Framework Programmable Platform for the Advanced Software Development Workstation

Preliminary System Design Document

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Richard J. Mayer, Thomas M. Blinn, Dr. Paula S.D. Mayer, Keith A. Ackley, John W. Crump, IV, Richard Henderson and Michael T. Futrell of Knowledge Based Systems, Inc. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

Framework Programmable Platform for the Advanced Software Development Workstation (FPP/ASDW)

Preliminary System Design Document

Produced For:

Software Technology Branch
NASA Johnson Space Center
Houston, TX 77058

Authors:

Dr. Richard J. Mayer
Thomas M. Blinn
Dr. Paula S.D. Mayer
Keith A. Ackley
John W. Crump, IV
Richard Henderson
Michael T. Futrell

Knowledge Based Systems, Inc.
2746 Longmire Drive
College Station, TX 77845-5424
(409) 696-7979

December 13, 1991

Table of Contents

List of Figures

1 Introduction

The Framework Programmable Software Development Platform (FPP) is a project aimed at combining effective tool and data integration mechanisms with a model of the software development process in an intelligent integrated software development environment. Guided by the model, this system development framework will take advantage of an integrated operating environment to automate effectively the management of the software development process so that costly mistakes during the development phase can be eliminated. This Platform is being developed under the Advanced Software Development Workstation (ASDW) Program sponsored by the Software Technology Branch at NASA Johnson Space Center. The ASDW program is conducting research into development of advanced technologies for Computer Aided Software Engineering (CASE).

1.1 FPP Overview

The FPP was conceived in response to difficulties of producing software systems. With the advent of more powerful and more economical computer hardware resources, the complexity of software systems has increased dramatically. As computer systems become more complicated, ensuring that systems are produced in a consistent manner, on time, and within budget, and ensuring that the system built is reliable and maintainable, requires a considerable management effort.

One characteristic of large software systems is the inability of a single person to fully understand the requirements, produce the design, and develop the system. Instead, the system development process must be executed by a team of managers and software engineers. Tasks within the development can occur concurrently, except where certain tasks depend on information produced by others. These interrelationships make the management of the development process very difficult. Regardless of how well a development project may be planned out, without some form of control over the actions of the development team, costly mistakes and setbacks are bound to occur during development. This is particularly true in multi-year projects that suffer from management and technical team leadership turnover.

One promise of Computer Aided Software Engineering (CASE) tools was to assist project managers in monitoring the progress of the development activities and in capturing the experiences of the development team. However, the existing CASE tools fail to cover the entire software development process and tend to concentrate instead on a particular aspect of the development process (i.e., project management, requirements analysis, code development and debugging). The result has usually been to use a piecemeal collection of various CASE tools that addresses only portions of the development process during the development of software systems.

Many of these tools are quite useful within their specified area of the system development process. A persistent problem with these tools, however, has been in trying to use the tools in some organized fashion to fully automate the system development process. Incompatible data formats along with the misuse of tools make interaction among these different tools very difficult. As a result, development of CASE environments that effectively automate the software engineering process are nonexistent.

The recognition of these difficulties has spurred the development of the FPP. The focus of the FPP is the management, control, and integration of the software system development process. The major goals in this definition of the FPP have been to provide:

1. a realistic integration strategy that supports function and data integration of a suite of tools (distributed and covering the entire life-cycle);
2. integrated access to and update of life cycle artifact data;
3. control of life cycle activities and data evolution; and
4. a site-specific development process support environment, enforcing the rules and preferred methods of the organization.

The FPP is also expected to provide these capabilities in a distributed, heterogeneous computing environment. Developing a platform that meets these goals should result in (1) a reduction in the time required to produce software systems, (2) an increase in the quality of the resulting software systems, (3) a decrease in the maintenance effort for the resulting software systems, and (4) an increase in the consistency in the development process by which software systems are constructed.

1.2 Scope of This Document

Following extensive work in the definition of the operational concepts ([FPP 90a]) and functional requirements ([FPP 90b]) of the Framework Programmable Platform, recent work on the FPP has focused on the design of the components that make up the FPP. Two previous documents detailed the design of the Integration Mechanism component ([FPP 91a]) and the Framework Processor component ([FPP 91b]). These two components represent the most interesting and complex aspects of the FPP and therefore warranted their own independent design documents.

The focus of this document is on the design of the remaining components of the FPP. These components serve as supporting systems for the Integration Mechanism and the Framework Processor and provide the "glue" that ties the FPP together. More specifically, this document covers the components of the FPP that allow the platform to operate in a distributed, heterogeneous environment and to manage the development and evolution of software system artifacts.

At present, this document represents the final design document to be produced as part of the FPP effort. As such, this document reflects the design of the FPP as it currently exists and as the FPP is to be implemented. As the FPP design has evolved over the last few months, this document will also reflect any changes in the designs of the Integration Mechanism and the Framework Processor.

As the next stage of the FPP development is to actually implement the designs represented in this document and the two previous documents, this document has a slightly different feel than the two previous design documents. The designs of the Integration Mechanism and the Framework Processor presented more of a system design. That is, the documents described more how the two components would be decomposed into subsystems and how they would operate than how they would actually be implemented. In this document, the system designs of the remaining components along with their relationships to the Integration Mechanism and the Framework Processor will be defined. But, more importantly, a detailed description of the software design for each of these components will be provided. In addition, the software designs of the Integration Mechanism and Framework Processor will be presented. Definition of the software design will allow the transition to implementation to occur with minimal effort.

1.3 Document Organization

The design of the FPP is presented in the following sections. The discussion begins in Section 2 with the presentation of the Preliminary Design of the FPP. In this discussion, the architecture of the FPP is presented and the components of the platform are identified. Also in this section, a scenario of how the various components will work together is presented.

In Section 3, the discussion shifts to a detailed discussion of the design of each of the components of the FPP. In this section, the software modules that make up each of the components of the FPP are identified and a discussion of the functionality to be provided by each of the modules is provided.

In Section 4, a discussion of data types and data repositories to be used by the FPP is provided. The data types provide the structures by which the various components of the FPP will be able to communicate. The data repositories represent data storage bases that will maintain information necessary for the proper operation of the FPP.

Finally, Section 5 describes the state of the FPP project and directions that will be pursued in the future. Section 6 presents a list of references and related documents. This section is followed by several appendices that provide syntaxes for languages to be used by various components of the

FPP. Appendix A describes conventions used in the specifications of these syntaxes. Appendix B describes the Data Query Language Syntax. Appendix C describes the Service External Representation Language Syntax. Appendix D describes the Service Request Language Syntax. Appendix E describes the Service Results Language Syntax. Appendix F provides a Network Message Format Description.

2 FPP Preliminary Design

In developing an environment that provides automated support for the management, control, and integration of the software development process, the necessary functionality tended to center around two broad themes: integration and process management. In the FPP Integration Mechanism Design Document ([FPP 91a]), the strategy for integrating CASE tools in a distributed, heterogeneous environment was described. In the FPP Framework Processor Design Document ([FPP 91b]), the means for representing a software development process and using that description to automatically monitor and control that process was described.

In producing the designs for the Integration Mechanism and Framework Processor, however, certain assumptions in the abilities of an operational FPP were made. Among other things, the Integration Mechanism assumed that the FPP would support network message passing between different hosts and The Framework Processor assumed interaction between the Integration Mechanism and the Framework Processor. It is these assumptions that this design document addresses. The discussion begins in this section by first describing the functional architecture of the FPP and then by describing how the various components of the architecture will work together to integrate and manage the software development process.

2.1 FPP Functional Architecture

In this section, a description of the architecture of the FPP and the components that make up that architecture will be provided. As the FPP is expected to operate in a distributed, heterogeneous hardware environment, the architecture of the FPP incorporates a layered approach. This approach allows the more machine or network dependent components to be separated from the components that provide the more generic functionality of the FPP. This design approach was required because of the multiple platforms that will be supported by the FPP. The separation will allow the generic components to be developed independent of a specific machine and will therefore be easily ported between machines. When support for a new machine is necessary or desired, only the system dependent components will have to be reexamined for modification.

It should be noted that the FPP architecture to be presented is a derivative of the Design Knowledge Management System platform architecture that KBSI is currently developing for the Air Force [DKMS 90], [DKMS 91]. The approach in designing the FPP has been to leverage off of the DKMS effort where possible. The most notable similarity between the two systems is the application of the Integration Services approach by the Integration Mechanism. The FPP uses the Integration Services approach as a basis for system integration and then combines the services capability with the framework programmability provided by the Framework Processor to support the management and control of the software development process.

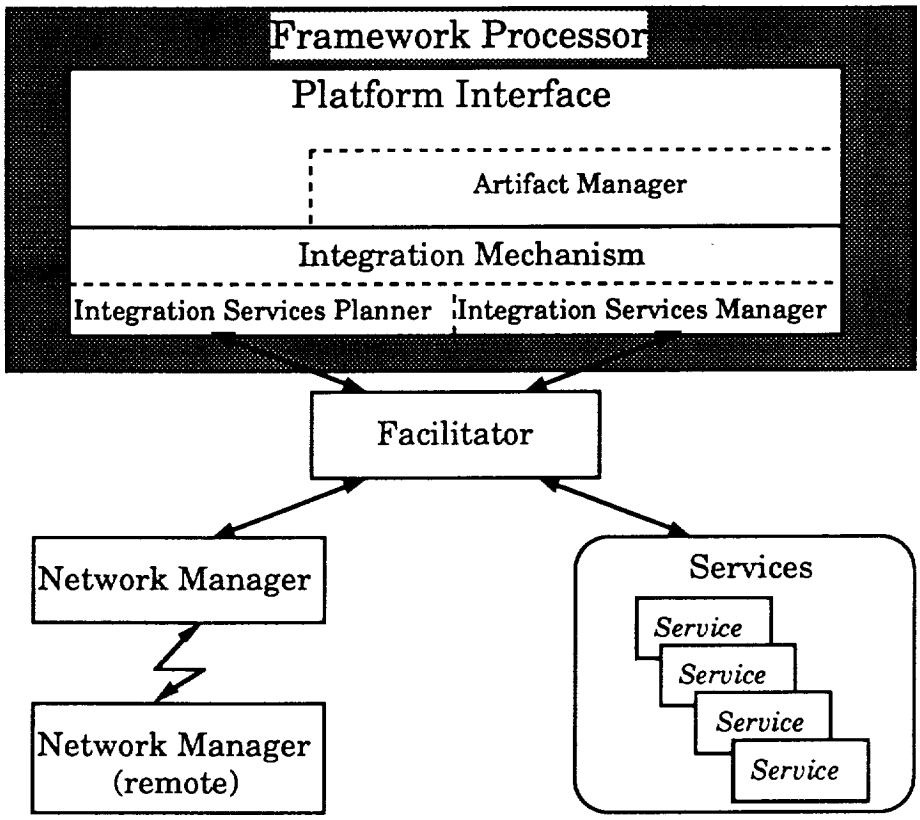

Figure 1. Framework Programmable Platform Architecture

Figure 1 illustrates the architecture of the FPP. The diagram shows the functional architecture where each box in the diagram represents a functional unit and the links between the boxes represent interfaces between functional units. The remainder of this section is dedicated to the discussion of the purpose and design of these units.

2.1.1 Platform Interface

The *Platform Interface* represent the external interface to the FPP. This component serves as a user's direct link to the FPP environment, either through the FPP user interface or through some application accessing the FPP through the specified programming protocols. The Platform Interface accepts user messages (commands). Messages will be interpreted by the Platform Interface and dispatched according to their general category. These categories include 1) Artifact Management, 2) Functional Service Requests, and 3) System Commands. Artifact management messages are dispatched to the Artifact Manager and Functional Service Requests are dispatched to the Integration Services Manager. System Commands are either handled in the Platform Interface itself or dispatched to the Integration Service Manager as appropriate.

It is important to note that the Platform Interface does not represent a program interface. Instead, the Platform Interface simply provides a set of functions and protocols that allow applications access to the functionality provided by the FPP. During the implementation of the FPP, the first application to be built using the functions provided by the FPP will be an FPP user interface. This application will provide a graphical user interface for using the FPP functionality and will access the FPP through the Platform Interface.

2.1.2 Artifact Manager

The *Artifact Manager* is responsible for the management of life cycle design artifacts. In managing life cycle artifacts, the Artifact Manager will provide functionality for registering data artifacts in the repository and for maintaining access control over the artifacts. The Artifact Manager will also include versioning and configuration management functionality and will be used to manage Integration Platform system resource artifacts. In performing these functions, the Artifact Manager will take advantage of the functionality provided by the Integration Services Manager to actually invoke and execute many of the functional services required to support the management of artifacts.

2.1.3 Integration Mechanism

The *Integration Mechanism* is responsible for monitoring and controlling the generation and execution of integration service plans. The idea behind the services approach is based on the view that computer tools and utilities provide services to users. The Integration Mechanism provides the means for defining the services provided by tools and support for the automatic execution of those services. The reader is referred to [FPP 90a] and [FPP 91a] for a more detailed discussion of the integration services concept.

The Integration Mechanism consists mainly of two components: the Integration Services Manager and the Integration Services Planner. The Integration Services Manager is responsible for monitoring and controlling the generation and execution of integration service plans. The Integration Services Manager provides the means for defining the services provided by tools and support for the automatic execution of those services. The Integration Services Planner is responsible for generating the service plans that will be used by the Integration Services Manager. The service request received by the Integration Services Manager may have a service plan stored in the plan repository. If this is the case, the Integration Services Planner simply retrieves this plan and returns it to be executed by the Service Plan Executor. If no plan can be found in the repository, a plan must be generated. This is what the Integration Services Planner does, using the information stored about the available services.

2.1.4 Framework Processor

The *Framework Processor* provides the functionality for interpreting a system development framework and using the information maintained in the framework to monitor and control the system development process. The reader is referred to [FPP 90a] and [FPP 91b] for a more detailed description of framework processing.

In the development of the FPP, the operational approach taken by the Framework Processor has evolved from being an integrated component of the FPP Platform to being a separate, but cooperating, component. This idea is reflected in Figure 1 by showing the Framework Processor as a shaded box that lies behind the Platform Interface and Integration Mechanism. A major reason for this new approach is the desire to separate the functionality provided by the Framework Processor from the functionality of the Integration Platform. Doing this allows the Framework Processor to be platform independent. That is, the Framework Processor can be coupled with any integration platform that might be used within an organization.

However, this approach requires that protocols for communicating with the Framework Processor be defined and that an interface between the Framework Processor and the chosen integration platform be constructed. In the case of the FPP, the Framework Processor will mainly interact with the higher level components (i.e., Artifact Manager and Integration Mechanism). As a result, interfaces will only have to be defined between those components and the Framework Processor. With these interfaces in place, the Framework Processor would simply respond to queries from the various components of the FPP about the framework contents. In addition, the various components would send messages to the Framework Processor to indicate the occurrence of events during the evolution of active projects. So, instead of a Framework Processor controlling the operation of every component, each component will access the information about the framework to ensure that the constraints of the framework are not violated.

2.1.5 Facilitator

The *Facilitator* serves as a dispatcher of messages between higher level components (Artifact Manager and Integration Mechanism) and the lower level components (Services and Network Transaction Manager). This separation between higher and lower level components is required since the Data Managers and Network Transaction Managers will be more machine dependent than the more portable Artifact Manager and Integration Mechanism. The Facilitator will provide a common interface between these two levels so that the impact of changes in one level will be reduced, if not eliminated, in the other level.

The Facilitator is also required because of the distributed nature of the FPP. When accessing data, whether that data resides on the local machine or on

a remote machine should be transparent to the Artifact Manager. To hide the location of the data from the Artifact Manager requires an intermediate party to parse the data id and route the data query to either the appropriate machine (through the Network Transaction Manager) or the appropriate data manager on the local machine. The Facilitator will be responsible for this routing of data requests. In a similar manner, the location of a functional service being accessed by the Integration Services Manager should be transparent to the Integration Services Manager. Again, an intermediate party is required to route the service request to the host on which the requested service is available. The Facilitator will be responsible for routing these service requests as well.

2.1.6 Services

The *Services* component represents the link between the FPP and the local host operating system. Actually, the Services component is not one single program, but rather a collection of programs that share a common interface with the FPP. These are the components that do the real work and provide the FPP with considerable flexibility. Ideally, each Service program performs one specialized task. This allows each of the programs to be quite small, and since the FPP is required to handle multiple service requests at the same time, this allows the impact of the multiple threads on system resources to be quite minimal. In order for a particular service to be accessible to the FPP, it must be registered in the Service Repository.

It is also through this Services interface that the FPP will access all system-specific and artifact-specific data. This will be accomplished by defining a data service built on top of a database system running at a particular site. Through the protocols established, the FPP will be able to access and manipulate data independent of the specific database used.

2.1.7 Network Transaction Manager

The *Network Transaction Manager (NTM)* is responsible for sending and receiving network operations for the FPP running on the local machine. The operations might include data queries or updates to database managers running on other machines, request for service execution on remote machines, or simple network file transfer operations. The goal of the NTM is to provide a common networking interface between different FPP nodes that provides a higher level of abstraction than the many existing networking protocols.

The combination of the Network Manager on the local machine and the Network Manager on the remote machine serves as a bridge between one Facilitator and another. The messages that the Facilitator receives from the Network Manager will be in the same format as those messages received from the Platform Interface, and the data returned to the Facilitator from the Network Manager will be in the same format as the data returned from the Data Services. Thus, the Facilitator could treat the Network Manager as simply a special service. (This is not completely true;

some operations require that the Facilitator retry a request on a remote machine after failing on the local machine. One would not want the remote machine to fail and retry remotely as well. This, however is a special case.)

2.2 FPP Operation

With a description of each major component of the FPP provided, it is now necessary to describe exactly how the various components will work together to satisfy the functional requirements of the FPP [FPP 90b]. The functionality provided by the FPP can be categorized as either Service Management or Framework Processing capabilities. This section will give a description of each of these functional categories and how the components of the FPP architecture support these categories.

2.2.1 Services Management

Services Management refers to the basic areas of functionality to be supported by the FPP: Artifact Management and Integration Services support. These capabilities are designed to effectively integrate data residing and tools running on different hardware platforms at a particular site. Artifact Management support allows users to maintain a registry of a life cycle artifacts and maintains control over access to and modification of those artifacts. Integration Services support allows users and applications access to defined services on local and remote machines. Taken together, these two capabilities provide an integrated environment by allowing tools to share data as well as functionality.

Initiation of either of these two FPP services is achieved through the Application Interface. Three types of operations are currently accessible through the Application Interface:

1. Functional Service Requests,
2. Artifact Management Operations, and
3. System Commands.

Functional Service Request operations represent requests for operations that are accessible through the Services interface, either locally or on a remote machine, described in Section 2.1.6. Artifact Management Operations represent requests for the manipulation of software life cycle artifacts. This group of requests includes such operations as Artifact Check In, Artifact Check Out, and Artifact Browse. Finally, System Commands represent operations that are supported directly by the local host operating system. These commands essentially give the FPP the ability to shell to the host operating system.

A system can access these operations by passing service request messages to the Application Interface. The structure of these messages must adhere to the Service Request Language Grammar defined in Appendix D and multiple service requests can be bundled into one message. When the

Application Interface receives a service request message, the Application Interface iteratively processes each request in the message bundle (in most cases there will only be one request). Based on the type of request being made (i.e., Functional, Artifact, or System), the Application Interface will route the request to the appropriate component. For Functional Service Requests or System Commands, the request message is routed directly to the Integration Mechanism. For Artifact Management Operations, the request message is routed directly to the Artifact Manager.

2.2.1.1 Function Service and System Operations

In the case of a Functional Service Request or System Command, the message is routed to the Integration Mechanism, and more specifically to the Integration Services Manager. If the message contains a System Command, the message is passed directly to the Facilitator, with no special action taken by the Integration Services Manager. However, if the message contains a Functional Service Request, a service plan must be generated before the execution plan can be passed to the Facilitator.

The first step in the service plan generation is a pre-plan lookup. The Integration Mechanism maintains a library of previously generated service plans. In the event that the requested service has been provided before, the plan will exist in this library and can be simply recalled. If, however, no pre-existing plan is found, the service request is passed to the Integration Services Planner. At this point, the Integration Services Planner attempts to generate a functional plan, a hardware and software independent form of an executable plan, based on the knowledge base of planner objects. These planner objects are defined for every service accessible by the FPP and are defined using the Service External Representation Language found in Appendix C. If no functional plan can be generated, the Integration Mechanism returns an unsupported service error to the Application Interface. If a functional plan can be generated, the resulting functional plan is transformed into an executable plan by determining which utilities will perform each step of the functional plan. Once an executable plan is generated, the plan is returned to the Integration Services Manager and is executed. Execution, as far as the Integration Services Manager is concerned, simply involves passing the executable service plan to the Facilitator and monitoring the progress of the plan execution.

When the Facilitator receives an executable service plan, the plan is processed by iterating over each step in the plan. For each step, the Facilitator determines if the step is to be executed locally or remotely. For local operations, the Facilitator spawns a process on the local machine to execute the desired operation. For remote operations, the Facilitator bundles the step explanation in a network message, as described in Appendix F, and passes the message to the Network Transaction Manager. In either case, the Facilitator waits for execution of the step to complete before beginning execution of the next step. As steps complete, the Facilitator collects intermediate results until the final step is completed, at

which point the Facilitator returns the service results to the Integration Services Manager.

Since the Facilitator waits for each step in a service plan to complete before proceeding, it is necessary to spawn multiple Facilitators to allow multiple services to be executed at the same time. Relying on a single Facilitator could result in a tremendous backlog of service requests, especially if the service currently being executed involves a long term action. To prevent this situation, the Integration Services Manager will spawn a Facilitator process for each service plan the ISM needs to have executed.

2.2.1.2 Artifact Operations

The previous scenario has shown how the components of the FPP are used to access and use Integration Services. Another capability provided by the FPP is that of Artifact Management. This capability is concerned with the management and control of software life cycle artifacts. As mentioned, Artifact Operations are received by the Platform Interface and routed to the Artifact Manager component of the FPP. The strategy taken by the Artifact Manager in processing these artifact operations is to first determine whether the operation should be executed and, if so, to actually perform the operation.

The first requirement that must be met for the operation to be performed deals with user authorization. As the FPP is intended to manage artifacts that are considered important to an organization, it is imperative that the Artifact Manager enforce access control policies established by the organization. These policies specify what users should have access to the artifacts managed by the Artifact Manager. As a result, every artifact operation request must be passed through an authorization procedure to ensure that the user submitting the artifact operation should be allowed to complete the operation. If the authorization fails, the operation is terminated and the user is sent an authorization failure message.

In the event that user authorization is approved, conditions that may prevent the execution of the operation are analyzed. For the most part, these conditions would involve a certain artifact being in a state that does not allow the requested operation to be performed. For example, if a user requests to Check Out a specific artifact, the Artifact Manager must determine if some other user has already checked out the artifact, in which case the artifact is locked and no other users can access the artifact (except on a read only basis).

If it is determined that no conditions prevent the execution of the requested operation, the Artifact Manager executes the necessary steps to perform the operation. In the case of a Check Out operation, the artifact information is updated to reflect that the artifact has been checked out and a message is sent to the user indicating where the artifact is located. For a Check In operation, the artifact information is updated and the Artifact Manager copies the artifact into the artifact library.

Looking back at Figure 1, notice that the Artifact Manager is layered on top of the Integration Mechanism. The reason for this is that the Artifact Manager takes advantage of the functionality provided by the Integration Mechanism to store and manipulate artifact data and access the host operating system (for file manipulation). Recall that the FPP will use the Services interface to provide access to databases maintaining FPP system information. The Artifact Manager is one component that will follow this practice. The advantage of this approach is that artifact information does not necessarily have to reside on the same machine that the Artifact Operation originated from. Because of the operation of the Integration Mechanism and the Facilitator, the artifact information can be distributed across all the machines running in the FPP environment.

Following this approach, when the Artifact Manager needs access to data pertaining to a specific artifact or group of artifacts, the Artifact Manager will produce a Function Service Request. The structure of this particular type of service request will adhere to the Data Query Language described in Appendix B. Once the service request is generated, the request will be sent to the Integration Mechanism and processed in the same manner as described above. When the Artifact Manager initiates a Functional Service Request to the Integration Mechanism, it is likely that the requested service will be a data service defined specifically for the operation of the FPP. Except for the initial request, this will almost ensure that a service plan for the requested service will exist in the plan library, thus reducing overhead that could arise from the Artifact Manager relying on the Integration Mechanism for execution of functional services.

2.2.2 Framework Processing

Framework Processing refers to the ability to use a representation of the system development process (i.e., the framework) to manage and control that development process. The Framework Processor component of the FPP will be responsible for taking a framework as input and using the information maintained in the framework to monitor various software development projects within an organization. Among other things, the framework will contain information pertaining to:

1. the development process to be followed in producing a software system,
2. the methods and tools that should be used in producing the software,
3. the users and user role types that will be responsible for tasks within the process, and
4. the artifacts produced and manipulated during the process.

The role of the Framework Processor is to extract this information from the framework representation, to make the information available to

components of the platform, and to use the information to monitor and control the progress of software development projects.

In the design of the FPP, effort has been taken to separate the Framework Processing capability from the integration platform aspects of the FPP. As a result, the Framework Processor operates somewhat independently of the of the remaining components of the platform. Instead, the Framework Processor has an interface built in that allows components of the platform to query the Framework Processor for information about the development process. Similarly, this interface has protocols built-in that allow the platform components to notify the Framework Processor of the occurrence of certain events. It is through these event notifications that the state of a particular development project will be updated.

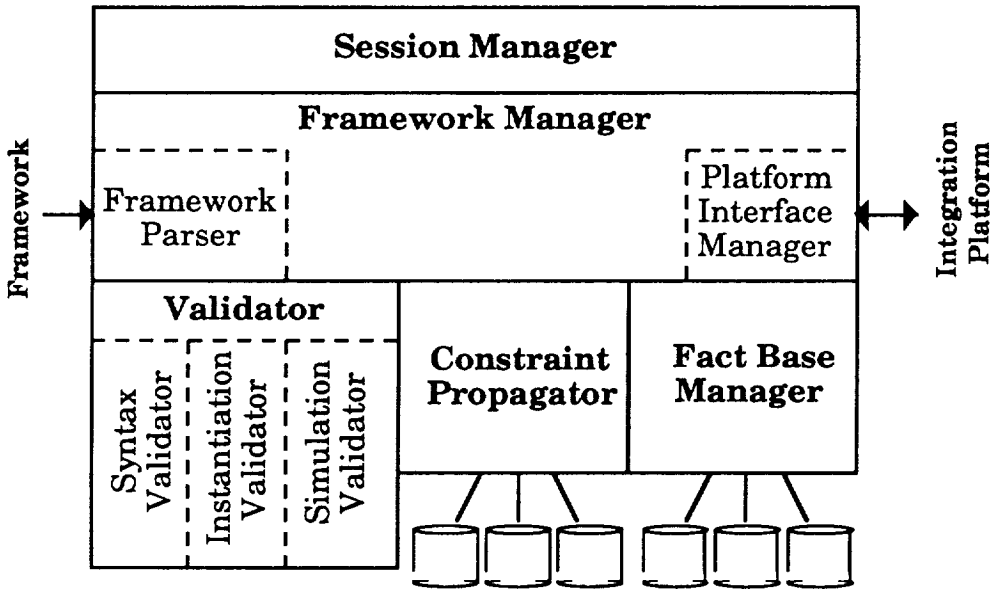

Figure 2. Framework Processor Architecture

Essentially, what this discussion says is that the Framework Processor is meant to lie between a framework definition and an organization's development environment (integration platform). The architecture for the Framework Processor reflecting this idea is shown in Figure 2. The basic operational philosophy of the Framework Processor is to take a framework as input, perform several validation checks on the framework, translate the framework into a set of constraints and facts, and then use the facts and constraints to control the development process. During this process, the set of facts and constraints are continuously updated as a result of actions by users and messages from the integration platform (i.e., the notification of the occurrence of certain events). This dynamic situation is continuously monitored to detect inconsistences between the process specified in the framework and the actual events occurring during the system development.

The overall operation of the Framework Processor is controlled by the Framework Manager. Prior to framework installation, the Framework Manager coordinates the framework validation process by parsing the framework and extracting the appropriate information for the Validator component. Three levels of validation are performed by the Validator component and each of these validation steps are complex enough to require an individual sub-component within the Validator. In the event that inconsistencies are detected, the Framework Manager would assist the framework administrator in correcting the inconsistencies.

After validation has been completed, the framework is installed and project instantiations can be performed. Once a project has been instantiated, the Framework Manager begins to monitor and control that project development. At project instantiation, an initial set of assertions are passed to the constraint propagator and a set of facts are passed to the Fact Base Manager. From that point, information about operations and events performed and requests for authorization by project members are continuously passed to the Framework Manager from the users (through the Session Manager) and the Integration Platform. This information is then passed to the appropriate knowledge base (i.e., constraint base or fact base). If contradictions or inconsistencies are detected, it is up to the Framework Manager to take appropriate action to resolve the conflict. If no problem is detected, the operations are performed, and the project state is updated.

Notice that the knowledge bases have been split between facts and constraints, and accordingly two different components have been devised to manage those knowledge bases. The reason for this is that the information represented in the framework is so diversified that using a single reasoning scheme became impractical. Instead, the fact base and the Fact Base Manager capture and manage information about access privileges, users, user roles, etc., while the constraint base and the Constraint Propagator contain and maintain the state of the project development process. As such, the Fact Base Manager processes the static framework information, while the Constraint Propagator processes the dynamic process-oriented framework information.

Since the current state of the development process will be maintained by the Framework Processor, the Framework Processor will provide an interface that gives the user a visualization of the state of the current project. This visualization can be used to determine what areas the user should focus their attention on. This visualization will reflect all constraints placed on the user by the framework definition and will allow the user to log into a specific project, check the status of the project, inquire about open tasks, and browse the system development process. This direct interface provides a nice complement to the indirect interfaces that exist between the platform components and the Framework Processor. With the indirect interfaces, it is difficult for a user to get a view of the "big picture." That is, the interfaces do not allow the user to place their current activities in context with other

tasks in the development effort. The visualization provided by the Framework Processor will allow the user to determine the current context and to better understand why certain operations are not allowed.

3 FPP Detailed Design

Given the previous discussion identifying the major components of the FPP architecture and describing how they operate and interact, it is now possible to provide more detailed designs for the individual components. Not surprisingly, the component modules to be detailed in this section parallel very closely to the conceptual components of the FPP architecture. The following list identifies the component modules that will make up the FPP and the order in which the modules will be discussed.

1. Platform Interface,
2. User Session Interface,
3. Artifact Manager,
4. Integration Mechanism,
5. Framework Processor,
6. Facilitator,
7. Network Transaction Manager, and
8. Services.

Before proceeding, a mention of the convention for describing these modules is necessary. Each of the elements in the list above represent modules that will be implemented in the FPP. Each of these modules has been broken down into submodules that will be required to provide the necessary functionality of the module. These submodules are first identified and then descriptions of each are provided. In many cases, a table is used to convey information such as input and output data as well as related submodules.

3.1 Platform Interface

The Platform Interface module provides the protocols necessary for applications to access the functionality provided by the FPP. The Platform Interface module has been broken down into five submodules:

1. Accept Service Request,
2. Perform Access Authorization,
3. Perform Service Request,
4. Monitor Service Request, and
5. Return Service Results.

Due to the fact that the Integration Platform could potentially process many service requests at the same time, the submodules of the Platform Interface must operate in an asynchronous fashion. Once a service request is received, the Platform Interface validates the request, through the Perform Access Authorization submodule. After the request is deemed valid, the

Perform Service Request submodule routes the request to the appropriate manager. The Monitor Service Request keeps up with the current status of the request while it is being serviced. When the processing of the request is complete, the Return Service Results submodule relays the result back to the application.

The Platform Interface receives input from three sources: 1) the Artifact Manager, 2) the Integration Services Manager, and 3) External Applications. The only external interface is with the Applications which communicate requests to the Platform Interface and receive results in return.

3.1.1 Accept Service Request

The purpose of the Accept Service Request submodule of the Platform Interface is to wait for an application to generate a request for service. This submodule operates by continuously checking for service requests sent by applications.

Input Data:	Application Message
Output Data:	Service Request Error Message (syntax)
Other Elements	Applications Perform Access Authorization

Figure 3. Accept Service Request Data Information

Figure 3 summarizes the data elements manipulated by the Accept Service Request unit. The Accept Service Request submodule builds a Service Request from the Application Message it receives from an application. This submodule must check the incoming message for syntactic validity. An error message is returned if the message is not valid.

3.1.2 Perform Access Authorization

The purpose of the Perform Access Authorization submodule is to control the access of objects and services supported by the FPP.

Input Data:	Service Request
Output Data:	Error Message (access violation)
Other Elements:	Accept Service Request Service Request Log Perform Service Request Access Policy Database

Figure 4. Perform Access Authorization Data Information

Figure 4 summarizes the data elements manipulated by the Perform Access Authorization unit. The Accept Service Request unit requests authorization for the request from the Perform Access Authorization unit. This submodule then checks the user's password, id, and role against the access policy database to verify that the request is acceptable. If the request is verified, the service request can then be passed on to be processed. If the request is not verified, an error message is reported and logged. Whether or not a service request is passed on to the Perform Service Request submodule depends on verification of the request by the Perform Access Authorization submodule.

3.1.3 Perform Service Request

The purpose of the Perform Service Request unit is to submit the requests to be serviced by the other components of the Integration Platform after the requests have been checked for syntax and authorization.

The Perform Service Request unit accepts service requests from the Perform Access Authorization submodule. There are three types of requests that must be handled by this module. These are 1) Artifact requests, 2) System requests, and 3) Functional service requests. The Perform Service Request decides which component should handle each request and then routes the service request to that component.

Figure 5 summarizes the data elements manipulated by the Perform Service Request submodule. The Perform Service Request submodule waits to receive Service Requests from the Perform Access Authorization submodule. Once a request has been received, the Perform Service Request unit dispatches the request to the appropriate manager. As this request is sent, the Perform Service Request unit also makes an entry in the Service Request Log. This entry will be accessed by the Monitor Service Request unit to monitor the status of the service request. If the Perform Service Request submodule cannot determine which manager should service the request, an error is reported. As a special case, the Perform Service Request submodule will service such system requests as Login and Logout.

Input Data:	Service Request
Output Data:	Error Message (unknown handler)
Other Elements:	Service Request Log Entry Perform Access Authorization Monitor Service Request Artifact Manager Integration Services Manager Access Policy Database

Figure 5. Perform Service Request Data Information

3.1.4 Monitor Service Request

The Monitor Service Request submodule monitors the progress of the service requests submitted by the Perform Service Request unit. This submodule checks the requests status and notices when special conditions arise (e.g. termination or error conditions).

Input Data:	Service Request Log Entry
Output Data:	Service Results Message
Other Elements:	Return Service Results Artifact Manager Integration Services Manager

Figure 6. Monitor Service Request Data Information

Figure 6 summarizes the data elements manipulated by the Monitor Service Request submodule. Once a service request has been submitted, the Platform Interface must monitor its progress. The Monitor Service Request unit will collect the results of the service and transmit those results to the Return Service Results submodule via a Service Results Message.

3.1.5 Return Service Results

The purpose of the Return Service Results submodule is to return the results of a request to the application which requested it.

Input Data:	Service Results Message
Output Data:	Service Results
Other Elements:	Monitor Service Request Service Request Log

Figure 7. Return Service Results Data Information

Figure 7 summarizes the data elements manipulated by the Return Service Results unit. This submodule takes a Service Results Message sent by the Monitor Service Request submodule and matches it with the service request that initiated it. This information can be found in the Service Request Log. Once the initiator of the request is identified, the results can be sent back to the application that requested the service.

3.2 User Session Interface

Though the FPP Architecture does not specifically contain a User Interface component, it is important to provide such an interface to allow direct access to the FPP functionality. This interface will not only allow users to browse and become familiar with the operation of the FPP but will also serve as a useful testing mechanism for the FPP components. The User Session Interface module consists of three submodules:

1. Process User Gestures,
2. Make Service Query, and
3. Present Service Results.

This component operates like an application which is connected to the FPP. It is necessary to the design in that it performs many of the administrative tasks during the operation of the Integration Platform. It can request services in the same way that an application can, only with more control by the user. Requests that are essential to the operation of the Integration Platform, such as system administration, are accomplished through the User Session Interface.

3.2.1 Process User Gestures

The Process User Gestures submodule would serve as the main command loop for the User Session Interface. The Process User Gestures component is responsible for collecting the input from the user and translating it into a data structure that can be processed by the platform. User Gestures is a term given collectively to the different types of inputs that a user might use to initiate a command, make a selection, choose a menu item, etc.

Input Data:	User Gestures
Output Data:	User Data Structure
Other Elements:	Platform Interface

Figure 8. Process User Gesture Data Information

Figure 8 summarizes the data elements manipulated by the Process User Gesture unit. The user interacting with the User Session Manager generates requests for services for the FPP to handle. These user gestures (i.e. mouse clicks, menu selections, commands, etc.) are interpreted by the Process User Gesture submodule. The resulting message is constructed and sent to the appropriate submodule, (i.e. Make Service Query or Process Service Request).

3.2.2 Make Service Query

The purpose of the Make Service Query submodule is to allow the user, during a user session, to access the databases which support the FPP.

Input Data:	Application Message
Output Data:	Service Request
Other Elements:	Platform Interface Process User Gestures

Figure 9. Make Service Query Data Information

Figure 9 summarizes the data elements manipulated by the Make Service Query unit. The Make Service Query submodule accepts the application request messages and converts them into a format that is understood by the Platform Interface. For example, the user may wish to browse the information kept about the artifacts currently managed by the platform. This request is made known to the system through the Process User Gestures component which interprets the command from the user. This information is sent to the Make Service Query submodule and made into a proper service request to be handled by the Services Manager.

3.2.3 Submit Service Request

The purpose of the Submit Service Request submodule is to gather the necessary information for a user service request to be made to the Platform Interface. This submodule takes the information from the user gestures

unit and converts it into a service request which can be sent to the Platform Interface.

Input Data:	User Data Structure
Output Data:	Service Request
Other Elements:	Accept Service Request User Interface Data Platform Interface

Figure 10. Submit Service Request Data Information

Figure 10 summarizes the data elements manipulated by the Submit Service Request unit. At this point, the User Session Interface will operate like any other application interacting with the Integration Platform. Once service request information has been input by the user, it is possible for the Submit Service Request unit to submit a Service Request to the Accept Service Request unit of the Platform Interface. The purpose of the Submit Service Request unit is to collect the service request information maintained in the data structures of the User Session Interface, generate a Service Request, and transmit that message to the Accept Service Request unit.

3.2.4 Present Service Results

The Present Service Results submodule returns the result from a user session to the user in a form that can be viewed by the user.

Input Data:	Service Results Message
Output Data:	Results Display
Other Elements:	Services Manager Service Request Log

Figure 11. Present Service Results Data Information

Figure 11 summarizes the data elements manipulated by the Present Service Results unit. The Present Service Results submodule receives a service results message from the Platform Interface and displays the results to the user. If there is an error result from the request, an appropriate error message is presented.

3.3 Artifact Manager

The Artifact Manager module provides the all functionality necessary to manage and control the manipulation of life cycle artifacts. The Artifact Manager module consists of the following eight submodules:

1. Distribute Service Request,
2. List Design Artifacts,
3. Check Out Artifact,
4. Check In Artifact,
5. Create New Artifact Version,
6. Create Artifact Configuration,
7. Execute Data Operation, and
8. Execute Artifact Operation.

The List Design Artifacts, Check Out Artifact, Check In Artifact, Create New Artifact Version, and Create Artifact Configuration submodules represent high level functions supported by the Artifact Manager. When the Artifact Manager receives an artifact service request from the Platform Interface, the Distribute Service Request submodule will pass the request on to one of these five submodules for proper execution.

3.3.1 Distribute Service Request

This submodule is responsible for distributing the service requests received from the Platform Interface to the proper submodule of the Artifact Manager.

Input Data:	Artifact Service Request
Other Elements:	Platform Interface Other Artifact Manager modules

Figure 12. Distribute Service Request Data Information

Figure 12 summarizes the data element manipulated by the Distribute Service Request submodule. This submodule accepts input from the Platform Interface and distributes the Artifact Service Requests to the various components of the Artifact Manager. This submodule has no output since it does nothing but pass on the input that it received.

3.3.2 List Design Artifacts

This submodule generates a list of all artifacts currently managed by the Artifact Manager. The list may be limited by providing search criteria which will filter out artifacts which are not needed by the user.

Input Data:	List Artifact Command
Output Data:	List of Artifact Object Descriptions
Other Elements:	Platform Interface Execute Artifact Operation

Figure 13. List Design Artifacts Data Information

Figure 13 summarizes the data elements manipulated by the List Design Artifacts submodule component of the Integration Platform. This submodule takes a List Artifact command and converts it into an Artifact Operation message that is sent to the Execute Artifact Operation submodule. The results of that operation will be a list of the artifacts requested. These results will be returned to the Platform Interface as the results of the List Design Artifacts submodule.

3.3.3 Check Out Artifact

This submodule allows users access to design artifacts which are registered in the Artifact Manager.

Input Data:	Check Out Artifact Command
Output Data:	Artifact Object Error Message (access violation)
Other Elements:	Platform Interface Execute Artifact Operation Execute Data Operation

Figure 14. Check Out Artifact Data Information

Figure 14 summarizes the data elements manipulated by the Check Out Artifact submodule. When an application requests an artifact, the Check Out Artifact submodule marks the artifact as being in use and restricts access by other applications until the artifact is checked back in to the

repository. This is accomplished by creating Artifact Operation messages for consumption by the Execute Artifact Operation submodule and Data Operation messages for consumption by the Execute Data Operation submodule. The Check Out Artifact submodule will determine whether or not the artifact in question is already checked out by way of an Artifact Operation message. Then the artifact will be retrieved by way of a Data Operation message. Finally, the artifact will be marked "checked out" by way of another Artifact Operation message. The artifact will then be returned to the Platform Interface.

3.3.4 Check In Artifact

This submodule provides the functionality for registering artifacts with the Artifact Manager.

Input Data:	Check In Artifact Command
Output Data:	Confirmation Message Error Message (access violation)
Other Elements:	Platform Interface Execute Artifact Operation Execute Data Operation

Figure 15. Check In Artifact Data Information

Figure 15 summarizes the data elements manipulated by the Check In Artifact. This submodule receives an artifact check in command and updates the status of the artifact in the Artifact Repository. After this operation the artifact in question is now available for check out by another user or application. Only artifacts that are checked out by a specific user may be checked in by that user. This process is accomplished by sending an Artifact Operation message to the Execute Artifact Operation submodule to determine: 1) if the artifact previously existed and 2) if it did exist, who, if anyone, is it checked out to. If access should be allowed, the artifact is updated by means of a Data Operation message being sent to the Execute Data Operation submodule. The artifact status will also be updated through another Artifact Operation message. If all went well, a confirmation message will be returned to the Platform Interface.

3.3.5 Create New Artifact Version

The purpose of the Create New Artifact Version is to provide a facility to retain multiple versions of an artifact in order to maintain the history of the development of the artifact.

Input Data:	Create New Artifact Command
Output Data:	Confirmation Message Error Message (access violation)
Other Elements:	Platform Interface Execute Artifact Operation

Figure 16. Create New Artifact Version Data Information

Figure 16 summarizes the data elements manipulated by the Create New Artifact Version submodule. This submodule provides the ability to create new versions of an artifact as the artifact goes through the stages of design. When a new version is needed the name of the artifact is sent to the Create New Artifact Version component. This component then updates the Artifact Repository to reflect the new version information. The old versions are still available to allow for an audit trail which can be used to review the development of the artifact to its present state. This process is accomplished by a series of Artifact Operation messages being sent to the Execute Artifact Operation submodule. If all goes well, a confirmation message is returned to the Platform Interface.

3.3.6 Create Artifact Configuration

The purpose of the Create Artifact Configuration is to allow the user to group related artifacts into a configuration.

Input Data:	Create Artifact Configuration Command
Output Data:	Confirmation Message Error Message (invalid artifact)
Other Elements:	Platform Interface Execute Artifact Operation Container Object System Artifact Repository

Figure 17. Create Artifact Configuration Data Information

Figure 17 summarizes the data elements manipulated by the Create Artifact Configuration submodule. This submodule allows a user to create a configuration which collects several artifacts together as a group. This

would allow the logical linking of related artifacts which are used or needed together. When a configuration is created, the artifact repository is updated by adding the configuration object. Just as in the previous submodules, this is accomplished by sending a series of Artifact Operation messages to the Execute Artifact Operation submodule. If all goes well, a confirmation message will be returned to the Platform Interface.

3.3.7 Execute Data Operation

This is the submodule responsible for making a request to the Integration Services Manager to actually perform an operation on a design artifact.

Input Data:	Data Operation Message
Output Data:	Data Operation Results
Other Elements:	Other Artifact Manager modules Integration Services Manager Artifact Repository Container Object System

Figure 18. Execute Data Operation Data Information

Figure 18 summarizes the data elements manipulated by the Execute Data Operation submodule. Up to this point operations have been defined on the information stored about the artifacts in the repository. This submodule allows operations to be performed on the artifact itself. The artifact location and format information is retrieved from the Artifact Manager. From this, the Integration Services Manager is invoked with a message to perform the data operation service on the artifact (e.g. load, save).

3.3.8 Execute Artifact Operation

The Execute Artifact Operation submodule is responsible for manipulating the objects used to represent a life cycle artifact.

Figure 19 summarizes the data elements manipulated by the Execute Artifact Operation submodule. This submodule is used to perform various housekeeping functions on the artifact information managed by the Artifact Manager. Artifact Operations are made to update information associated with each artifact (e.g. location, format).

Input Data:	Artifact Operation Message
Output Data:	Artifact Operation Results
Other Elements:	Other Artifact Manager modules Integration Services Manager Container Object System

Figure 19. Execute Artifact Operation Data Information

3.4 Integration Mechanism

The Integration Mechanism is responsible for the execution of Function Service Request and System Commands. The Integration Mechanism consists of the following five submodules:

1. Accept Service Request,
2. Request Service Plan,
3. Execute Service Plan,
4. Integration Services Planner, and
5. Service Registration Manager.

As these submodules are relatively complex, several of the submodules will be further decomposed. These decompositions will be reflected in the discussion of each submodule.

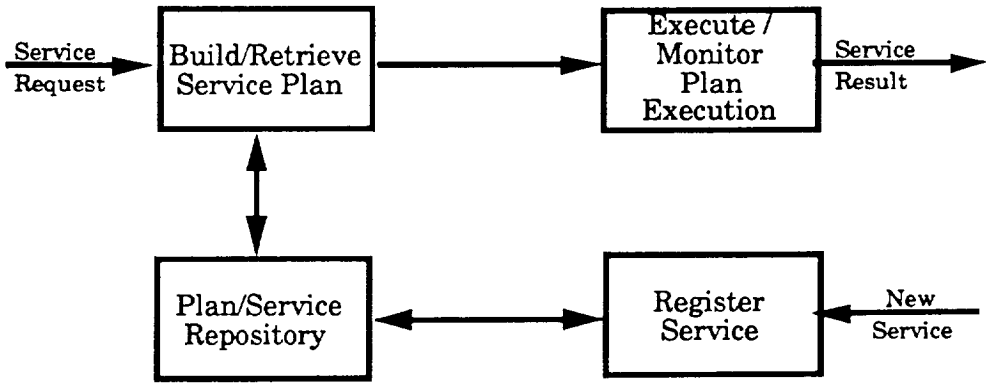

Figure 20. Integration Mechanism Operation

Figure 20 illustrates the general operation of the Integration Mechanism. A service request coming in causes a service plan to be produced. This is done either through lookup or by generating a plan. The resulting plan, if it can be found or produced, is executed by the Service Executor and the results are returned.

3.4.1 Accept Service Request

The Accept Service Request submodule is responsible for accepting Functional Service Requests from either the Application Interface or the Artifact Manager and directing the flow of control in the submodules of the Integration Mechanism.

Input Data:	Functional Service Request
Output Data:	Service Results Error Message (semantics, no plan, plan error)
Other Elements:	Platform Interface Artifact Manager Service Registration Manager Request Service Plan Execute Service Plan Failure Recovery

Figure 21. Accept Service Request Data Information

Figure 21 summarizes the data elements manipulated by the Accept Service Request submodule. After a Functional Service Request is received, the first step is to check the request for semantic validity. (Requests from users have already passed syntactic checks in the Platform Interface; requests from other managers is trusted.) Next, an Executable Service Plan is obtained from the Request Service Plan submodule. Third, this service plan is sent to the Execute Service Plan submodule for execution and monitoring. If an error occurs during processing, the Failure Recovery submodule is called with both the plan and the error. If Failure Recovery determines that the error is recoverable, it returns a new Executable Service Plan and the process repeats with step three. If there was no error in step three or the error was deemed non-recoverable, then the results are returned to the caller.

3.4.2 Request Service Plan

The purpose of the Request Service Plan is to retrieve a service plan from the Service Registration Manager.

Input Data:	Functional Service Request
Output Data:	Executable Service Plan Error Message (no plan)
Other Elements:	Service Registration Manager Integration Services Planner Accept Service Request

Figure 22. Request Service Plan Data Information

Figure 22 summarizes the data elements manipulated by the Request Service Plan submodule. The Request Service Plan unit first requests the Service Registration Manager to search the Plan Repository for a stored plan which can carry out the request. If a pre-generated plan is not stored in the repository, one must be generated. This is accomplished by the Integration Services Planner component. Once the plan is returned, either from the Service Plan Registration Manager or from the planner, the Request Service Plan submodule sends the plan to the Execute Service Plan submodule. If a plan cannot be found or generated, the Request Service Plan unit generates an error to that effect.

3.4.3 Execute Service Plan

The purpose of the Execute Service Plan submodule is to construct a multi-step service request from the original service request and the executable service plan received from the Integration Services Planner. The service request plan generated may contain only one service request, if a service is found that can handle the request as it is given. In general, the service request will be made into several steps which, when executed, will satisfy the service request. It must also relay an Executable Service Plan to the Facilitator and to monitor its execution.

Figure 23 summarizes the data elements manipulated by the Execute Service Plan submodule. This submodule takes the Executable Service Plan that it received from the Accept Service Request submodule and relays it to the Facilitator. It then monitors the progress of the plan. If all went well, the results of the operation are returned. If not, an error message is returned that estimates how far the execution got before it failed.

Input Data:	Executable Service Plan Service Request
Output Data:	Service Results Error Message (plan error)
Other Elements:	Request Service Plan Facilitator Accept Service Request

Figure 23. Execute Service Plan Data Information

The Execute Service Plan submodule can actually be decomposed into the following four submodules:

1. Step Completion Detection,
2. Step Failure Detection,
3. Failure Recovery, and
4. Perform Plan Context Switch.

3.4.3.1 Step Completion Detection

The purpose of the Step Completion Detection submodule is to detect when an operation represented by a step in the service plan has completed executing.

Output Data:	Notification Message
Other Elements:	Facilitator Route Executable Service Plan Results Plan Status Table

Figure 24. Step Completion Detection Data Information

Figure 24 summarizes the data elements manipulated by the Step Completion Detection submodule. This unit monitors the execution of each service plan being processed. When a step terminates, the Plan Status Table is updated to reflect the current state and the next step in the plan can be started.

3.4.3.2 *Step Failure Detection*

The purpose of the Step Failure Detection submodule is to monitor the execution of plans to determine if a failure occurs.

Output Data:	Plan Status Table
Other Elements:	Facilitator Route Executable Service Plan Results Plan Status Table

Figure 25. Step Failure Detection Data Information

Figure 25 summarizes the data elements manipulated by the Step Failure Detection submodule. This unit updates the Plan Status Table in the event of an error in one step of a service plan.

3.4.3.3 *Failure Recovery*

The purpose of the Failure Recovery submodule is to attempt to recover from errors in an Executable Service Plan.

Input Data:	Executable Service Plan Error Message
Output Data:	Executable Service Plan Error Message (non-recoverable)
Other Elements:	Service Registration Manager Accept Service Results

Figure 26. Failure Recovery Data Information

Figure 26 summarizes the data elements manipulated by the Failure Recovery submodule. When an error occurs midway through the execution of a service plan, it could be that the services listed in the Service Registration database is out of date or in error. If the error resulted from not being able to execute a particular Data Service, then the Service Registration Manager is asked to generate a new Executable Service Plan that does not use a particular service. If the error was due to a syntax error in the translation of an artifact, then nothing can be done: either it is a bug

in the Data Service or a bug in the artifact (most likely the latter). Network errors are non-recoverable in this sense; the Functional Service Request should simply be resubmitted when the network is again active.

3.4.3.4 Perform Plan Context Switch

The purpose of the Perform Plan Context Switch submodule is to allow the Service Plan Executor to process more than one plan at a time by switching to another plan while a step in the first plan is executing.

Input Data:	Plan Status Table
Other Elements:	Step Completion Detection Facilitator Accept Executable Service Plan Route Executable Service Plan Results

Figure 27. Perform Plan Context Switch Data Information

Figure 27 summarizes the data elements manipulated by the Perform Plan Context Switch submodule. This unit allows the Execute Service Plan submodule to switch to another plan while a step in a plan is executing. This allows many plans to be invoked at the same time and simultaneous execution of various steps from different plans. When a step is sent to be executed, the Perform Plan Context Switch checks the plan status table to find a plan that is ready to execute the next step. This plan is made active and the next step is begun. The process then continues in this manner.

3.4.4 Integration Services Planner

The Integration Services Planner module is responsible for generating valid service plans in response to a service plan request from the Request Service Plan module. The Integration Services Planner consists of the following five submodules:

1. Lookup Service Plan,
2. Generate Functional Plan,
3. Perform Service Advertisement Search,
4. Perform Service Protocol Search, and
5. Generate Executable Plan.

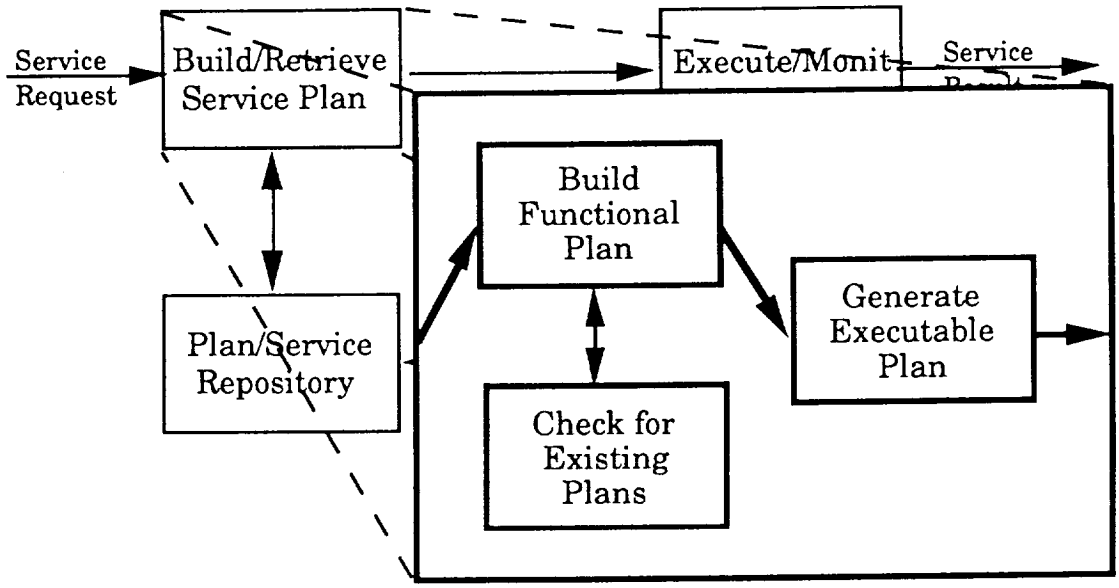

Figure 28. Integration Services Planner Operation

Figure 28 illustrates the general operation of the Integration Services Planner module. A submodule exists for each box represented in this diagram. The Perform Service Advertisement Search and Perform Service Protocol Search submodules are used in support of the Generate Functional Plan submodule.

3.4.4.1 Lookup Service Plan

The purpose of the Lookup Service Plan is to check the Service Plan Cache for the given Functional Service Request to see if a functional plan exists already for the requested service.

Input Data:	Functional Service Request
Output Data:	Functional Service Plan Error Message (no plan)
Other Elements:	Service Plan Cache Generate Functional Plan Generate Executable Plan

Figure 29. Lookup Service Plan Data Information

Figure 29 summarizes the data elements manipulated by the Lookup Service Plan submodule. The Functional Service Request is used as a key

into the Service Plan Cache. If an associated Functional Service Plan exists then execution continues with the Generate Executable Plan submodule, otherwise a functional plan is generated by the Generate Functional Plan submodule.

3.4.4.2 Generate Functional Plan

The purpose of the Generate Functional Plan submodule is to provide a functional plan for a service request in the event that one is not stored in the service plan library.

Input Data:	Plan Request
Output Data:	Functional Service Plan
Other Elements:	Service Repository

Figure 30. Generate Functional Plan Data Information

Figure 30 summarizes the data elements manipulated by the Generate Functional Plan submodule. The first step for this submodule is to find a path using the format groups found in the plan request and calling the Perform Service Advertisement Search. If a path is not found an error is returned as no plan can be generated for this service request. Otherwise, the service advertisement protocols are then used for a second search using the Perform Service Protocol Search. If a path is found, it is this path that represents the functional plan. For a more detailed account of the algorithm, the reader is directed to [Ackley 91].

3.4.4.3 Perform Service Advertisement Search

The purpose of the Perform service Advertisement Search submodule is to search the service advertisements for a path whose success determines the feasibility of a request and to scope the plan search.

Input Data:	Plan Request
Output Data:	Functional Service Plan
Other Elements:	Service Repository

Figure 31. Perform Service Advertisement Data Information

Figure 31 summarizes the data elements manipulated by the Perform Service Advertisement submodule. This submodule tries to search the service advertisements to find out if a path can be found between two format groups. The format groups are part of the Plan Request received by the

submodule. This is the first level of path searching which determines the feasibility of finding an executable plan.

3.4.4.4 Perform Service Protocol Search

The purpose of the Perform Service Protocol Search submodule is to perform a second level search at the service protocol level. This search is performed only when a search of the service advertisements between format groups has been successful.

Input Data:	Plan Request
Output Data:	Functional Service Plan
Other Elements:	Service Repository

Figure 32. Perform Service Protocol Search Data Information

Figure 32 summarizes the data elements manipulated by the Perform Service Protocol Search submodule. While the Perform Service Advertisement Search generates a high level plan path, this submodule executes a more detailed search to begin final construction of a functional plan. It is possible that, though a service advertisement search was successful, a service protocol search could still fail.

3.4.4.5 Generate Executable Plan

The purpose of the Generate Executable Plan submodule is to take the functional plan and the service request and create a sequence of steps which can be executed. This process involves replacing the functional plan steps with detailed specifications of the functions and their arguments that must be executed to provided the requested service.

Input Data:	Functional Plan Service Request
Output Data:	Executable Service Plan
Other Elements:	Service Repository

Figure 33. Generate Executable Plan Data Information

Figure 33 summarizes the data elements manipulated by the Generate Executable Plan submodule. This submodule generates the multi-step executable plan from the functional plan and the service request. The executable plan contains all of the services and command arguments that

must be provided to the service utility to perform the steps in the executable plan.

3.4.5 Service Registration Manager

For the Integration Services Planner to have the ability to search on the service advertisements and protocols and to generate executable plans, the Planner must have some knowledge of the existing services. The Service Registration Manager module provides the platform functionality to register service definitions. This module is composed of the following five submodules:

1. Accept System Request,
2. File Service Plan,
3. Add Service,
4. Delete Service, and
5. Invalidate Plan.

This module handles the administration of the services definition repository. Services and Plans are added, deleted, invalidated, and stored through the use of the submodules contained in this module.

3.4.5.1 Accept System Request

The purpose of the Accept System Request submodule is to perform administrative operations on the Service Advertisement Database.

Input Data:	System Service Request
Output Data:	Confirmation Message Error Message
Other Elements:	File Service Plan Invalidate Plan Add Service Delete Service

Figure 34. Accept System Request Data Information

Figure 34 summarizes the data elements manipulated by the Accept System Request submodule. This submodule handles all System Service Requests that have to do with manipulating the Service Advertisement Database. The request is passed on the the appropriate submodule -- either

Add Service or Delete Service -- and if all goes well, a confirmation message is returned. Otherwise an error is reported.

3.4.5.2 File Service Plan

The purpose of the File Service Plan submodule is to save a generated plan in the Plan Repository for future use. Often, it will be the case that a service request is repeated. Having the plan stored will eliminate the need to regenerate a plan, saving time and resources.

Input Data:	Functional Service Request Executable Service Plan
Other Elements:	Service Plan Cache Accept Plan Request

Figure 35. File Service Plan Data Information

Figure 35 summarizes the data elements manipulated by the File Service Plan submodule. The given Functional Service Request and its associated Executable Service Plan are stored in the Service Plan Cache. No output is produced.

3.4.5.3 Add Service

The purpose of the Add Service submodule is to add a new service to the Service Advertisement Database. As new services are written, they can be added to the functionality of the Integration Platform.

Input Data:	System Service Request
Output Data:	Error Message
Other Elements:	Service Repository Service Plan Cache Integration Services Manager Accept System Request

Figure 36. Add Service Data Information

Figure 36 summarizes the data elements manipulated by the Add Service submodule. Via Functional Service Requests to the Integration Services Manager, the service indicated in the System Service Request is added to

the Service Repository. If the service already existed, it is overwritten. The Service Plan Cache is flushed so that new service plans will incorporate the capabilities of the new service.

3.4.5.4 Delete Service

The purpose of the Delete Service submodule is to delete a service from the Service Advertisement Database.

Input Data:	System Service Request
Output Data:	Error Message (no such service)
Other Elements:	Service Repository Integration Services Manager Accept System Request Invalidate Plan

Figure 37. Delete Service Data Information

Figure 37 summarizes the data elements manipulated by the Delete Service submodule. This is a two step process; first the service indicated in the System Service Request is removed from the Service Repository via a Functional Service Request to the Integration Services Manager. Second, Invalidate Plan is called so that the deleted service is not referenced.

3.4.5.5 Invalidate Plan

The purpose of the Invalidate Plan submodule is to remove references to a particular service from the Service Plan Cache.

Input Data:	Service name
Other Elements:	Service Plan Cache Accept Service Plan Delete Service

Figure 38. Invalidate Plan Data Information

Figure 38 summarizes the data elements manipulated by the Invalidate Plan submodule. Invalidate Plan searches the entire Service Plan Cache and removes any plan that references the given service. No output is produced.

3.5 Framework Processor

The Framework Processor module is responsible for providing the functionality necessary to parse development process frameworks and to use the information maintained in the framework for monitoring and controlling the software development process. The Framework Processor module consists of the following eight submodules:

1. Parse Framework Definition,
2. Perform Framework Validation,
3. Initialize Framework Definition,
4. Session Manager Interface Management,
5. Project Query Processor,
6. Process Event Notification Message,
7. Resolve Process Violation Condition, and
8. Process Project Query Message.

The first three modules are necessary for the initialization of a site specific framework. The third submodule provides functionality for visualizing the status of projects being monitored. The final four modules are used to update and query the current state of a specific project.

3.5.1 Parse Framework Definition

The Parse Framework Definition submodule is responsible for taking a framework definition as input and extracting pertinent information from that definition.

Input Data:	Framework Definition Framework Data Extraction Message
Output Data:	Extracted Data
Other Elements:	Perform Framework Validation Initialize Framework Definition

Figure 39. Parse Framework Definition Data Information

Figure 39 summarizes the data elements manipulated by the Parse Framework Definition submodule. This submodule is used mainly in support of the Perform Framework Validation and Initialize Framework Definition submodules and produces output based on data extraction messages passed from those two submodules.

3.5.2 Perform Framework Validation

The Perform Framework Validation submodule is responsible for checking a framework for errors and inconsistencies. This process is only performed when a framework is installed into the Framework Processor.

Input Data:	Framework Data
Output Data:	Error Messages
Other Elements:	Parse Framework Definition

Figure 40. Perform Framework Validation Data Information

Figure 40 summarizes the data elements manipulated by the Perform Framework Validation submodule. This module takes Framework Data as input that has been extracted by the Parse Framework Definition submodule. The Perform Framework Validation submodule then passes the framework through three levels of validation checking: syntax for adherence to framework definition syntax violations, instantiation for inconsistencies that result from initialization of the framework, and simulation for problems in the process flow description. Throughout these validation checks, the Perform Framework Validation submodule would produce error messages that would require resolution by the framework administrator.

3.5.3 Initialize Framework Definition

The Initialize Framework Definition submodule instantiates a new project that is to be monitored by the Framework Processor.

Input Data:	Project Instantiation Request
Output Data:	Confirmation Message
Other Elements:	Session Manager Interface Management Parse Framework Definition

Figure 41. Initialize Framework Definition Data Information

Figure 41 summarizes the data elements manipulated by the Initialize Framework Definition submodule. After receiving the project instantiation request, this submodule initializes the Constraint Base and Fact Base to reflect the initial or beginning state of the new project. At termination, the module returns an instantiation confirmation message.

3.5.4 Session Manager Interface Management

The Session Manager Interface Management submodule is responsible for providing a user interface to the Framework Processor. This interface will allow a user to browse the current status of a particular project and will provide a visualization of the development process in the context of the specified project.

Input Data:	User Gestures
Output Data:	Operation Results
Other Elements:	Project Query Processor

Figure 42. Session Manager Interface Management Data Information

Figure 42 summarizes the data elements manipulated by the Session Manager Interface Management submodule. This module will receive user gestures as input. These gestures will be interpreted to formulate project queries that are passed to the Project Query Processor submodule. The results of the query are received and then displayed to the user.

3.5.5 Project Query Processor

The Project Query Processor submodule is responsible for accessing the constraint and fact base maintained for a specific project to acquire status information.

Input Data:	Project Query Message
Output Data:	Query Results
Other Elements:	Session Manager Interface Management Process Project Query Message

Figure 43. Project Query Processor Data Information

Figure 43 summarizes the data elements manipulated by the Project Query Processor submodule. The module receives project query messages as

input, performs the query on the appropriate knowledge base, and returns the result to the calling module, either the Session Manager Interface Management or Process Project Query Message modules.

3.5.6 Process Event Notification Message

The Process Event Notification Message submodule is responsible for processing an event notification to update the state of a specific project. This submodule represents a portion of the Framework Processor's interface to the other components of the FPP. When an event, such as an artifact check in is performed, the Artifact Manager will send an event notification to this submodule.

Input Data:	Event Notification Message
Output Data:	Violation Condition
Other Elements:	Project Query Processor Resolve Process Violation Condition

Figure 44. Process Event Notification Message Data Information

Figure 44 summarizes the data elements manipulated by the Process Event Notification Message submodule. As mentioned, this module operates on Event Notification Messages received from external components. During processing of the message, the Process Event Notification Message module updates the constraint and fact base for the project indicated in the message. In the event that a process inconsistency occurs, the module raises a violation condition to be passed to the Resolve Process Violation Condition module.

3.5.7 Resolve Process Violation Condition

The Resolve Process Violation Condition submodule is responsible for resolving process violations.

Input Data:	Violation Condition
Output Data:	Process Violation Message
Other Elements:	Process Event Notification Message

Figure 45. Resolve Process Violation Condition Data Information

Figure 45 summarizes the data elements manipulated by the Resolve Process Violation Condition submodule. After the module receives the violation condition and the operation that resulted in the violation, the Resolve Process Violation Condition module retracts assertions made on the basis of the original event notification. After restoring the state of the project, a violation message is generated and returned to the component that sent the original event notification.

3.5.8 Process Project Query Message

The Process Project Query Message submodule is responsible for processing project queries from other components of the FPP. This module is distinct from the Project Query Processor module in that Process Project Query Message module must perform certain user access authorization steps before the project query message can be submitted to the Project Query Processor.

Input Data:	Remote Project Query Message
Output Data:	Project Query Results
Other Elements:	Project Query Processor

Figure 46. Process Project Query Message Data Information

Figure 46 summarizes the data elements manipulated by the Process Project Query Message submodule. A remote query message is received from an external component through the Framework Processor Platform Interface Manager. After establishing the project to be queried against and the access authorization for the user making the request, this module can submit a query to the Project Query Processor in a manner similar to the Session Manager Interface Management module. The results of the query are then passed back through the Framework Processor Platform Interface Manager.

3.6 Facilitator

The Facilitator module is responsible for routing service messages to either the local operating system or to a remote host through the Network Transaction Manager. The Facilitator consists of the following four submodules:

1. Accept Executable Service Plan Message,
2. Route Operation Results,
3. Make Network Operation Call, and
4. Make Operation Call.

3.6.1 Accept Executable Service Plan Message

This unit accepts an operation message and routes the message to either the Network Manager or the Services component.

Input Data:	Executable Service Request
Output Data:	Routed Service Request
Other Elements:	Network Manager Services

Figure 47. Accept Executable Service Plan Data Information

Figure 47 summarizes the data elements manipulated by the Accept Executable Service Plan module. This submodule takes the Executable Service Plan and sends the single step operations to the appropriate component (i.e. the Make Network Operation Call or the Make Data Operation Call).

3.6.2 Route Operation Results

This Route Operation Results module routes the results of an operation to the originator of the service request.

Input Data:	Service Results
Output Data:	Returned Results
Other Elements:	Integration Services Manager Network Transaction Manager

Figure 48. Route Operation Results Data Information

Figure 48 summarizes the data elements manipulated by the Route Operation Results module. As an operation message can be received from either the Data Services Manager or from a remote host (i.e., through the Network Transaction Manager), the Route Operation Results unit ensures that the results are sent to the correct calling unit.

3.6.3 Make Network Service Operation Call

The purpose of the Make Network Service Operation Call submodule is to send the service request to the Network Transaction Manager.

Input Data:	Executable Service Step
Output Data:	Network Message
Other Elements:	Network Manager

Figure 49. Make Network Operation Call Data Information

Figure 49 summarizes the data elements manipulated by the Make Network Operation Call component. This submodule takes the single service step which is to be sent over the network and places the network destination information around it. It then sends the network message to the Network Transaction Manager.

3.6.4 Make Service Operation Call

The purpose of the Make Service Operation Call submodule is to execute a single data operation step by calling the service that can satisfy the request step in the service plan.

Input Data:	Command Line
Output Data:	Results of Operation
Other Elements:	Data Services

Figure 50. Make Service Operation Call Data Information

Figure 50 summarizes the data elements manipulated by the Make Service Operation Call submodule. This submodule takes the executable command line that it is given and has the operating system load and run the service.

3.7 Network Transaction Manager

The Network Transaction Manager is responsible for transmitting messages from the local host to remote hosts as well as receiving messages from remote hosts that have been sent to the local host. The Network Transaction Manager module consists of the following three submodules:

1. Send Network Message,
2. Receive Network Message, and
3. Monitor Network Traffic.

3.7.1 Send Network Message

The purpose of the Send Network Message component is to encapsulate the executable service request received from the Facilitator with the network

specific information for routing the message to its destination host. Then, it must send the network message over the network to be received by the Network Transaction Manager at the destination host end.

Input Data:	Network Service Request Network Service Results
Output Data:	Network Specific Message
Other Elements:	Local Area Network (LAN) Facilitator Host Table

Figure 51. Send Network Message Data Information

Figure 51 summarizes the data elements manipulated by the Send Network Message submodule. When the Facilitator decides that the service must be made across the network, it sends the Network Manager a Network Service Request. (See Section Make Network Operation Call above.) The Send Network Message unit takes this data and wraps the network specific information around it for a network transmission to the destination Network Manager. It then sends the message across the network. The procedure is the same if the Facilitator is returning results of a service request. It does not matter to the Network Manager what kind of message is to be sent (e.g. Network Service Request or Network Service Results).

3.7.2 Receive Network Message

The Receive Network Message submodule is the complement to the Send Network Message. It receives a Network Specific Message and converts it back to a Network Service Request or Network Service Result which is then sent to the Facilitator running on the local host.

Figure 52 summarizes the data elements manipulated by the Receive Network Message submodule. This unit takes incoming messages from the Monitor Network Traffic submodule. When a message is received that is directed to the local host, the Receive Network Message unit collects the message from the network and removes the network specific protocol data from it. Then, it sends the request or result, as the case may be, to the local Facilitator.

Input Data:	Network Specific Message
Output Data:	Network Service Request Network Service Result
Other Elements:	Monitor Network Traffic LAN Facilitator Host Table

Figure 52. Receive Network Message Data Information

3.7.3 Monitor Network Traffic

The purpose of the Monitor Network Traffic submodule is to watch the network for messages addressed to the local host. When one is detected the Monitor Network Traffic unit captures it and passes it on to the Receive Network Message submodule. In addition the Monitor Network Traffic submodule logs the incoming messages so that they can be tracked.

Input Data:	Network Specific Message
Output Data:	Network Specific Message Network Log Entry
Other Elements:	Receive Network Message Network Log LAN Host Table

Figure 53. Monitor Network Traffic Data Information

Figure 53 summarizes the data elements manipulated by the Monitor Network Traffic submodule. This unit has two functions: 1) monitor the network for messages and 2) log incoming messages to the Network Log. When a message addressed to the local host is detected on the network the Monitor Network Traffic submodule must collect the data. Also, this submodule must make an entry in the Network Log so that network traffic may be tracked.

3.8 Services

The Services component of the FPP consists of many services which may be available from various sources (e.g., the operating system, translation utilities, database managers, or legacy systems which may be used for a service). Any computer system running the FPP should have the ability to invoke a service utility from the operating system. At this point, most of the work necessary to turn a request into a result has already been done. The form of the request is now in the native language of the service with the program execution syntax correctly specified and command line arguments inserted into position by the Service Plan Executor. The service in question may be the operating system, a utility program, a database manager, or a user defined program. It would be very difficult to exhaustively list every submodule (or service) that is called by the Services module even if they were all known *a priori*, so the following sections are given as a template for the numerous services which will be available from the Services component. Each service must support the following minimum set of functions. Other functionality supported by the service would depend entirely on the service.

3.8.1 Accept Command Line Arguments

The Accept Command Line Arguments unit is responsible for decoding the command line arguments into an internal data structure to be used by the service.

Input Data:	Command Line
Output Data:	Internal Service Data Structure
Other Elements:	Facilitator

Figure 54. Accept Command Line Arguments Data Information

Figure 54 summarizes the data elements manipulated by the Accept Command Line Arguments submodule. This submodule parses the command line for the arguments and sends this information to the Internal Processing portion of the service utility in the form it requires.

3.8.2 Return Result

The Return Result unit is responsible for sending output from a service to the Facilitator module. The envisioned technique for this is through the standard output for the host computer or possibly a data file.

Input Data:	(depends on service)
Output Data:	Result
Other Elements:	Facilitator

Figure 55. Return Result Data Information

Figure 55 summarizes the data elements manipulated by the Return Result submodule. Depending on the service in question, this submodule sends the result of the processing back to the Facilitator, either through the standard output or in a data file containing the results of the service.

3.8.3 Internal Processing

The Internal Processing unit carries out the requested service.

Input Data:	(depends on service)
Output Data:	Result
Other Elements:	(depends on service)

Figure 56. Internal Processing Data Information

Figure 56 summarizes the data elements manipulated by the Internal Processing submodule. Further description would depend on the individual service.

4 FPP Data Structures and Data Collections

Throughout the discussion of the design of the FPP, reference was made to messages and objects being passed between the various modules. In addition, access to various data collections and databases was described. In this section, a more detailed discussion of the data objects and persistent collections of data that are being manipulated and accessed by the components of the FPP is provided.

4.1 FPP Data Structures

The Data Structures manipulated by the FPP represent objects that allow various modules that make up the FPP to communicate and work together. The following subsections will provide a brief descriptions of the data objects necessary for the FPP to function.

4.1.1 User Data Structure

This data element is used in the User Session Interface to encode gestures made by users of the FPP. This gesture object will be accessed to produce a corresponding Function or Artifact Service Request that will be transmitted to either the Artifact Manager or the Integration Services Manager.

4.1.2 Application Message

The Application Message object is used by external applications to make a service request of the FPP. This object is passed to the Platform Interface for processing and is also used to capture the results of the requested service. The syntax for specifying the contents of this message object is described in Appendices D and E of this document.

4.1.3 Artifact Object

Artifact Objects contain information that is stored by the Artifact Manager about the artifacts that it manages. These objects are used internally by the Artifact Manager to determine the validity of certain artifact operations, to maintain versioning information of artifacts, and to maintain location information about the artifacts.

4.1.4 Confirmation Message

This data element is used throughout the FPP to indicate to the calling unit that the operation was successful. The format for the message is identical to the Results message, making the confirmation message a special case. The syntax for a results message is given in Appendix E.

4.1.5 Error Messages

Error Messages are also used throughout the FPP to indicate to the calling unit that an operation was unsuccessful. Like the confirmation message, this data element has the format of a results message.

4.1.6 Service Plan Structures

The following subsections describe the structures manipulated by the Integration Mechanism to produce and manipulate functional and executable service plans.

4.1.6.1 Plan Status Table

The Plan Status Table is a data element processed by the units of the Services Manager to document the progress of a service request. As a service request moves from stage to stage (i.e., from planning to execution), the table entry for that service is updated to reflect each stage. With this information the Integration Services Manager will be able to respond to queries about the status of service requests.

4.1.6.2 Executable Service Plan

An Executable Service Plan is a plan that can be executed by the Service Executor. This plan has been checked and will produce the proper result in answer to the request made. In addition, the services are available to perform the plan. This data element is used by the Integration Services Planner, the Integration Services Manager, the Facilitator, and the Services Executor.

4.1.6.3 Functional Service Plan

The Functional Service Plan is used only within the Integrations Services Planner during plan generation to narrow the search before finding the Executable Service Plan. A Functional Service Plan is a plan in which the path is found using only the format groups of the services involved. It is possible that an executable plan still does not exist.

4.1.6.4 Service Plan Cache

The Service Plan Cache contains the pre-generated plans which may be used if a similar service request is made. This prevents duplicating the generation step of the Integration Services Planner. The Service Plan Cache is used by the Integration Services Planner and the Services Registration Manager.

4.1.6.5 Executable Service Step

An Executable Service Step is a single atomic request within a multi-step service plan. This executable service step will represent a utility and arguments to that utility that, when executed, will fulfill part of a service plan.

4.1.7 Service Operation Structures

4.1.7.1 Service Request

This message is used by the internal components of the FPP to initiate a service request in the system. This message object is produced by the Application Interface from the Application Message received from an external application. The syntax for these message objects is described in Appendices B and D.

4.1.7.2 Service Results

This message object is used by the internal components of the FPP to transmit results of services back to the Platform Interface. The syntax for this message is described in Appendix E.

4.1.8 Network Operation Structures

This section describes the structures that will be necessary to perform network operations. These structures will be manipulated mainly by the Network Transaction Manager.

4.1.8.1 Network Message

A Network Message is used to initiate a network operation. This type of message is generated by the Facilitator and passed to the Network Transaction Manager. The contents of the message represent a functional service that must be executed on a remote host. The format that Network Messages must take is described in Appendix F.

4.1.8.2 Network Specific Message

When the Network Transaction Manager receives a Network Message from the Facilitator, the Network Transaction Manager must transform the Network Message into a format that can be transmitted on the network being used by the FPP. The structure that this Network Specific Message must take is described in Appendix F.

4.2 Data Collections

The previous section described objects that will be dynamically allocated and passed between components of the FPP. In this section, a description of the more persistent data objects that are required for the proper operation of the FPP are described. It should be noted that the data pertinent to the Framework Processor is maintained separately and is not described in this section.

4.2.1 Platform Interface Active Service Request Log

The Platform Interface Active Service Request Log is manipulated by the units of the Platform Interface to maintain information about active service requests being processed by the components of the FPP. In the event that certain components are not currently accessible or that a step in the service execution fails, the active request log can be used to restart the service execution at the point of failure.

4.2.2 Network Log

The Network Log data element is used by the Network Manager to monitor the traffic across the local area network. Each time a message is sent over the network, the Network Log is updated by adding an entry. This log is useful for linking message responses with the calling message as well as assisting the FPP in failure recovery (i.e. when the network dies and a message is lost).

4.2.3 Access Policy Database

The Access Policy Database contains the information necessary to prevent the unauthorized access to artifacts managed by the Artifact Manager. A user of the FPP must give their user id, password, and user role information so that a check can be made against this policy database for verification. In environment without access to a development process framework, this Access Policy Database should be maintained by the Platform Interface. However, within the FPP this information will be maintained by the Framework Processor. So, where the Platform Interface would normally access its own internal access policy database, the Platform Interface could instead query the Framework Processor for the access authorization.

4.2.4 Artifact Repository

The Artifact Repository is the term given to the collection of artifacts managed by the FPP. It includes the information about the artifacts as well as the actual artifacts themselves. The principle component that handles artifacts is the Artifact Manager.

4.2.5 Service Repository

This data storage element contains the information required to search for a service plan and execute an executable plan. Each service must be registered with the Service Repository by the Service Registration Manager before it can be considered for inclusion in a service plan. The components which reference this data element are the Integration Services Manager, the Integration Services Planner and the Services Registration Manager.

4.2.6 Plan Repository

The Plan Repository maintains information about the service contracts and the pre-stored plans that may be used to service a request. This data will be manipulated by the Service Registration Manager and accessed by the Integration Services Planner

4.2.7 Host Definitions

This data file maintains information about the hosts that currently run as part of the configured FPP. As a minimum, the host information will include:

1. the Host Logical Name,
2. the Host Network Address, and
3. Supported Services.

5 Status and Future Directions

This document represents the final installation in a series of documents detailing the design of the Framework Programmable Platform. As a result, the design stage of the FPP system can be considered complete. The next step in the FPP development will be to begin implementation of the designs represented in these documents. During this process, any design flaws will hopefully be discovered and the designs updated to reflect the corrections.

The initial focus of the FPP implementation will center on the Framework Processor. This focus is justified as several projects currently exist that address the issue of integration platforms. However, the characteristics of the Framework Processor represent new concepts that have not received much attention in the past. In an effort to advance this framework technology as far as possible, the Framework Processor will receive the most attention.

One other aspect of the FPP project that will be addressed concurrently with the implementation of the Framework Processor will be the development of a demonstration framework for use in demonstrating an operational FPP. This framework definition process will be a complex procedure as the entire software development process must be analyzed and modeled. However, this task will be important as the framework will provide the opportunity to fully test the Framework Processor that will be developed over the next few months.

While the framework technology will be the center of attention over the next few months, it should be mentioned again that the FPP architecture is a derivative of the DKMS system currently being developed by KBSI. Ongoing development of the DKMS architecture will allow us to get ahead in developing the FPP architecture. DKMS development will also provide us with an integration platform with which the Framework Processor can test its Platform Interface.

6 References and Related Papers

[Ackley 91] Ackley, K. A., *A Knowledge Based Planner for Intelligent Data Integration within an Integrated Services Mechanism Architecture,* Thesis Project, Department of Industrial Engineering, Texas A&M University, August, 1991.

[Aho 86] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[CFI 91] *Tool Encapsulation Specification*, CAD Framework Initiative, April 17, 1991.

[DKMS 90] *A Design Knowledge Management System (DKMS)*, SBIR Phase I Final Report, April 1990, Knowledge Based Systems, Incorporated. Contract F41622-89-C-1018, AFHRL, WPAFB.

[DKMS 91] *Software Design Document for the Integration Platform of the Design Knowledge Management System*, KBSI-DKMS-90-TR-01-1191-04, Volume 1, Revision 1, AL/HRGA, Wright-Patterson AFB, November, 1991.

[EIS 86] *The Department of Defense Requirements for Engineering Information Systems: Volume 1 - Operational Concepts; Volume 2 - Requirements.* J.L. Linn, R.I. Winner, editors, EIS Requirements Team, The Institute for Defense Analyses, Alexandria, Virginia, 1986.

[EIS 89] *Engineering Information Systems: Volume 1 - Organization and Concepts; Volume 2 - Specifications and Guidelines.* Honeywell Systems and Research Center, Minneapolis, MN, October, 1989.

[FPP 90a] *Framework Programmable Platform for the Advanced Software Development Workstation: Concept of Operations Document.* Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc under subcontract SE.37, NCC9-16. September, 1990.

[FPP 90b] *Framework Programmable Platform for the Advanced Software Development Workstation: Requirements Document.* Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc. under subcontract SE.37, NCC9-16. November, 1990.

[FPP 91a] *Framework Programmable Platform for the Advanced Software Development Workstation: Integration Mechanism Design Document.* Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc. under subcontract SE.37, NCC9-16. June, 1991.

[FPP 91b] *Framework Programmable Platform for the Advanced Software Development Workstation: Framework Processor Design Document*. Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc. under subcontract SE.37, NCC9-16. September, 1991.

[I^2S^2 85] Judson, D.L., Integrated Information Support Systems, 1986; *Integrated Information Support System (IISS): An Evolutionary Approach to Integration, Manufacturing Technology Division*, Materials Laboratory, Air Force Wright Aeronautical Laboratories, 1985.

[IDS 89] *Integrated Design Support System (IDS) AFHRL-TR-89-6: Volume I - Executive Overview; Volume II - IDS Introduction and Summary; Volume III - IDS Requirements; Volume IV - IDS Task Results; Volume V - IDS Software Documentation*. AFHRL, WPAFB, December 1989.

Appendix A Lexical and Grammar Conventions

Lexical Conventions

This section describes the lexical conventions used in the definition of the specifications found in the following appendices. Where necessary a regular definition [Aho 86] has been provided to explicitly and unambiguously express a lexical item. The lexical conventions are:

1. A semicolon (';') starts a comment and the comment is terminated by the end of the line.
2. Spaces (' ') between tokens are optional. However, keywords must be surrounded by spaces and newlines.
3. An identifier is made up of a letter followed by letters, digits, or underscores. The regular definition form of an identifier is as follows:

   letter ::= [a-zA-Z]

   digit ::= [0-9]

   identifier ::= *letter* (*letter* | *digit* | _)*
4. An integer is composed of optionally a plus or minus sign followed by at least one digit. The integer regular definition is as follows:

   digits ::= *digit digit**

   integer ::= (+ | - | ε) *digits*
5. A real number may be represented either in decimal notation or scientific notation. Therefore, a real number is represented by the following regular definition:

   fraction ::= . *digits* | ε

   optional-exponent ::= ((E | e) (+ | - | ε) *digits*) | ε

   real ::= (+ | - | ε) *digits fraction optional-exponent*
6. A string is delimited by double quotes ("") containing any printable ASCII character.

Grammar Conventions

Shown below are the conventions for the grammar of the specifications. The grammar is specified by listing its productions, with the productions for the start symbol listed first.

1. *non-terminal* - Non-terminals symbols are represented in italics.
2. **terminal** - Terminal symbols are represented in bold. They represent keywords in the language. The parenthesis contained in the grammar are part of the specification. They are considered to be terminal symbols. However they will not be in bold.
3. An expression is made up of terminals, non-terminals, and other complex expression built from rules 4 through 7.
4. { *expression* | *expression* | *expression* } - The vertical bar ('|') represents a selection of one and only one item from the set of alternatives.
5. { *expression* }? - A question mark ('?') indicates that the expression can occur zero or one times.
6. {*expression* }+ - A plus sign ('+') indicates that the expression can occur one or more times.
7. {*expression* }* - An asterisk ('*') indicates that the expression can occur zero or more times.

Appendix B Data Query Language Specification

This appendix describes the language used to perform database operations.

The lexical and grammatical conventions for this grammar are identical to those given in Appendix A.

database-command ::= *query-command* |
create-command |
change-command

query-command ::= (**select** *entity-name* { *where-clause* }) |
(**select*** *entity-name* { *where-clause* }

entity-name ::= **id**

attribute-name ::= **id**

where-clause ::= **:where** *criterion*

criterion ::= (*rev-op attribute-spec attribute-spec*) |
(*equality-op attribute-spec attribute-spec*) |
(*non-rev-op attribute-spec attribute-spec*)

attribute-spec ::= (*entity-name attribute-name* **id**) |
(**:any** (*entity-name attribute-name* **id**) |
constant

rev-op ::= *equality-op* | **<** | **>** | **>=** | **<=** |
string< | **string>** | **string<=** | **string>=**

equality-op ::= **eq** | **eql** | **equal** | **=** | **string=**

non-rev-op ::= **string-search** | **substring**

constant ::= *string* | *number* | *symbol*

create-command ::= (**defschema** *schema-name*
list-of-entity-names) |
(**defentity** *entity-name*
list-of-attribute-descriptions) |
(**make-database** *schema-name*)

schema-name ::= id

list-of-entity-names ::= (*entity-names*)

entity-names ::= *entity-name* | *entity-name entity-names*

list-of-attribute-descriptions ::= (*attribute-clauses*)

attribute-clauses ::= *attribute-clause* |
attribute-clause attribute-clauses

attribute-clause ::= (*attribute-name type attribute-options*)

attribute-options ::= {**:no-nulls** *boolean*}?
{**:index** *boolean*}?
{**:unique** *boolean*}?
{**:documentation** *string*}?

change-command ::= (**put-attribute** *entity-name value*) |
(**get-attribute** *entity-name*)

Appendix C Service External Representation Language Grammar

This appendix contains the complete lexical and grammar specification for the service contract argument specification, the service contract data specification, the service contract invocation structure, the utility environment specification, and the termination codes specification. These specifications are used for both the knowledge store representation and utility registration representation. All of these specifications are derived from the CAD Framework Initiative (CFI) tool abstraction specification [CFI 91].

The lexical and grammatical conventions for this grammar are identical to those given in Appendix A.

Notes:

1. The identifier in an argument abstraction is unique for the given argument specification.
2. The expression (**value** *identifier*) returns the string value for a given argument.

utility-specification ::=
(**utility**
(**pretty_name** *string*)
(**name** *string*)
(**version** *string*)
(**host** *string*)
(**location** *string*)
(**environment** *environment-specification**)
(**termination** *termination-specification**)
(**arguments** *argument-specification**)
(**services** *service-specification**))

environment-specification ::=
(**env** *string string-value*)

termination-specification ::=
(**code** *integer*
{ **success** | **warning** | **error** }
{ (**label** *string*) }?)

argument-specification ::=
arg-boolean-decl |
arg-choice-decl |
arg-integer-decl |
arg-real-decl |
arg-string-decl

service-specification ::=
(**protocol**
(**source** *format-specification*)
(**destination** *format-specification*)
(**contract**
{ (**rate** *number-value*) }?
(**query** *query-specification**)
(**invocation** *identifier**)
{ (**manual** *string*) }?
{ (**description** *string*) }?))

format-specification ::=
(**format** *string string string*)

query-specification ::=
(*identifier*
{ **source** |
destination |
user |
default |
framework |
value-specification })

value-specification ::=
value { **true** |
false |
real |
integer |
string |
(**choice** integer) }+

arg-boolean-decl ::=
(**arg_boolean**
identifier
true-rewrite-rule?
false-rewrite-rule?
{ (**default** { **true** | **false** }) }?
constraint-decl?
{ (**label** *string*) }?
{ (**description** *string*) }?)

arg-choice-decl :: =
(**arg_choice**
identifier
choice-decl choice-decl+
repeat-decl?
constraint-decl?
{ (**label** *string*) }?
{ (**description** *string*) }?)

arg-integer-decl ::=
(**arg_integer**
identifier
condition-decl?
{ (**default** *integer*) }?
{ (**format** { **decimal** | **octal** | **hex** }) }?
{ (**range** *range-decl*) }?
{ (**step** *integer*) }?
repeat-decl?
constraint-decl?
{ (**label** *string*) }?
{ (**description** *string*) }?)

arg-real-decl ::=
(**arg_real**
identifier
condition-decl?
{ (**default** *real*) }?
{ (**format scientific**) }?
{ (**range** *range-decl*) }?
repeat-decl?
constraint-decl?
{ (**label** *string*) }?
{ (**description** *string*) }?)

arg-string-decl ::=
(**arg_string**
identifier
condition-decl?
{ (**default** *string*) }?
{ (**format** { **to_upper** | **to_lower** }) }?
{ (**length** *integer*) }?
repeat-decl?
constraint-decl?
{ (**label** *string*) }?
{ (**description** *string*) }?)

choice-decl ::=
(**choice** *true-rewrite-rule*?
false-rewrite-rule?
(**default** { **true** | **false** })
{ (**label** *string*) }?
{ (**description** *string*) }?)

true-rewrite-rule ::=
(**if_true** *string-value*)

false-rewrite-rule ::=

(**if_false** *string-value*)

constraint-decl ::=
(**constraint** *boolean-expression*)

repeat-decl ::=
(**repeat** *range-decl delimiter-decl*)

range-decl ::=
exactly-decl |
at-most-decl |
at-least-decl |
greater-than-decl |
less-than-decl |
between-decl

exactly-decl ::=
(**exactly** *number-value*)

at-most-decl ::=
(**at_most** *number-value*)

at-least-decl ::=
(**at_least** *number-value*)

greater-than-decl ::=
(**greater_than** *number-value*)

less-than-decl ::=
(**less_than** *number-value*)

between-decl ::=
(**between** { *at-least-decl* | *greater-than-decl* }
{ *at-most-decl* | *less-than-decl* })

delimiter-decl ::=
(**delimiters** *string-value string-value string-value*)

string-value ::=
string |
condition-expression |
(**string** *identifier*) |
(**value** *identifier*) | ;; only string arguments
(**concatenate** *string-value*+)

condition-expression ::=
(**condition**
{ (**clause** *boolean-expression string-value*) }+)

boolean-expression ::=
true |
false |
(**has_value** *identifier*) |
(**value** *identifier*) | ;; only boolean arguments
(**and** *boolean-expression+*) |
(**or** *boolean-expression+*) |
(**xor** *boolean-expression+*) |
(**not** *boolean-expression*) |
(**equal** *number-value number-value*) |
(**string_equal** *string-value string-value*)

number-value ::=
integer | *real*

Appendix D Service Request Language

This appendix contains the specification for the service request language that is used to pass requests among the components of the FPP.

The lexical and grammatical conventions for this grammar are identical to those given in Appendix A.

message ::= [*requests*]

requests ::= *request* | *request requests*

request ::= (*command arguments*)

command ::= **translate** | **system** | **artifact**

arguments ::= *argument* | *argument arguments*

argument ::= *arg* | *boolean-arg*

arg ::= **id** = *constant*

boolean-arg ::= **id** | **id** = *t-or-f*

t-or-f ::= **t** | **y** | **yes** | **true** | **1** | **f** | **nil** | **n** | **no** | **false** | **0**

constant ::= **string** | **symbol** | **number** | **list**

Appendix E Service Results Language

This appendix contains the specification for the service results language that is used to pass results among the components of the FPP.

The lexical and grammatical conventions for this grammar are identical to those given in Appendix A.

result-message ::=	*status-messages* {*error-message* }? {*results* }?
status-messages ::=	*status-message* | *status-message status-messages*
status-message ::=	*number string* **#\return**
error-message ::=	*number string* **#\return**
results ::=	*number string* **#\return** *data eof-marker*
eof-marker ::=	*number* **done #\return**
data ::=	a number of bytes of data

Note: This specification is not so much a grammar as it is a message format. The BNF grammar specification language does not have the ability to count the bytes that are sent via a message. The system can support this ability, it is just that the syntax specification is not enough to specify this behavior. It must be done in the semantics of the language.

Appendix F Network Message Format Description

The underlying structure of the network messages that are used by the Network Transaction Manager is based on the OSI layered network model. The individual requests or results messages are prepended first with Network Transaction Manager information. This is used to identify such things as the originating host and Network Transaction Manager IDs. Before a message is sent over the network the Network Transaction Manager, using the network protocol installed at the site, prepends the network message with any network handling information. This includes any data that is required to route the message to the proper host. The Network Transaction Manager at the other end can now strip off the network specific data to be left with a message which can be interpreted by the new Network Transaction Manager. Figure 57 demonstrates how a typical network message is constructed and sent via the Network Transaction Manager.

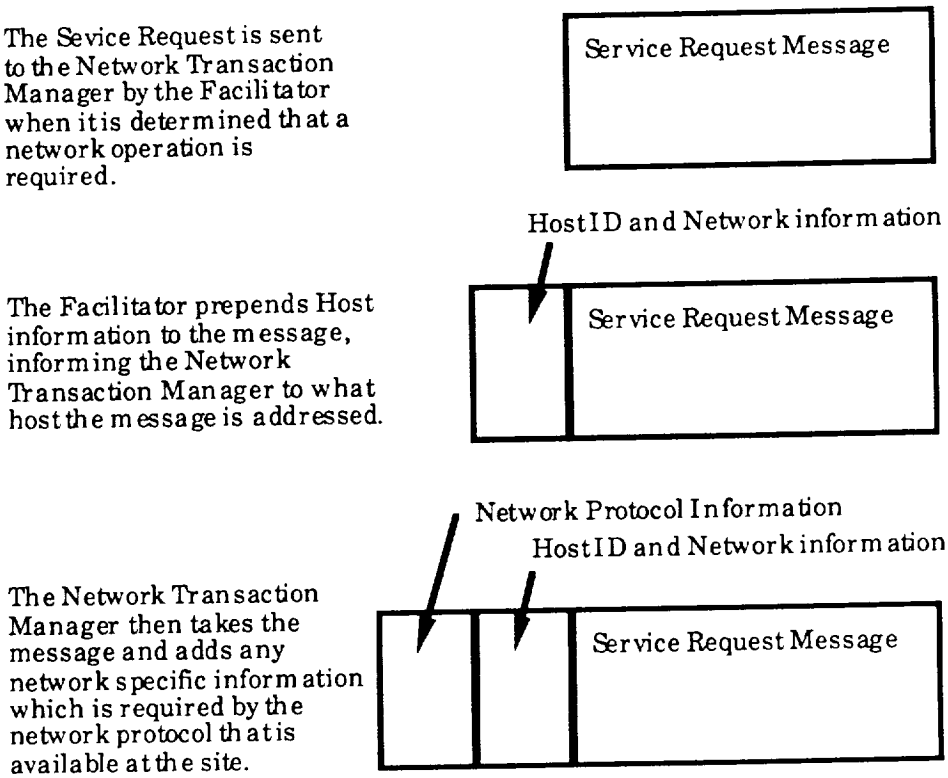

Figure 57. Network Message Format

This layering approach allows the flexibility of using different network protocols with the FPP, while keeping the Network Transaction Manager data separate from the actual service request data and the network protocol information.