)) < 0)
    {
        perror("relay_att_list: writing comp_num");
        exit(1);
    }

    /* — write the word component to grim — */
    comp_name = read_name(read_sock, comp_name, size_of_compname);
    write_name(Sock2, comp_name, size_of_compname);
} /* — end for ncomps — */

free(comp_name);
free(responder);
free(requester);
return;
} /* — end relay_attrib_list.c — */

```

NAME: respond_to_buffer,9

TITLE: IC specific respond_to_buffer

PARAMETERS:

read_sock : data_in

header : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This module is responsible for responding to a request for the current buffer data of the application. The module will send data to a module in the server which will read in the data in the order it was sent from this module, and transform it, before sending it, into the format of the receiving client. Please note that the header information set in this module is very important as it will enable the server to determine which module will accept this buffer information.

This module is highly application dependent and will be written on a case-by-case basis. As an example, the module respond_to_buffer.c which is part of the ACSYNT integrated client is included in this m-spec. It is suggested that the header defined in this example be used, unless a new minor opcode is defined and carried through for communication with the server and receiving client.

Belongs to client application

===== */

#define _BSD

#define size_of_name 50

#include <stdio.h>

#include <sys/types.h>

#include <sys/socketvar.h>

#include <sys/socket.h>

#include <sys/uio.h>

#include <netinet/in.h>

#include <netdb.h>

#include "../mysock2.h"

void acs_hermite();

char *read_name();

void respond_to_request(int read_sock)

```

{
char *request_name;

/* — responds to the serverqs appeal for data — */
/* — read the requesterqs name off of the socket — */
request_name = read_name(read_sock, request_name, size_of_name);
acs_hermite(read_sock, request_name);

return;
} /* — end resp_to_bspline.c — */
/*=====*/
#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include “./mysock2.h”

/* — function declarations for RS/6000 — */
void gtgmpk();
void gticmp();
void wr_herm();
void mid_herm();

void acs_hermite(int read_sock, char *requester_name)
{
int i, j;
int icomp,          /* — counter for components — */
    err,           /* — error return code — */
    ncomps,        /* — number of components in model — */
    comps[150],    /* — array of component numbers — */
    glob,          /* — global flag — */
    newnum,        /* — update component number for glob sym — */
    gsym,          /* — global symmetry — */
    comnum;        /* — ? — */
static int nglob = 14;
HEADER header;     /* — header used for protocol send to svr — */

/* — get the component list — */
(void) gtgmpk(&ncomps, comps);

/* — compensate for additional components if global symmetry exists — */
newnum = ncomps;
for (icomp = 0; icomp < ncomps ; icomp++)
{
    (void) gticmp(&nglob, &comps[icomp], &glob, &err);
    /* — check for global symmetry — */
    if ( glob != 0 )

```

```

    {
        newnum += 1;
    }
} /* — end for icoomp — */

/* — write out header to send to server — */
header.size_in_bytes = newnum;
header.maj_opcode = 2;
header.min_opcode = 1;

if(write(read_sock, &header, sizeof(HEADER)) < 0)
{
    perror("acs_hermite: writing header");
    exit(1);
}

/* — send back the requesterqs name — */
if (write(read_sock, requester_name, size_of_name) < 0)
{
    perror("acs_hermite: writing requester name");
    exit(1);
}

/* — initialize the starting component number — */
comnum = 1;

/* — loop through the components — */
for (icoomp = 0; icoomp < ncomps ; icoomp++)
{
    mid_herm(read_sock, comps[icoomp]);
} /* — end for icoomp — */

return;
} /* — end acs_hermite — */
/* ===== */
#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"
/* — function declarations for RS/6000 — */
void gticmp();
void gtccomp();
void ckhtmls();
void getint();
void trancd();
void rshmls();

```

```

void gtsmmt();
void vt3();
void wr_herm();

void mid_herm( int read_sock, int comp_num)
{
int i, j;
int icoomp,
nxsect,
nppxs,
err,
dummy,
ncomps,
comps[150],
locsym,
glob,
newnum,
gsym,
comnum;
float pt_list[30][30][3],
gsymm[4][4],
newpts[30][30][3];
float po[3],
pi[3];
static int nxsitm = 5,
nppitm = 6,
ngsym = 14,
nglob = 14;

/* — get the number of x-secs for the component — */
(void) gticmp(&nxsitm, &comp_num, &nxsect, &err);

/* — get the number of points per x-sec — */
(void) gticmp(&nppitm, &comp_num, &nppxs, &err);

/* — check for local symmetry in the component — */
(void) ckhmls( &comp_num, &locsym);

/* — get the points of the component — */
(void) getint( &comp_num, &nxsect, &nppxs, pt_list);

/* — reset the local symmetry of the component — */
(void) rshmls(&comp_num, &locsym);

/* — transform the coordinates of the points — */
(void) trandc( &comp_num, &nxsect, &nppxs, pt_list);

for ( i = 0; i < nxsect; i++ )
{
for ( j = 0; j < nppxs ; j++ )
{

```

```

    } /* — end for j — */
} /* — end for i — */
/* — invert the ordering of the points so the normals face out — */
for ( i = 0; i < nxsect; i++ )
{
    for ( j = 0; j < nppxs ; j++ )
    {
        newpts[j][i][0] = pt_list[nppxs-j-1][i][0];
        newpts[j][i][1] = pt_list[nppxs-j-1][i][1];
        newpts[j][i][2] = pt_list[nppxs-j-1][i][2];
    } /* — end for j — */
} /* — end for i — */

/* — write the component data to the server — */
wr_herm(read_sock, comp_num, ncomps, nxsect, nppxs, newpts, &comnum);

/* — get hte global symmetry flag for the component — */
gticmp( &ngsym, &comp_num, &gsym, &err);
if(( gsym >= 1) & (gsym <= 3))
{
    /* — get the global symmetry matrix — */
    (void) gtsmmt( &gsym, gsymmt);
    /* — get the point to be transformed — */
    for ( i = 0; i < nxsect; i++ )
    {
        for ( j = 0; j < nppxs; j++ )
        {
            pi[0] = pt_list[j][i][0];
            pi[1] = pt_list[j][i][1];
            pi[2] = pt_list[j][i][2];

            /* — transform the point — */
            (void) vt3(&pi[0],&pi[1],&pi[2],gsymmt,&po[0],&po[1],&po[2]);

            /* — get the transformed points — */
            pt_list[j][i][0] = po[0];
            pt_list[j][i][1] = po[1];
            pt_list[j][i][2] = po[2];
        } /* — end for j — */
    } /* — end for i — */
    /* call routine to write all this wonderful information to server */
    wr_herm(read_sock,comp_num,ncomps, nxsect,nppxs,pt_list,&comnum);
} /* — end if gsym — */
return;
} /* — end mid_herm.c — */
/* ===== */
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

```

```

#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"

/* — function declarations for RS/6000 — */
void gtccmp();

void wr_herm (int read_sock, int comp, int ncomps, int nxsect, int nppxs,
              float pt_list[][30][3], int *comnum)
{
int dummy;
int i, j;
int color,
err;
static int nxsitm = 5,
clitm = 11,
one = 1;
char comp_name[21];

/* — get the component name — */
/* bzero((char *)comp_name, sizeof(comp_name));*/
for (i = 0; i < 20; i++)
{
    comp_name[i] = '\0';
}

comp_name[20] = '\0';

/* — get the component name for each component — */
(void) gtccmp(&one, &comp, comp_name, &err);
if (write(read_sock, comp_name, sizeof(comp_name)) < 0)
{
    perror("acs_hermite: write comp name");
    exit(1);
}

/* — write the component number — */
if (write(read_sock, comnum, sizeof(int)) < 0)
{
    perror("acs_hermite: write comnum");
    exit(1);
}

/* — get the component color — */
(void) gtccmp(&clitm, &comp, &color, &err);
if (write(read_sock, &color, sizeof(int)) < 0)
{
    perror("acs_hermite: write color");
    exit(1);
}
}

```

```

/* — write the number of x-secs for the component — */
if (write(read_sock, &nxsect, sizeof(int)) < 0)
{
    perror("acs_hermite: write nxsect");
    exit(1);
}

/* — write the number of points per cross section — */
if (write(read_sock, &nppxs, sizeof(int)) < 0)
{
    perror("acs_hermite: write nppxs");
    exit(1);
}

/* — write out the list of points — */
for (i = 0; i < nxsect; i++)
{
    for (j = 0; j < nppxs; j++)
    {
        if(write(read_sock, &pt_list[j][i][0], sizeof(float)) < 0)
        {
            perror("wr_herm: write pt_list 0\n");
            exit(1);
        }
        if(write(read_sock, &pt_list[j][i][1], sizeof(float)) < 0)
        {
            perror("wr_herm: write pt_list 1\n");
            exit(1);
        }
        if(write(read_sock, &pt_list[j][i][2], sizeof(float)) < 0)
        {
            perror("wr_herm: write pt_list 2\n");
            exit(1);
        }
    } /* — end for j — */
} /* — end for i — */

/* — increment comnum — */
*comnum += 1;

return;
} /* — end wr_herm.c — */

```

NAME: give_attrib_list;4

TITLE: IC give_attrib_list

PARAMETERS:

header : data_in

read_sock : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This module is responsible for supplying an attribute list to send to clients in the integrated system which request one. The order and form in which this list is sent is important, for it is how the server will read the data and subsequently send it on to the requesting client. This module is application dependent and will be written on a case-by-case basis. As an example, the module give_attrib_list.c which is included in the ACSYNT client application is included in this m-spec. Please note the relationship of the header definition with the server and subsequently the serverqs header definition for transmission of this data to the responding client.

Belongs to client application

===== */

```
#define _BSD
#define my_name "ACSYNT"
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include "../mysock2.h"

/* — function declarations — */
void write_header();
void write_name();
char *read_name();
void gtgmpk();

void give_attrib_list( int read_sock)
{
```

```

HEADER header; /* — header for protocol msg to server — */
int ncomps;      /* — number of components in model — */
int icoomp,     /* — counter for components — */
comps[150];     /* — array of component numbers — */
char *requester; /* — name of requesting client — */
char *responder; /* — name of responding client — */
char cname[] = "geometric component";
int size_of_compname = 21;

/* — determine the current components of the model displayed — */
/* — get the component list — */
(void) gtgmpk(&ncomps, comps);

/* — set up header — */
header.size_in_bytes = ncomps;
header.maj_opcode = 2;
header.min_opcode = 3;

/* — write header to the server — */
write_header(read_sock, header);

/* — read requesting client's name off of the socket — */
requester = read_name(read_sock, requester, size_of_name);
write_name(read_sock, requester, size_of_name);

/* — read responding client's name off of the socket — */
responder = read_name(read_sock, responder, size_of_name);
write_name(read_sock, responder, size_of_name);

for (icoomp = 0; icoomp < ncomps ; icoomp++)
{
    /* — write the component number — */
    if (write(read_sock, &comps[icoomp], sizeof(int)) < 0)
    {
        perror("give_a_1: write icoomp");
        exit(1);
    }
    write_name(read_sock, cname, size_of_compname);
} /* — end for icoomp — */

free(requester);
free(responder);
return;
} /* — end give_attrib_list.c — */

```

NAME: respond_attrib_item;7

TITLE: IC specific respond_attrib_item

PARAMETERS:

read_sock : data_in

header : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This module is responsible for furnishing data as a reply to a request from a client who has sent a list item identifier from his attribute list. If an attribute list exists for an application, so must a module to handle the task of passing item related data to a requesting client.

As an example, the module respond_attrib_item.c which belongs to the client application ACSYNT is included in this m-spec. Special attention should be paid to the header definitions, since the server uses them to determine the module which will relay this information to the client.

Belongs to client application

===== */

```
#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include “./mysock2.h”
```

/* — external functions — */

```
char *read_name();
void write_header();
void write_name();
void gticmp();
```

```
void respond_attrib_list(int read_sock)
{
int comp_num;
HEADER header;
char *requester_name;
int gsym,
```

```

err;
static int ngsym = 14;

/* — read requesterqs name and send it later— */
requester_name = read_name(read_sock, requester_name, size_of_name);

/* — read the component number — */
if (read(read_sock, &comp_num, sizeof(int)) < 0)
{
    perror("ral: reading comp_num");
    exit(1);
}

/* — check if this component has global symmetry — */
(void) gticmp(&ngsym, &comp_num, &gsym, &err);
if (gsym != 0)
{
    header.size_in_bytes = 2;
} else {
    header.size_in_bytes = 1;
}

/* — set up header and send it — */
header.maj_opcode = 2;
header.min_opcode = 1;
write_header(read_sock, header);
write_name(read_sock, requester_name, size_of_name);

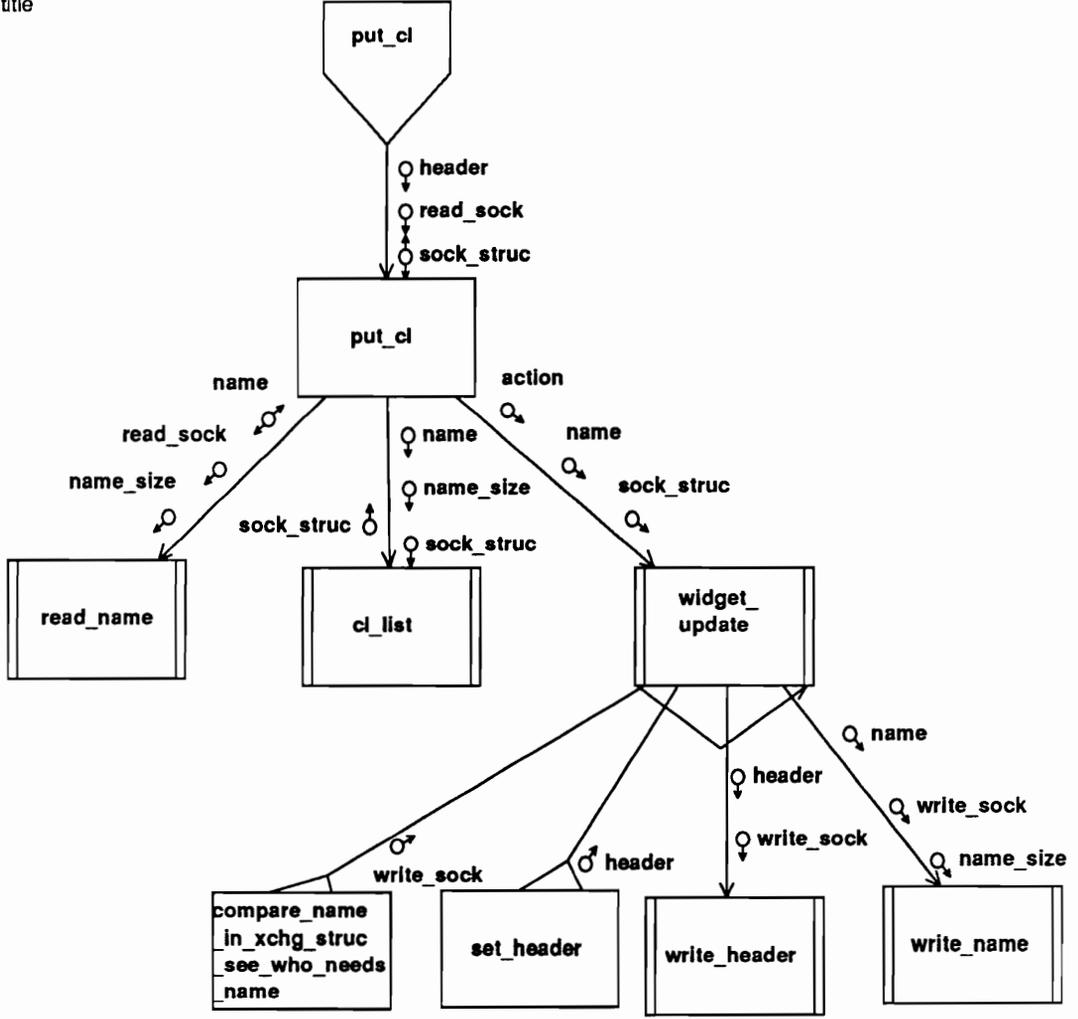
/* — send component number to only chose that one — */
mid_herm(read_sock, comp_num);
return;
} /* — end repond_attrib_list.c — */
/* ===== */
midherm.c is contained in the respond_to_request module spec which appears before this m-spec.

/* ===== */

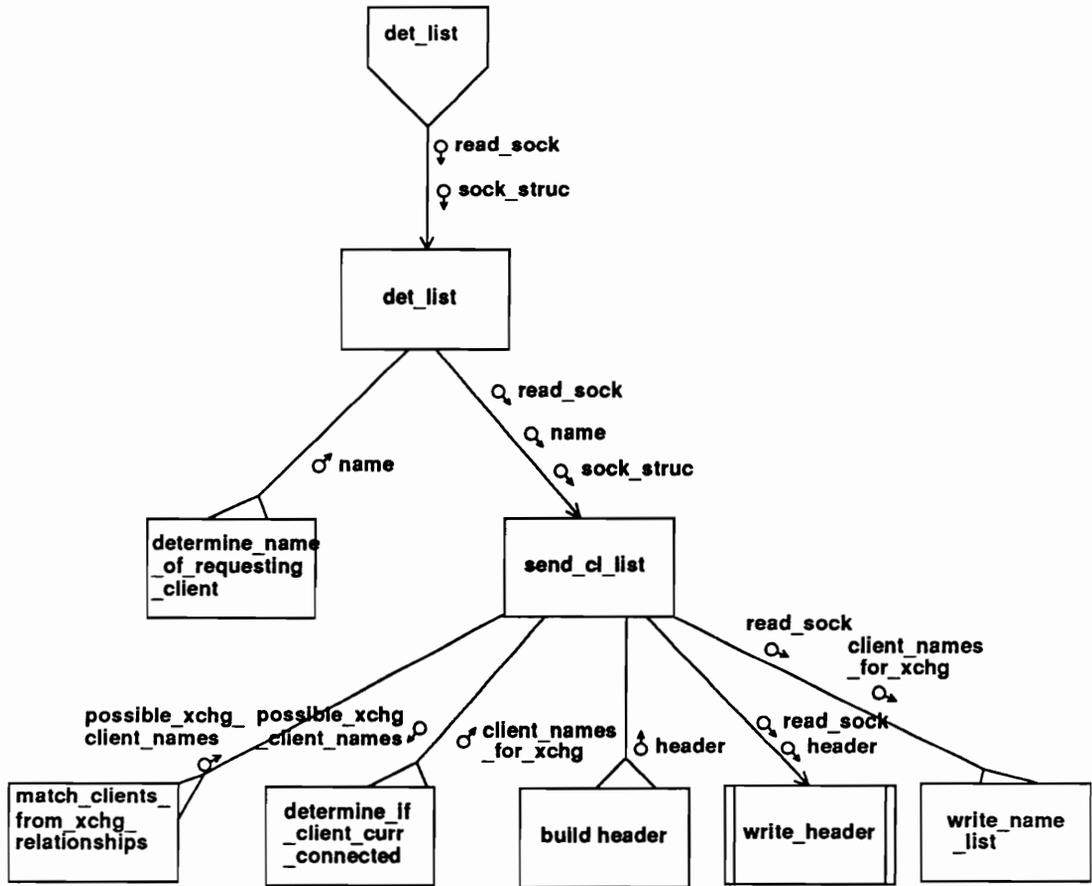
```

APPENDIX E: INTEGRATION SERVER STRUCTURE CHARTS / M-SPECS

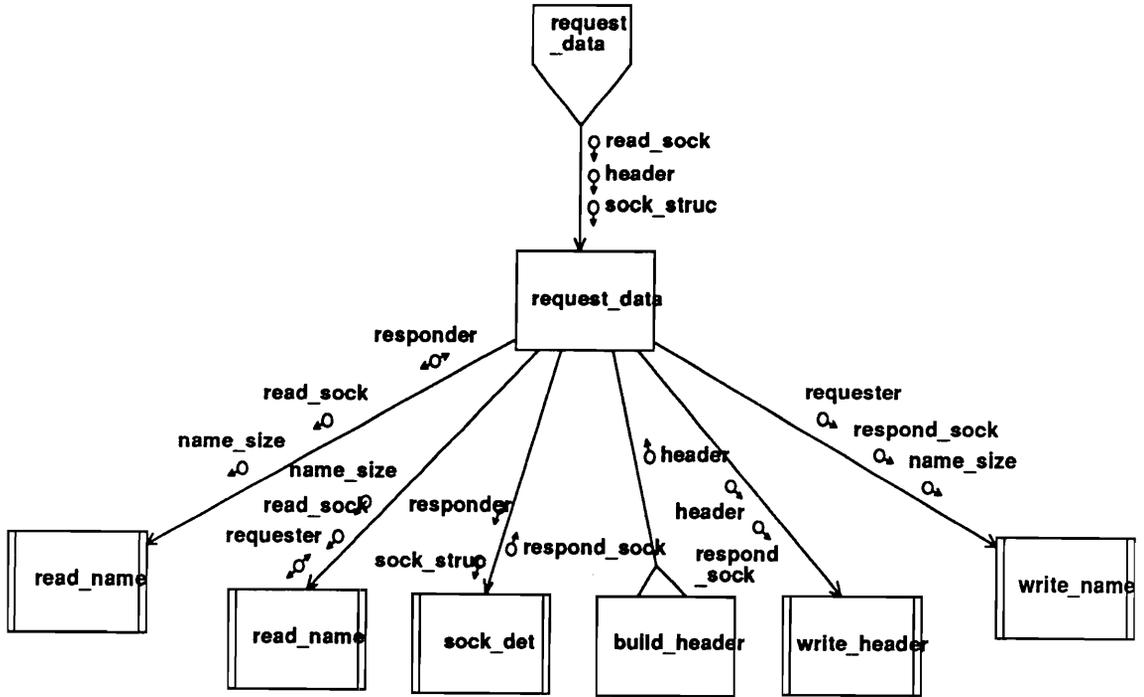
put_cl;2
 No title



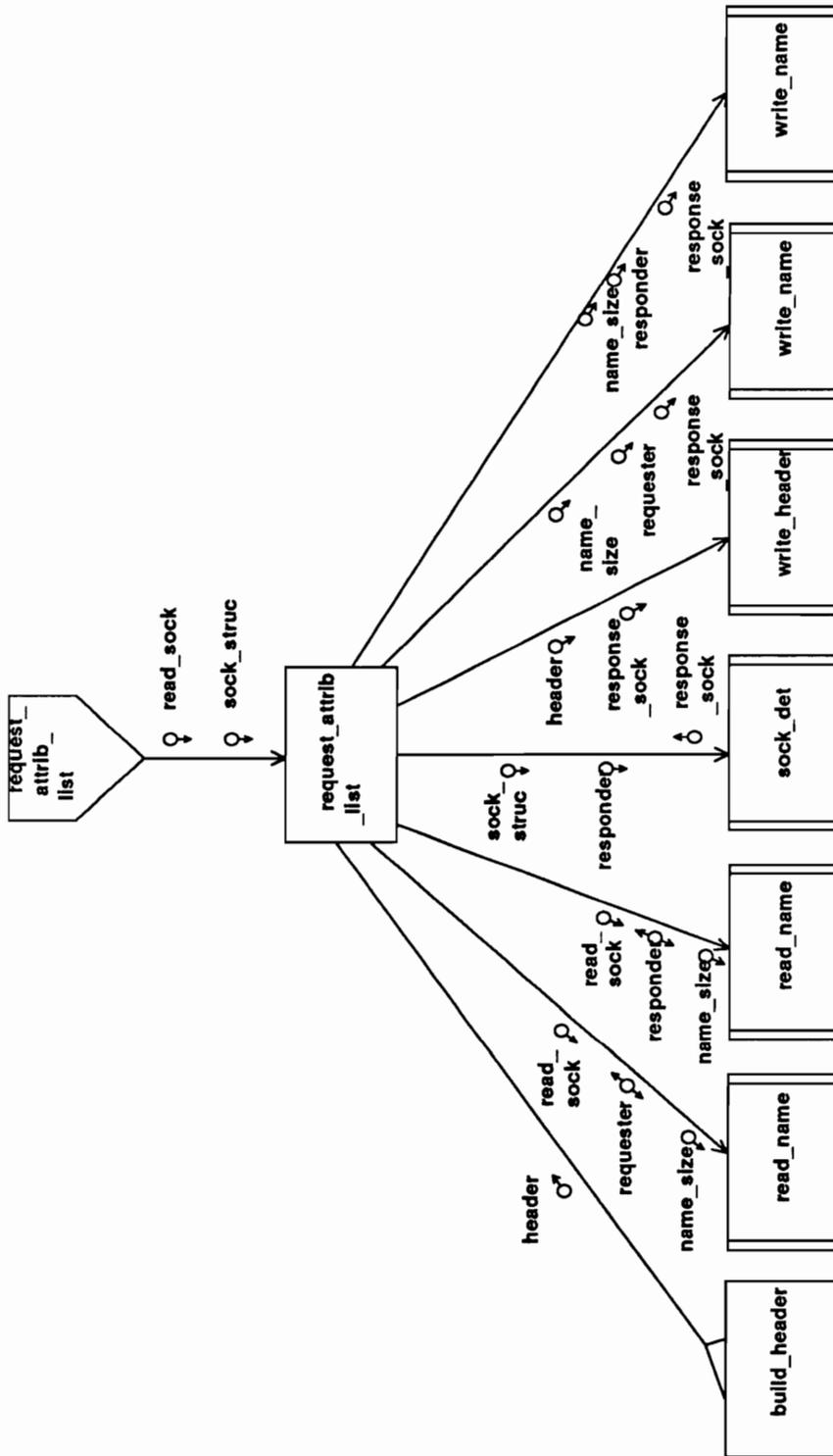
det_list;1
No title

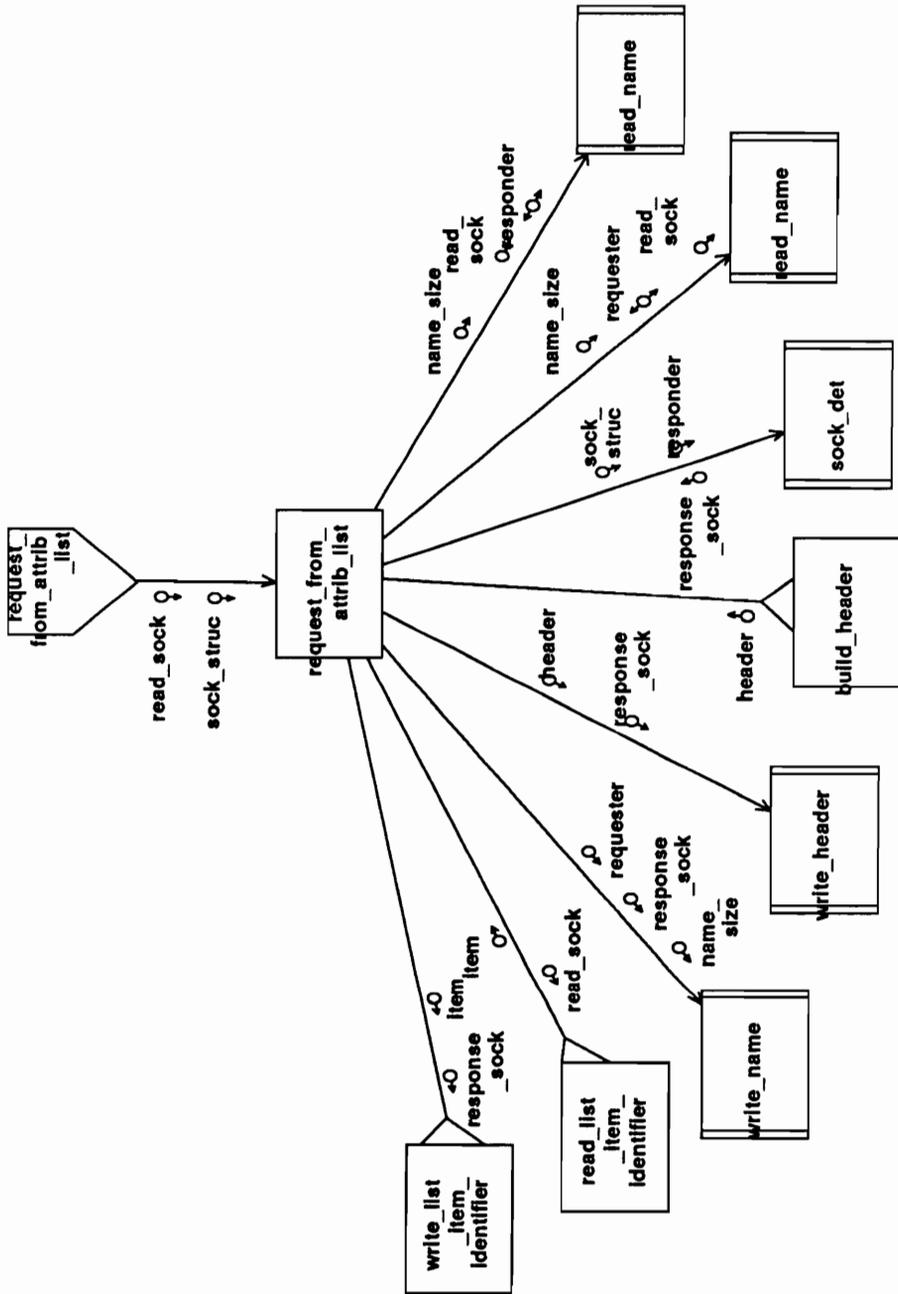


request_data;3
 No title

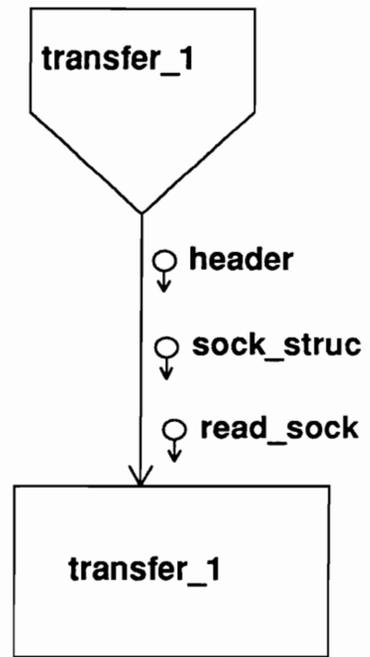


request_attr_list;2
 No title

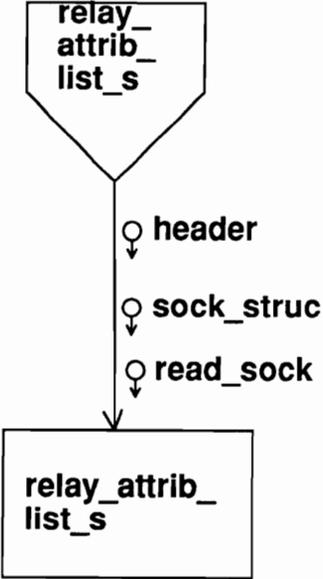




transfer_1;2
No title



relay_attrib_list_s;1
No title



NAME: serv;4

TITLE: IS main module

PARAMETERS:

LOCALS:

BODY:

```
/* =====  
Source Code Filename: serv.c  
Special Considerations: NONE  
Purpose: main program of the integraion server.  
Establishes listening socket to accept connections  
from integration clients  
Belongs to integration server  
===== */  
  
/  
*=====*/  
/*  
serv.c  
Function: starts the server portion of the integrated system  
Variables: none at the moment  
Coded by: Michele Grieshaber  
Date : 06/10/91  
*/  
/  
*=====*/  
#include <stdio.h>  
#include "../mysock2.h"  
#define Port 2000  
#define filename "exchange_buds"  
  
/* —— supporting subroutines —— */  
void sock_ear();  
void set_sel();  
void init_xchg();  
/* —— end supporting subroutines —— */  
  
main ()  
{  
SOCK_INFO *sock_struc; /* structure containing socket data */  
  
/* — zero the sock structure — */  
bzero((char *)&sock_struc, sizeof(sock_struc));  
  
/* — set the num_socks to zero — */  
sock_struc = (SOCK_INFO *)malloc(sizeof(SOCK_INFO));  
sock_struc->num_socks = 0;  
  
/* — initialize the exchange data structure — */  
init_xchg(filename);
```

```

/* — create socket on which to listen — */
sock_ear(Port, sock_struct);

/* — set the select on — */
set_sel(sock_struct);

} /* end main */

```

NAME: init_xchg;6

TITLE: IS init_xchg

PARAMETERS:

filename : data_in
xchg_struct: data_out
num_xchgs : data_out

LOCALS:

in_file
charac

BODY:

```

/* =====
Source Code Filename: init_xchg.c
Special Considerations: NONE
Purpose:
To initialize the xchg_struct which contains a
list of all relations which exist in the integrated system.
This structure is checked against the structure containing
the client names of the connected applications to determine
which will receive the name of the newest client in the
system for use his list of clients from whom data
can be requested.
Belongs to integration server
===== */

```

```

#include <stdio.h>
#include "../mysock2.h"

```

```

XCHG_STRUCT *xchg_struct;
int num_xchgs;

```

```

void init_xchg(char *filename)
{
FILE *in_file;
int i, j;
char charac[1];

```

```

if((in_file = fopen(filename, "r")) == (FILE *) NULL)
{
    printf("open_file: could not find file %s\n", filename);
} else {
    fscanf(in_file, "NUMBER OF EXCHANGES IN FILE = %d\n", &num_xchgs);
    xchg_struct = (XCHG_STRUCT *) malloc ( sizeof(XCHG_STRUCT) * num_xchgs);
    for (i = 0; i < num_xchgs; i++)
    {
        fscanf(in_file, "\nSENDER = %[^/] %c\n", xchg_struct[i].sender, charac);
        fscanf(in_file, "RECEIVER = %[^/] %c", xchg_struct[i].receiver, charac);
    } /* — end for i to num_xchgs — */
} /* — end if fopen — */

return;
} /* — end init_xchg — */

```

NAME: sock_ear;4

TITLE: No title

PARAMETERS:

Port : data_in
sock_struc : data_in

LOCALS:

Sock
server
one
name

BODY:

/* =====

Source Code Filename: sock_ear.c

Special Considerations: NONE

Purpose:

To create a socket to listen for connections
from clients in the integrated system.

Belongs to integration server

===== */

/

=====/

/*

sock_ear.c

Function: opens a socket on which to listen for incoming connections
from clients wishing to join the integrated system.

Variables: Port - well known location of the server
sock_struc - structure containing socket information

Coded by: Michele Grieshaber

Date : 06/10/91

*/

/

=====/

#define _BSD

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <sys/ioctl.h>

#include <errno.h>

#include "../mysock2.h"

/* — supporting routines — */

void sock_l();

SOCK_INFO *cl_list();

/* — end supporting routines — */

sock_ear(int Port, SOCK_INFO *sock_struc)

{

int Sock; /* socket on which listening occurs */

struct sockaddr_in server; /* server internet information */

static int one = 1; /* set as a constant */

char name[] = "listening socket"; /* for the client list in sock_struc */

/* — open socket to listen on and use a stream connection — */

Sock = socket(AF_INET, SOCK_STREAM, 0);

if (Sock < 0)

{

 perror("server:socket");

 exit(-3);

}

/* — clear the server structure — */

bzero((char *)&server, sizeof(server));

/* — initialize the server structure — */

server.sin_family = AF_INET;

server.sin_port = Port;

/* — set the socket so that it is reuseable — */

if (setsockopt(Sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one)) != 0)

{

 perror("setsockopt");

}

/* — bind the Sock to the server — */

if (bind(Sock, &server, sizeof(server)) < 0)

{

```

    perror("server:bind");
    exit(-3);
}

/* — add the socket to the sock_list — */
sock_l(Sock, sock_struct);

/* — add the server to the client list — */
sock_struct = cl_list(sock_struct, name, sizeof(name));

return;
} /* end sock_eat */

```

```

*****

```

```

NAME: sock_l;6

```

```

TITLE: IS sock_l

```

```

PARAMETERS:

```

```

listening_sock : data_in

```

```

sock_struct : data_in

```

```

LOCALS:

```

```

BODY:

```

```

/* ===== */

```

```

Source Code Filename: sock_l.c

```

```

Special Considerations: NONE

```

```

Purpose:

```

```

To add a socket to the portion of the socket
information structure, sock_struct, which contains
socket descriptor information (sock_struct.sock_list).

```

```

Belongs to integration server

```

```

===== */

```

```

/

```

```

*===== */

```

```

/*

```

```

sock_l.c

```

```

Function: adds the most recently connected socket to the socket
          listing inside the SOCK_INFO structure (sock_struct).

```

```

Variables:  new_sock      - socket descriptor of connected sock
            sock_struct   - structure into which new_sock goes

```

```

Coded by: Michele Grieshaber

```

```

Date : 06/01/91

```

```

*/

```

```

/

```

```

*===== */

```

```

#include <stdio.h>

```

```

#include "../mysock2.h"

```

```

sock_l(int new_sock, SOCK_INFO *sock_struct)
{
/* — add new socket to list — */
sock_struct->num_socks += 1;

/* — add the socket to the socket array — */
sock_struct->sock_list[sock_struct->num_socks - 1] = new_sock;

return;
} /* — end sock_l.c — */

```

NAME: cl_list;7

TITLE: IS cl_list

PARAMETERS:

- sock_struct : data_out
- sock_struct : data_in
- socket_name : data_in
- name_size : data_in

LOCALS:

size

BODY:

```

/* =====
Source Code Filename: cl_list.c
Special Considerations: NONE
Purpose:
To add an application name to the portion of
the socket information structure, sock_struct, which
contains client name (sock_struct.client_name).
This enables the server to associate a client name
with a corresponding socket descriptor in the
sock_struct. This allows cross referencing to occur,
meaning if the client name is known, so is the socket
descriptor, and vice-versa.
Belongs to integration server
===== */
/
*=====*/
/*

```

cl_list.c
Function: places the name returned by the client in the client
list portion of the SOCK_INFO structure (sock_struct)
Variables: sock_struct - structure containing socket info

name - name of the connected client
size_of_name - size of the client's name

Coded by: Michele Grieshaber

Date : 06/10/91

*/

/

=====*/

```
#define _BSD
#include <stdio.h>
#include <errno.h>
#include "./mysock2.h"
```

```
SOCK_INFO *cl_list(SOCK_INFO *sock_struct, char name[], int size_of_name)
```

```
{
static int size = 20; /* size of character array */
```

```
/* — add the name of the current socket to the client_list — */
```

```
/* — clear out the array entry — */
```

```
bzero((char *)sock_struct->client_list[sock_struct->num_socks-1], size);
sprintf(sock_struct->client_list[sock_struct->num_socks-1],
        "%s", name);
```

```
return(sock_struct);
```

```
} /* — end cl_list.c — */
```

NAME: set_sel;6

TITLE: No title

PARAMETERS:

sock_struct : data_in

LOCALS: read_mask

to

i

rc

lsock

errno

BODY:

/* =====

Source Code Filename: set_sel.c

Special Considerations: NONE

Purpose:

To set the read mask to contain all known sockets in the integrated system, and then to check for incoming signals on these sockets. If no signal occurs the process repeats itself. When a signal does register its socket

is determined in eval_sel.c.
Belongs to integration server

```
===== */
/
*===== */
/*
set_sel.c
Function: sets the select mode on for the server to screen incoming
          connections
Variables: sock_struct   - structure containing socket info
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
*===== */
#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include "../mysock2.h"

/* — supporting routines — */
void is_eval_sel();
fd_set set_mask();
/* — end supporting routines — */

void set_sel(SOCK_INFO *sock_struct)
{
fd_set read_mask;           /* mask which filters sockets for reading */
struct timeval to;         /* time structure for select timeout */
int i, rc;                 /* rc is the return code variable */
int lsock;                 /* product of a socket sort — largest sock */
extern int errno;         /* error number for debug purposes */

listen(sock_struct->sock_list[0], 5); /* set the listening sock to listen */

do
{
/* — compare the mask against all available sockets — */
/* — also keep track of the largest socket value for later use — */
/* — but to do that, set sock initially to zero — */
lsock = 0;
read_mask = set_mask(&lsock, sock_struct);
/* — set the timeout values for the select — */
bzero((char *)&to, sizeof(to));
```

```

to.tv_sec = 5;

/* — hang out in the select — */
rc = select(lsock+1, &read_mask, (fd_set *)0, (fd_set *)0, &to);
if(rc < 0)
{
    perror("select");
    continue;
} else if (rc > 0) {
    /* — evaluate the response to select if any — */
    is_eval_sel(sock_struct, read_mask);
} /* — end if rc — */

} while(TRUE);

return;
} /* — end set_sel — */

```

NAME: is_eval_sel;5

TITLE: IS is_eval_sel

PARAMETERS:

read_mask : data_in

sock_struct : data+control_in

LOCALS:

BODY:

/* ===== */

Source Code Filename: is_eval_sel

Special Considerations: NONE

Purpose:

Once an incoming signal has been detected,
this function determines on which socket it occurred
and either accepts a new connection (listening socket)
or sends it on to have the HEADER read by another
routine (any other socket besides the listening sock).

Belongs to integration server

===== */

/

=====/

/*

is_eval_sel.c

Function: evaluates the value of the read mask returned from the
select call in set_sel.

If the signal comes in on the listening socket, the
client is requesting to be accepted for connection by

the server.

If the signal comes on a socket that has already been established (accepted), the header is read by `cl_rdmmsg` and appropriate action is taken.

Variables: `sock_struct` - structure containing socket info
`read_mask` - indicates which sockets have info on them.

Coded by: Michele Grieshaber

Date : 06/10/91

*/

/

-----/

```
#define _BSD
```

```
#define TRUE 1
```

```
#define FALSE 0
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
#include <netdb.h>
```

```
#include <sys/ioctl.h>
```

```
#include <fcntl.h>
```

```
#include <sys/file.h>
```

```
#include <signal.h>
```

```
#include <sys/select.h>
```

```
#include <errno.h>
```

```
#include "../mysock2.h"
```

```
/* — supporting routines — */
```

```
void rd_msg();
```

```
void ask_msg();
```

```
void new_sock_info();
```

```
/* — end supporting routines — */
```

```
eval_sel(SOCK_INFO *sock_struct, fd_set read_mask)
```

```
{
```

```
int new_sock;          /* new socket accepted by the server */
```

```
int i;                /* just your ordinary everyday integer */
```

```
struct sockaddr_in sin; /* structure containing client ip stuff */
```

```
int length = sizeof(sin); /* length of above structure */
```

```
/* — check to see if the read_mask matches any of the available sockets — */
```

```
if(FD_ISSET(sock_struct->sock_list[0], &read_mask))
```

```
{
```

```
    /* — accept the new connection — */
```

```
    if((new_sock = accept(sock_struct->sock_list[0], &sin, &length)) < 0)
```

```
    {
```

```
        perror("Server:accept");
```

```
        exit(-3);
```

```
    }
```

```
    /* — add new socket to the sock_list — */
```

```

    /* — send a message to the newly connected client to get his name
for incorporation into the client list contained in the server — */
    new_sock_info(new_sock, sock_struct);

} else {
    /* — check the other connected sockets one at a time for info — */
    for(i = 1; i < sock_struct->num_socks; i++)
    {
        if (FD_ISSET(sock_struct->sock_list[i], &read_mask))
        {
            /* — read message on socket — */
            rd_msg(sock_struct->sock_list[i], sock_struct);
        } /* — end if — */
    } /* — end for — */
} /* — end if — */
return;
} /* — end is_eval_sel.c — */

```

NAME: new_sock_info;3

TITLE: IS new_sock_info

PARAMETERS:

new_sock : data_in
sock_struct : data_in

LOCALS:

BODY:

```

/* =====
Source Code Filename: new_sock_info.c
Special Considerations: NONE
Purpose:
To request the application name from the
application who has just recently connected to
the server.
===== */

```

```

#include <stdio.h>
#include “./mysock2.h”

```

```

void sock_l();
void ask_msg();

```

```

void new_sock_info(int new_sock, SOCK_INFO *sock_struct)
{
    /* — put new sock into socket structure — */
    sock_l(new_sock, sock_struct);
}

```

```

/* — request client name from new client as an identifier — */
ask_msg(sock_struct);

return;
}/* --- end new_sock_info --- */

```

```

*****

```

```

NAME: ask_msg;4

```

```

TITLE: IS ask_msg

```

```

PARAMETERS:

```

```

new_sock : data_in

```

```

sock_struct : data_in

```

```

LOCALS:

```

```

header

```

```

w_sock

```

```

BODY:

```

```

/* ===== */

```

```

Source Code Filename: ask_msg.c

```

```

Special Considerations: NONE

```

```

Purpose:

```

```

To request the name of the application

```

```

who has just connected to the server.

```

```

Belongs to integration server

```

```

===== */

```

```

/

```

```

*===== */

```

```

/*

```

```

ask_msg.c

```

```

Function: to ask the newly connected clients for information on

```

```

                  themselves that can be placed in a client list located

```

```

                  in the SOCK_INFO structure for later use .

```

```

Variables:      sock_struct      - structure(SOCK_INFO) with sock info

```

```

Coded by: Michele Grieshaber

```

```

Date : 06/10/91

```

```

*/

```

```

/

```

```

*===== */

```

```

#define _BSD

```

```

#define TRUE 1

```

```

#define FALSE 0

```

```

#include <stdio.h>

```

```

#include <sys/types.h>

```

```

#include <sys/socketvar.h>

```

```

#include <sys/socket.h>

```

```

#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"

/* — external function calls — */
void write_header();
/* — end external functions — */

ask_msg(SOCK_INFO *sock_struct)
{
HEADER header;          /* contains size and major and minor opcode info */
int w_sock;             /* socket to which message is sent */

header.size_in_bytes = 0;
header.maj_opcode = 0;
header.min_opcode = 0;

/* — send this info to the socket correspondinf to ACSYNT — */
w_sock = sock_struct->sock_list[sock_struct->num_socks - 1];
write_header(w_sock, header);

return;
} /* — end ask_msg.c — */

```

NAME: rd_msg;4

TITLE: IS rd_msg

PARAMETERS:

read_sock : data_in
sock_struct : data_in

LOCALS:

nval
header

BODY:

```

/* =====
Source Code Filename: rd_msg.c
Special Considerations: NONE
Purpose:
If the signal is of a normal type, to read
a headers worth of data from the socket, and if
the signal is a disconnect, to close the socket
which corresponds to the disconnecting client.
Belongs to integration server
===== */

```

```

/
*=====*/
/*
rd_msg.c
Function: reads the header from the information coming in on a
          socket.
          Header info then sent to a routine which does a switch
          on the major and minor opcodes contained in the header.
Variables:  read_sock      - socket on which info is waiting
            sock_struct    - structure containing socket info
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
*=====*/
#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"

/* — supporting routines — */
void sw_op();
void close_sock();
/* — end supporting routines — */

rd_msg(int read_sock, SOCK_INFO *sock_struct)
{
int nval;           /* return code from read */
HEADER header;     /* header read from the socket. Contains info */
                  /* such as size of info on socket, major opcode */
                  /* and minor opcode _____ */

/* — read the header from the information sitting on the socket — */
nval = read(read_sock, &header, sizeof(HEADER));
if(nval == -1)
{
perror("rd_msg: read");
exit(1);
} else if(nval == 0) {
/* — go to routine to close connection and take socket out of list - */
close_sock(read_sock, sock_struct);
} else {
/* — send to sw-op to determine action associated with opcode — */
sw_op(header, sock_struct, read_sock);
} /* — end if — */

```

```
return;
} /* — end rd_msg.c — */
```

NAME: close_sock;4

TITLE: IS close_sock

PARAMETERS:

read_sock : data_in
sock_struc : data_in

LOCALS:

ij
action

BODY:

```
/* =====
Source Code Filename: close_sock.c
Special Considerations: NONE
Purpose:
To delete a closed socket from the array
of sockets in the socket information socket.
Belongs to integration server
===== */
```

```
/* ===== */
/* close_sock.c
Function: deletes a socket from the socket list when a client is
closed.
Arguments: int dead_sock — socket that has been closed
SOCK_INFO *sock_struc — structure containing number of
sockets and the socket list
Coded by: Michele Grieshaber
Date : 06/05/91
*/
/* ===== */
```

```
#include <stdio.h>
#include "../mysock2.h"
#define ADD 1
#define DELETE 0
```

```
void widget_update();

close_sock(int dead_sock, SOCK_INFO *sock_struc)
{
int ij;
int action;
```

```

/* — loop thru socket list to find entry which matches dead socket — */
for (i = 0 ; i < sock_struct->num_socks; i++)
{
  if(sock_struct->sock_list[i] == dead_sock)
  {
    /* — before deleting it from the list, send delete msg to widget—*/
    action = DELETE;
    widget_update(action, sock_struct->client_list[i], sock_struct);
    for(j = i; j < (sock_struct->num_socks - 1); j++)
    {
      sock_struct->sock_list[j] = sock_struct->sock_list[j+1];
    } /* — end for j — */
    i = sock_struct->num_socks;
    sock_struct->num_socks -= 1;
  } /* — end if dead_sock — */
} /* — end for i — */

return;
} /* — end close_sock.c — */

```

NAME: widget_update;5

TITLE: IS widget_update

PARAMETERS:

action : data_in
name : data_in
sock_struct : data_in

LOCALS:

BODY:

```

/* =====
Source Code Filename: widget_update
Special Considerations: NONE
Purpose:
To determine which clients need to be
informed that a client in their selection list
has disconnected from the integrated system.
Belongs to integration server
===== */

```

```

#define _BSD
#define TRUE 1
#define FALSE 0
#define size_of_name 50
#define ADD 1
#define DELETE 0
#include <stdio.h>

```

```

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"
#define size_of_name 50

extern XCHG_STRUCT *xchg_struct;
extern int num_xchgs;
void write_header();
void write_name();

void widget_update( int action, char *name, SOCK_INFO *sock_struct)
{
int i,j;
int n = 0;
HEADER header;

header.size_in_bytes = action;
header.maj_opcode = 2;
header.min_opcode = 0;

/* — match the new name against senders in the xchg_struct — */
for (i = 0; i < num_xchgs; i++)
{
if(strcmp(xchg_struct[i].sender, name) == 0)
{
for( j = 1; j < sock_struct->num_socks; j++)
{
if(strcmp(xchg_struct[i].receiver, sock_struct->client_list[j])==0)
{
/* — send the new name to the client interested — */
write_header(sock_struct->sock_list[j], header);
write_name(sock_struct->sock_list[j], name, size_of_name);

} /* — end if strcmp receiver — */
} /* — end for j — */
} /* — end strcmp sender — */
} /* — end for i — */

return;
} /* — end widget_update.c — */

```

```
*****  
NAME: resolve_header;6
```

```
TITLE: IS specific resolve_header
```

```
PARAMETERS:
```

```
header : data_in  
read_sock : data_in  
sock_struct : data_in
```

```
LOCALS:
```

```
BODY:
```

```
/* =====  
Source Code Filename: resolve_header.c  
Special Considerations: NONE  
Purpose:  
Based on the major opcode portion of the  
header (header.maj_opcode) and then on the minor  
opcode portion of the header (header.min_opcode),  
this module will determine how to evaluate each  
message received by a socket of the server.  
Belongs to integration server  
===== *  
/  
*=====*/
```

```
/*  
resolve_header.c  
Function: based on the major and minor opcodes contained in the  
          header structure passed in from the rd_msg routine,  
          this routine (using switch statements) will determine  
          the appropriate action to take  
Variables: header - contains size and maj and minor opcodes  
          sock_struct - structure containing socket info  
          read_sock - socket on which information resides
```

```
Coded by: Michele Grieshaber
```

```
Date : 06/10/91
```

```
*/  
/  
*=====*/
```

```
#define _BSD  
#define TRUE 1  
#define FALSE 0  
#define size_of_name 50  
#define ADD 1  
#define DELETE 0  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <sys/socketvar.h>  
#include <sys/uio.h>  
#include <errno.h>
```

```

#include "./mysock2.h"
/* — supporting routines — */
SOCK_INFO *put_cl();
void det_list();
void request_data();
void acsynt_to_bspline();
char *read_name();
void write_name();
void write_header();
void request_attrib_list();
void relay_attrib_list_s();
void request_from_attrib_list();
/* — end supporting routines — */

resolve_header(HEADER header, SOCK_INFO *sock_struct, int read_sock)
{
/* — begin major opcode switch — */
switch(header.maj_opcode)
{
case 0:
/* — begin minor opcode switch for major case 0 — */
switch(header.min_opcode)
{
case 0:
/* — read client name and send to client list — */
sock_struct = put_cl(read_sock, header, sock_struct);
break;

case 1:
/* — gives list of exchange requests to client — */
/* — determine the client who requested info — */
det_list(read_sock, sock_struct);
break;

case 2:
break;

default:
printf("resolve_header: not a valid minor opcode\n");
break;
} /* — end switch(min_opcode) — */
/* — end minor opcode switch for major case 0 — */
break;

case 1:
switch(header.min_opcode)
{
case 0:
break;

case 1:

```

```

    request_data (read_sock, header, sock_struct);
break;

case 3:
    request_attrib_list(read_sock, sock_struct);
break;

case 4:
    request_from_attrib_list(read_sock, sock_struct);
break;

default
    printf("resolve_header: not a valid minor opcode for major = 1\n");
break;
} /* — end switch minor for major = 1 — */
break;

case 2:
switch(header.min_opcode)
{
    case 0:
break;

    case 1:
        acsynt_to_bspline(read_sock, header, sock_struct);
break;

    case 3:
        relay_attrib_list_s(read_sock, sock_struct, header.size_in_bytes);
break;

    default
        printf("resolve_header: not a valid minor opcode for major = 2\n");
break;
} /* — end switch min_opcode for case major = 2 — */
break;

default:
    printf("resolve_header: not a valid major opcode \n");
break;

}/* — end switch(maj_opcode) — */
/* — end major opcode switch ————— */

return;
} /* — end resolve_header.c — */

```

NAME: put_cl;6

TITLE: IS put_cl

PARAMETERS:

header : data_in
read_sock : data_in
sock_struct : data_inout

LOCALS:

name
action

BODY:

```
/* =====  
Source Code Filename: put_cl.c  
Special Considerations: NONE  
Purpose:  
To read in the name of an application who  
is responding to the request generated after  
its connection has been accepted by the server.  
The name is sent to all clients in the system  
to whom it can supply data.  
Belongs to integration server  
===== */
```

```
#define _BSD  
#define ADD 1  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <sys/socketvar.h>  
#include <sys/uio.h>  
#include <errno.h>  
#include "../mysock2.h"
```

```
/* — supporting routines — */  
SOCK_INFO *cl_list();  
char *read_name();  
void widget_update();
```

```
SOCK_INFO *put_cl(int read_sock, HEADER header, SOCK_INFO *sock_struct)  
{  
char *name;  
int action;  
  
if(header.size_in_bytes > 50)  
{  
return;  
}  
}
```

```

/* — read client name from the msg on socket from client — */
name = read_name(read_sock, name, header.size_in_bytes);

/* — add client name to list kept by server — */
sock_struct = cl_list(sock_struct, name, header.size_in_bytes);

/* — send name to clients which can use it in widget — */
action = ADD;
widget_update(action, name, sock_struct);
free(name);

return(sock_struct);
} /* — end put_cl.c — */

```

NAME: det_list;5

TITLE: IS det_list

PARAMETERS:

read_sock : data_in
sock_struct : data_in

LOCALS:

BODY:

```

/* =====
Source Code Filename: det_list.c
Special Considerations: NONE
Purpose:
To compile a list of exchange clients to
send to a client requesting the list. He needs
to know from whom he can request data in the
integrated system.
Belongs to integration server
===== */

```

```

#include <stdio.h>
#include "../mysock2.h"

```

```

/* — external function calls — */
void send_cl_list();
/* — end external functions — */

```

```

void det_list( int read_sock, SOCK_INFO *sock_struct)
{
int i;

```

```

/* — gives list of exchange requests to client — */
/* — determine the client who requested info — */

```

```

for (i = 1; i < sock_struct->num_socks; i++)
{
    if(sock_struct->sock_list[i] == read_sock)
    {
        send_cl_list(read_sock, sock_struct->client_list[i],
            sock_struct);
        i = sock_struct->num_socks;
    } /* — end if read_sock — */
} /* — end for num_socks — */

return;
} /* — end det_list.c — */

```

NAME: send_cl_list;5

TITLE: IS send_cl_list

PARAMETERS:

read_sock : data_in
name : data_in
sock_struct : data_in

LOCALS:

BODY:

```

/* =====
Source Code Filename: send_cl_list
Special Considerations: NONE
Purpose:
Used by the server to compile a list
of exchange client names and send them to
the requesting client.
Belongs to integration server
===== */

```

```

#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include “./mysock2.h”
#include <string.h>

```

```

extern XCHG_STRUCT *xchg_struct;
extern int num_xchgs;

```

```

void write_header();
void send_cl_list(int sock, char client_name[], SOCK_INFO *sock_struct)
{
int i,j;
int n = 0;
HEADER header;
struct list {
                char list_item[80];
                struct list *p1;
} *xchg_list;

/* — match up the client name with the xchg list — */
for (i = 0; i < num_xchgs; i++)
{
    if(strcmp(xchg_struct[i].receiver ,client_name) == 0)
    {
        /* — need to check the match against connected clients — */
        for (j = 1; j < sock_struct->num_socks; j++)
        {
            if(strcmp(xchg_struct[i].sender, sock_struct->client_list[j]) == 0)
            {
                if ( n == 0)
                {
                    xchg_list = (struct list *)malloc(sizeof(struct list));
                } else {
                    xchg_list->p1 = (struct list *) malloc(sizeof(struct list));
                    xchg_list = xchg_list->p1;
                }
                bzero((char *)xchg_list->list_item, 80);
                sprintf(xchg_list->list_item, "%s", xchg_struct[i].sender);
                xchg_list->p1 = NULL;
                n += 1;
            } /* — end if strcmp — */
        } /* — end for — */
    } /* — end if strcmp — */
} /* — end for num_xchgs — */

/* — build a message to send to the client containing info — */
header.size_in_bytes = n;          /* — indicates the number of infos — */
header.maj_opcode = 0;
header.min_opcode = 2;

write_header(sock, header);

for ( i = 0; i < n; i++)
{
    if(write(sock, xchg_list->list_item, 50) < 0)
    {
        perror("sendl: write header ");
        exit(1);
    }
}

```

```

    xchg_list = xchg_list->p1;
}

return;
} /* — end send_cl_list — */

```

NAME: request_data;4

TITLE: IS request_data

PARAMETERS:

```

read_sock : data_in
header : data_in
sock_struct : data_in

```

LOCALS:

```

responder
requester
respond_sock

```

BODY:

```

/* =====
Source Code Filename: request_data.c
Special Considerations: NONE
Purpose:
To relay the request for buffer data from
the requesting client to the responding client.
Belongs to integration server
===== */

```

```

#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include “./mysock2.h”

```

/* — supporting routines — */

```

char *read_name();
void write_name();
void write_header();
int sock_det();

```

```

void request_data(int read_sock, HEADER header, SOCK_INFO *sock_struct)
{

```

```

char *responder, *requester;
int respond_sock;

/* — read info from client on requested data exchange — */
responder = read_name (read_sock, responder, header.size_in_bytes);

/* — read the name of the application requesting data — */
requester = read_name(read_sock, requester, header.size_in_bytes);

/* — determine the socket upon which requested comms — */
respond_sock = sock_det(responder, sock_struct);

/* — request info be sent from requested client — */
header.size_in_bytes = 0;
header.maj_opcode = 3;
header.min_opcode = 1;
write_header(respond_sock, header);

/* — send it the requester's name — */
write_name(request_sock, requester, size_of_name);
free(requester);
free(responder);

return;
} /* — end request_data.c — */

```

```

*****
NAME: sock_det;4

TITLE: IS sock_det

PARAMETERS:
responder : data_in
respond_sock : data_out

LOCALS:
i
next_sock

BODY:
/* =====
Source Code Filename: sock_det.c
Special Considerations: NONE
Purpose:
To determine a socket identifier based
on the connected application's name.
Belongs to integration server
===== */

```

```

#include <stdio.h>
#include "../mysock2.h"
#define TRUE 1
#define FALSE 0

int sock_det(char *name, SOCK_INFO *sock_struct)
{
int i;
int next_sock;

/* — determine the socket number that corresponds to the name — */
for (i = 1; i < sock_struct->num_socks; i++)
{
if(strcmp(sock_struct->client_list[i], name) == 0)
{
next_sock = sock_struct->sock_list[i];
i = sock_struct->num_socks;
}
}

return(next_sock);
} /* — end sock_det.c — */

```

NAME: request_attrib_list;4

TITLE: IS request_attrib_list

PARAMETERS:
read_sock : data_in
sock_struct : data_in

LOCALS:
header
response_sock
responder
requester

BODY:
/* =====
Source Code Filename: request_attrib_list.c
Special Considerations: NONE
Purpose:
To request the attribute list from the
responding client.
Belongs to integration server
===== */

#define _BSD

```

#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include "../mysock2.h"

/* — function declarations — */
void write_header();
void write_name();
char *read_name();
int sock_det();

void request_attrib_list( int read_sock, SOCK_INFO sock_struct)
{
HEADER header;
int response_sock;
char *responder;
char *requester;

/* — for now get the component list from the client and send it on — */
header.size_in_bytes = 0;
header.maj_opcode = 3;
header.min_opcode = 3;

/* — read the requester's name — */
requester = read_name( read_sock, requester, size_of_name);

/* — read next client name from requester — */
responder = read_name(read_sock, responder, size_of_name);
response_sock = sock_det(responder, sock_struct);

write_header(response_sock, header);
write_name(response_sock, requester, size_of_name);
write_name(response_sock, responder, size_of_name);

free(responder);
free(requester);

return;
} /* — end request_attrib_list.c — */

```

NAME: request_from_attrib_list;6

TITLE: IS request_from_attrib_list

PARAMETERS:

read_sock : data_in

sock_struct : data_in

LOCALS:

responder

requester

respond_sock

list_num

header

BODY:

```
/* =====  
Source Code Filename: request_from_attrib_list.c  
Special Considerations: NONE  
Purpose:  
To request a specific list item from the  
attribute list previously supplied by the responding  
client.  
Belongs to integration server  
===== */
```

```
#define _BSD  
#define size_of_name 50  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socketvar.h>  
#include <sys/socket.h>  
#include <sys/uio.h>  
#include <netinet/in.h>  
#include <netdb.h>  
#include "../mysock2.h"  
  
/* — external functions — */  
char *read_name();  
void write_header();  
void write_name();  
int sock_det();  
  
void request_from_attrib_list(int read_sock, SOCK_INFO sock_struct)  
{  
char *responder, *requester;  
int respond_sock;  
int list_num;  
HEADER header;  
  
/* — read source name — */
```

```

responder = read_name(read_sock, responder, size_of_name);
requester = read_name(read_sock, requester, size_of_name);

/* — determine the socket for source — */
respond_sock = sock_det(respond_name, sock_struct);

/* — define and send header — */
header.size_in_bytes = 0;
header.maj_opcode = 3;
header.min_opcode = 4;
write_header(respond_sock, header);

/* — write the requester id — */
write_name(respond_sock, requester, size_of_name);

/* — read the component number identifier and send it — */
if (read(read_sock, &list_num, sizeof(int)) < 0)
{
    perror("reading list_num");
    exit(1);
}
if (write(respond_sock, &list_num, sizeof(int)) < 0)
{
    perror("write list_num");
    exit(1);
}

return;
} /* — end request_from_attrib_list.c — */

```

NAME: transfer_1;6

TITLE: IS specific transfer_1

PARAMETERS:

header : data_in
sock_struct : data_in
read_soc : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This is just one example of where the transformation functions need to be in the

server. This location corresponds to the B-Spline Toolkit/ACSYNT example where the transformation function `acsynt_to_bspline` is found. As an example, that module will be included in this m-spec.

Belongs to integration server

```

===== */
#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "./mysock2.h"

/* — supporting routines — */
int sock_det();
void from_acs_to_b();
char *read_name();
void write_name();
void write_header();

void acsynt_to_bspline(int read_sock, HEADER header, SOCK_INFO *sock_struct)
{
char *name;
int j, request_sock;
int ncomps;

/* — receive the name of the application requesting — */
name = read_name(read_sock, name, size_of_name);

/* — receive data from ACSYNT for xfer to B-SPLINE — */
ncomps = header.size_in_bytes;

/* — set new header, except for header.size_in_bytes which is ncomps — */
header.maj_opcode = 2;
header.min_opcode = 1;
request_sock = sock_det(name, sock_struct);
write_header(request_sock, header);

/* do this for the number of components that exist — */
for (j = 0; j < ncomps; j++)
{
from_acs_to_b(read_sock, request_sock);
} /* — end for j — */
free(name);
return;
} /* — end acsynt_to_bspline.c — */

```

```

=====*/
from_acs_to_b.c

Function: receives information from the ACSYNT module and passes
          it directly to the B_Spline Toolkit
Variables: read_sock - socket on which info is being read
          write_sock - socket which is being written to
Coded by: Michele Grieshaber
Date : 06/10/91
*/
/
=====*/

#define _BSD
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"
/*-----*/

char *read_name();
void write_name();

void from_acs_to_b(int read_sock, int write_sock)
{
char *comp_name;
int i, j, k ;
int comp_number,
color,
rc,
nxsect,
nppxs;
static int size_of_name = 21;
float pt;                /* points describing the component */

/* — read and write comp name — */
comp_name = read_name(read_sock, comp_name, size_of_name);

write_name(write_sock, comp_name, size_of_name);
/* — read and write component number — */
rc = read(read_sock, &comp_number, sizeof(int));
if(rc < 0)
{
perror("from_acs_to_b: read comp_number\n");
exit(1);
}
}

```

```

if(write(write_sock,&comp_number,sizeof(int)) < 0) /* send num of comps*/
{
    perror("fatb: write comp_number ");
    exit(1);
}

/* — read and write component color — */
rc = read(read_sock, &color, sizeof(int));
if(rc < 0)
{
    perror("from_acs_to_b: read color\n");
    exit(1);
}
if(write(write_sock,&color,sizeof(int)) < 0) /* send comp color */
{
    perror("fatb: write color");
    exit(1);
}

/* — read and write number of cross sections — */
rc = read(read_sock, &nxsect, sizeof(int));
if(rc < 0)
{
    perror("from_acs_to_b: read nxsect\n");
    exit(1);
}
if(write(write_sock,&nxsect, sizeof(int)) < 0) /* send x_sec num*/
{
    perror("fatb: write nxsect");
    exit(1);
}

/* — read and write number of points per x section — */
rc = read(read_sock, &nppxs, sizeof(int));
if(rc < 0)
{
    perror("from_acs_to_b: read nppxs\n");
    exit(1);
}
if(write(write_sock,&nppxs, sizeof(int)) < 0) /* send pts/x_sec */
{
    perror("fatb: write nppxs");
    exit(1);
}

/* — read and write point data — */
for ( i = 0; i < nxsect; i++)
{
    for(j = 0; j < nppxs; j++)
    {
        for(k = 0; k < 3; k++)

```

```

{
    rc = read(read_sock, &pt, sizeof(float));
    if(rc < 0)
    {
        perror("from_acs_to_b: read pt");
        exit(1);
    }

    if(write(write_sock, &pt, sizeof(float)) < 0)
    {
        perror("from_acs_to_b: write pt");
        exit(1);
    }
} /* — end for k — */
} /* — end for j — */
} /* — end for i — */
return;
} /* — end from_acs_to_b — */

```

NAME: relay_attrib_list_s;5

TITLE: IS specific relay_attrib_list_s

PARAMETERS:

header.size_in_bytes : data_in

sock_struct : data_in

read_sock : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: to be determined

Special Considerations: NONE

Purpose:

This module is the server version which relays a list of attributes sent from one client, to another.

This module is dependent on the application, and must be written for each client sending an attribute list to other clients in the system. The structure of this module is entirely dependent on the way in which the client sending the list, transmits his data.

As an example, the module relay_attrib_list.c, which is part of the server library related to the B-Spline Toolkit/ACSYNT relationship, is included in this m-spec.

Belongs to integration server

===== */

```

#define _BSD
#define size_of_name 50
#include <stdio.h>
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <netdb.h>
#include “./mysock2.h”

/* — function declarations — */
void write_header();
void write_name();
char *read_name();
int sock_det();

void relay_attrib_list_s(int read_sock, SOCK_INFO *sock_struct, int ncomps)
{
int i;
char *comp_name;
int size_of_compname = 21;
char *requester;
char *responder;
int request_sock;
HEADER header;
int comp_num;

/* — read requesting and responding clients name — */
requester = read_name(read_sock, requester, size_of_name);
responder = read_name(read_sock, responder, size_of_name);

/* — determine the requesting socket number — */
request_sock = sock_det(requester, sock_struct);

header.size_in_bytes = ncomps;
header.maj_opcode = 2;
header.min_opcode = 3;
write_header(request_sock, header);
write_name(request_sock, requester, size_of_name);
write_name(request_sock, responder, size_of_name);

/* — loop thru reads for the number of components — */
for (i = 0; i < ncomps ; i++)
{
if (read (read_sock, &comp_num, sizeof(int)) < 0)
{
perror(“ relay_att_1: reading comp_num”);
exit(1);
}
}
}

```

```
if (write(request_sock, &comp_num, sizeof(int)) < 0)
{
    perror("relay_att_l: writing comp_num");
    exit(1);
}
comp_name = read_name(read_sock, comp_name, size_of_compname);
write_name(request_sock, comp_name, size_of_compname);
} /* — end for ncomps — */

free(requester);
free(responder);
free(comp_name);

return;
} /* — end relay_attrib_list_s.c — */
```

APPENDIX F: UTILITIES, ETC.

This appendix contains miscellaneous files and utilities necessary for complete understanding of the integration system. One important note is on the directories in which each component of the prototype integration system was developed. This information may be necessary when studying the source code of the components. The source code is located in the module specifications found in the each component's appendix. The components and their directories are:

integration server	/u/michele/grim/server
ACSYNT client application	/u/michele/grim/acsynt
ACSYNT GRIM widget (grim2)	/u/michele/grim/grim2
B-Spline client application	/u/michele/grim/apsock
and	/u/michele/grim/execs
B-Spline GRIM widget (grimmy)	/u/michele/grim/grimmy
Utility functions	/u/michele/grim/utility

Included in this appendix are the scripts necessary to run the two prototype clients, a sample relations file which is used by the server to define data exchange possibilities, and a header file called mysock2.h which contains data structures used by all components of the integration system.

```
/*=====*/
```

This is the RACS exec

This is the exec which will start the B-Spline Toolkit Client.

It will start the GRIM widget (grimmy) in the background, sleep a few seconds, then invoke the B-Spline Toolkit using acsnubs as the executable.

```
/*=====*/
```

```
../grimmy/grimmy &  
sleep 3  
./acsnubs
```

```
/*=====*/
```

This is the RACSYNT exec

This is the exec which will start the ACSYNT Client.

It will start the GRIM widget (grim2) in the background, sleep a few seconds, then invoke ACSYNT using acsynt as the executable.

```
/*=====*/
```

```
../grimmy2/grim2 &  
sleep 3  
./acsynt
```

```
/*=====*/  
This file contains typedefs used in the integration client and server.
```

```
CODED BY: Michele Grieshaber  
DATE:    May 28, 1991
```

```
/*=====*/
```

```
typedef struct {  
    int size_in_bytes;  
    int maj_opcode;  
    int min_opcode;  
} HEADER;  
  
typedef struct {  
    int num_socks;  
    int sock_list[20];  
    char client_list[50][50];  
} SOCK_INFO;  
  
typedef struct {  
    char sender[50];  
    char receiver[50];  
} XCHG_STRUCT;  
  
typedef struct list_type {  
    char list_item[50];  
    struct list_type *p1;  
} LIST;
```

NAME: read_name;4

TITLE: Utility read_named

PARAMETERS:

read_sock: data_in

name_size : data_in

name : data_inout

LOCALS: nval

BODY:

```
/* =====
```

Source Code Filename: read_name.c

Special Considerations: NONE

Purpose:

To read a character string (usually a name)
off of a specified socket identifier.

This is a utility function

```
===== */
```

```
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>

char *read_name(int sock, char *name, int name_size)
{
int nval; /* — return code value for read — */

/* — allocate space for the name — */
name = (char *)malloc(name_size);

/* — read name — */
nval = read(sock, name, name_size);
if (nval < 0)
{
perror("read_name: read");
exit(1);
}
return(name);
} /* — end read_name.c — */
```

NAME: write_header;4

TITLE: Utility write_headerd

PARAMETERS:

header : data_in

write_sock : data_in

LOCALS:

BODY:

```
/* =====
```

Source Code Filename: write_header.c

Special Considerations: NONE

Purpose:

To write a header to a specified socket identifier.

This is a utility function used by all components

```
===== */
```

```
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>
#include "../mysock2.h"

void write_header(int sock, HEADER header)
{
if(write(sock, &header, sizeof(HEADER)) < 0)
{
perror("write header: write header");
exit(1);
}
return;
} /* — end write_header.c — */
```

NAME: write_name;4

TITLE: Utility write_name

PARAMETERS:

name : data_in

write_sock : data_in

name_size : data_in

LOCALS:

BODY:

/* =====

Source Code Filename: write_name.c

Special Considerations: NONE

Purpose:

To write a character string (usually a name)
to a specified socket identifier.

This is a utility

===== */

```
#define _BSD
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>
#include <errno.h>

void write_name(int sock, char *name, int name_size)
{
    if(write(sock, name, name_size) < 0)
    {
        perror("write name: write name");
        exit(1);
    }

    return;
} /* — end write_name.c — */
```

VITA

As a child, Michele Grieshaber spent much of her time doing arts and crafts. Some of her greatest accomplishments include a surrealist drawing of a clown done in a Crayola medium, a key chain made from gimp (stringy plastic stuff that melts if left in a hot car), and a set of wind chimes made from driftwood, sea shells, and dental floss. Not many people realize the artistic potential of dental floss. Life was not always easy for a struggling young artist, so when it came time to plan for the future, she decided to train herself for a more practical career. In the fall of 1983, she was accepted into the engineering curriculum of Virginia Polytechnic Institute and State University. During her senior year, she was one of four participants in the first exchange program between Virginia Tech and the Universite de Technologie de Compiegne, FRANCE. A semester after her return to the U.S., she received her Bachelor's degree. She then completed her Masters in 1988, just before leaving for Paris to study at Ecole Centrale des Arts et Manufactures as a Fulbright Fellow. Although tempted to remain in Paris to study under Fifi Van Gogh (third cousin twice removed of Vince), a renowned artist in dental floss sculpture, Michele returned to Virginia Tech to pursue a PhD.