Framework Programmable Platform for the Advanced Software Development Workstation

Integration Mechanism Design Document

Richard J. Mayer
Thomas M. Blinn
Paula S.D. Mayer
Uday Reddy
Keith Ackley
Mike Futrell

Knowledge Based Systems, Inc.

June 17, 1991

Cooperative Agreement NCC 9-16
Research Activity No. SE.37

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

ricis

Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Framework Programmable Platform for the Advanced Software Development Workstation

Integration Mechanism Design Document

Richard J. Mayer
Thomas M. Blinn
Paula S.D. Mayer
Uday Reddy
Keith Ackley
Mike Futrell

Knowledge Based Systems, Inc.

June 17, 1991

Cooperative Agreement NCC 9-16
Research Activity No. SE.37

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Richard J. Mayer, Thomas Blinn, Dr. Paula S.D. Mayer, Uday Reddy, Keith Ackley, and Mike 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.

Framework Programmable Platform for the Advanced Software Development Workstation

Integration Mechanism Design Document

Produced For:

Software Technology Branch
NASA Johnson Space Center
Houston, TX 77058

Produced By:

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

Dr. Paula Mayer, Thomas Blinn
Co-Principal Investigators

Under Subcontract to:

RICIS Program
University of Houston - Clear Lake
Houston, Texas 77058-1096
Subcontract Number 077:
Cooperative Agreement Number: NCC 9-16

March 18, 1991 - June 17, 1991

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

Integration Mechanism 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
Uday Reddy
Keith Ackley
Mike Futrell

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

June 17, 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:

- a realistic integration strategy that supports function and data integration of a suite of tools (distributed and covering the entire life-cycle);
- integrated access to and update of life cycle artifact data;
- control of life cycle activities and data evolution; and
- 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

Prior to the work presented in this document, work related to the FPP focused on defining how the FPP should operate at a conceptual level and then reducing those concepts to functional requirements. As the conceptual design and requirements definition have been completed ([FPP 90a], [FPP 90b]), the FPP project is now entering the design phase. At this stage, we will begin to define how operations and capabilities defined in the Concept Document will be provided by components of the FPP.

The focus of this document is on the FPP Integration Mechanism. This mechanism is responsible for providing a realistic and flexible integration capability for the FPP. The Integration Mechanism will take a service based approach to integration where the platform focuses on advertisement, specification, and facilitation of integration services. These services are functions and utilities provided by tools running on the platform that support the sharing of data and functions between different tools. It is through this approach that we feel realistic integration can be provided.

A description of the mechanisms by which the Integration Mechanism will represent, manage, and invoke these services is the main focus of this document. Other information describing the integration services approach and the relationships between the Integration Mechanism and other FPP components are provided so that the discussion of the role of the Integration Mechanism can be fully understood.

1.3 The Nature of Design

An iterative design approach being taken as the scope of the project is broad. The iterative approach will allow the design team to examine particular aspects of the FPP while making certain assumptions about other components of the platform. As designs of components are completed and new components are examined and detailed, the previous designs will be re-examined to determine if the assumptions made during the design of that component still hold.

This document is the first design produced as part of the overall FPP design process. In light of the design process described above, this document can be considered a "living" design. This means that revisions can, and probably will, be made to this design. The changes that may occur will expand and clarify the areas where the current design is lacking. However, major revisions are not expected.

1.4 Document Organization

This document takes a top-down approach to describing the design of the Integration Mechanism. The idea is to begin with a high level view of the architecture of the overall FPP and then move lower into the more detailed definition of the Integration Mechanism. The higher level descriptions will provide the boundary and scope for the design of the Integration Mechanism presented at the lower levels.

The discussion begins in Section 2 by addressing the areas that the Integration Mechanism is trying to address. This sections presents a discussion of what an integrated system should be and then discusses the strategy to be incorporated by the FPP to provide these integration characteristics. An understanding of this strategy is necessary to follow the design of the Integration Mechanism.

Section 3 presents the design of the overall FPP. The FPP design presented is by no means complete. Instead, it describes the current architecture of the FPP and describes the functional roles of each component currently identified as part of the FPP. Understanding of the roles played by the various parts of the FPP is important to understand the design of the Integration Mechanism presented in Section 4. The Integration Mechanism design was developed under the assumption that certain functions would be provided by these other components of the FPP. Without knowing the roles of these other components, understanding the Integration Mechanism design would be difficult.

The presentation of the Integration Mechanism in Section 4 is divided into two subsections. The first subsection discusses the representation language for services. The Integration Mechanism revolves around the management and manipulation of integration services. As a result, the structure of the representation of these integration services is very important to the operation of the Integration Mechanism. The discussion of the service representation language is followed by a discussion of the current functional breakdown of the Integration Mechanism. The IM has been partitioned into four components and the role of each component is discussed.

Finally, Section 5 presents the current status of the Integration Mechanism design and points out where future work concerning the FPP project as a whole and the Integration Mechanism design as a part will be directed. Also, two appendices provide more detailed specifications of the design of the Integration Mechanism. Appendix A presents the grammar for the Service Representation Language that is introduced in Section 4.1. Appendix B presents an IDEFØ Functional Model of the Integration Mechanism design that is presented in Section 4.2.

2 Integration and Strategy

A key concept in any strategy to control the software development process centers around integration - integration of CASE tools; integration of project tasks; or integration of data artifacts. As these different "levels" of integration show, integration encompasses many different aspects of system development (tools, tasks, and artifacts). Therefore, before a discussion of the design of the Integration Mechanism is presented, a discussion of integration and the aspects of integration that the Integration Mechanism will address is provided.

2.1 Characteristics of Integration

One of the lessons learned over the past 15 years, relative to integrated information systems, is that the term "integration" itself has no canonical meaning. The development an integration strategy requires that the characteristics of integration be identified up front. The following paragraphs describe some characteristics that one would be expected of a system that was integrated.

An important note must be made at this point. Despite being labeled "Integration Mechanism", the Integration Mechanism will not entirely address each of the characteristics described below. The Integration Mechanism directly relates to those characteristics that involve interaction and data exchange among various tools running under the FPP. This does not mean that the other characteristics of integration will not be addressed by the FPP. Instead, these other aspects of integration will be present in other components of the FPP. The purpose of this section is to describe the characteristics of integration that the FPP is attempting to address and to show where the Integration Mechanism will be used to support those characteristics.

Sharing of Common Data. Perhaps the most beneficial integration characteristic, the ability to share common data is critical to the success of an integration platform. It would appear that the most important aspects of this characteristic are that data need be input only once, only stringently controlled duplication of data is allowed, and only organizational policy restrict access to the data (not technological barriers). These aspects of sharing data place requirements across the entire FPP system. The Integration Mechanism will be directly involved in providing a single data entry point by supporting the transfer and translation of data between different applications running on the platform. The Integration Mechanism will play a secondary role in the control of data duplication. Control of data duplication will be the responsibility of the data management components of the FPP. While data management components will also enforce data access control, the Integration Mechanism will adhere to access control policies based on the access privileges of the user.

Rights to Privacy. In spite of the heavy emphasis on the shareability of data and computing resources, the expectations of an integrated system include an assurance of individual rights to control access to certain premature or sensitive data. Again, the Integration Mechanism will not be responsible for the control of this private data, but the manipulation of this data will be controlled by the Integration Mechanism. The reason for this is that the Integration Mechanism has been designed to manipulate data independent of the source of that data.

Support Flow of Work. A basic assumption of an integrated system is that it possess (or be built around) an accurate model of the organization it supports. In fact, some of the earliest uses of the term "integrated system" referred to manufacturing systems where the physical organization of the facility and the design of the material handling system were designed to fit closely with the process flow of the product. Integration supports the flow of work by allowing data object or artifacts produced at one stage of the development process to be accessible to team members working on another stage of the process. Support for flow of work will be shared between the data management and the Integration Mechanism components. Data management functions will provide support for the management and control of the life cycle of these artifacts. The integration Mechanism will allow data to be accessible to various CASE tools at different stages of the development process. This capability allows data to be distributed along the development path.

Support Change Propagation. In a manner similar to the support of flow of work, integration also implies support for change propagation. By maintaining relationships between data artifacts (and having an accurate and usable internal model of change semantics), modification to one artifact can be propagated to other artifacts that are related to it. To support change propagation, the definition of the relationships between data artifacts must be provided. The Integration Mechanism will not be responsible for managing and controlling these relationships between data artifacts as this functionality is reserved for the data management components. However, the Integration Mechanism will allow for the creation of relationships between two artifacts when an integration operation is performed.

Flexibility with Respect to Change. An integration system is normally assumed to have a degree of flexibility associated with it. This characteristic of flexibility is a measure of how easily the system can be expanded, contracted or evolved to respond to new requirements. For example, a system to integrate three specific tools produced by a single vendor would not be as flexible as an integration system that should be able to integrate any number of tools from any number of vendors. The degree of flexibility an integrated system exhibits usually determines the life-span and scope of application of that system design. Support for flexibility of the integration platform will occur at two levels. Through the Framework

Processor, the platform will be programmable with a system development framework. This programming will configure the platform to effectively control and manage the development process. Changes to the framework will reconfigure the platform to reflect the changes in the development process. While this flexibility allows the process to change, it does nothing for the addition of new tools and expansion of the functions provided by the platform. It is directly through the Integration Mechanism that flexibility with respect to tools will be provided. Protocols have been established that allow tools and functions to be generically represented within the Integration Mechanism. This generality allows tools to be incorporated into the platform regardless of their vendor or external interfaces.

2.2 Integration Strategy

Now that an understanding of the type of functionality the Integration Mechanism is intended to provide, a discussion of the strategy to be used to provide this functionality would be useful. It is our thesis that the integration of an application into a software engineering platform should be viewed as an opportunity to provide greater functionality to that system by providing new services and resources. With a service based approach to information integration, an application simply advertises the services it will provide, as well as the invocation procedures for that service. In essence, the advertisements define external interfaces that allow other tools or users to take advantage of the functionality provided by the new application. The integration platform provides the required support for organizing and maintaining these interface definitions, as well as for routing the integration service requests.

The integration services approach represents a new way of looking at the integration problem. Rather than focusing on the construction of an "integrated system", the focus is on the "integration services" that the integration platform and functional applications provide. In previous approaches to integrated systems architectures, the burden was assumed to be on the platform to provide the integration support desired [EIS 86], [I^2S^2 85], [IDS 89]. This traditional approach has severe problems. One of the problems is the need to define, in advance, the comprehensive standards and to build applications that meet these standards. This presumes that an organization can foresee, (a) the integration services that are required, and (b) the relative demands for those integration services. Presuming we could solve (a) and do not know the answer to (b), the traditional integration approaches force an equivalence of integration support across all needs. This implies a massive overhaul of existing legacy systems and unjustifiable modifications by vendors of their existing systems to achieve even a minimal level of integration support.

Historical experience, trends in the software industry, and the explosive demands of organizations for information services indicate that this traditional approach just won't work. What is clearly needed is an approach where such services can be incrementally introduced as the user demand forces suppliers of the needed services to provide it. The key is the establishment of the appropriate guidelines and structures for representing and executing these services. The result is that, once the expense of setting up a service has been incurred, that service is available to all subscribers and thus the provision of services evolves in an organized fashion.

This flexibility is also important when concerned with the overall development process. Previous integration efforts have focused on a particular domain and how elements within that domain interact [EIS 89], [CFI 91]. This vertical integration, while important and useful, does not address how the various domains (project management, design, implementation, testing, documenting) interact. A problem with integrating systems from these different areas is that, generally, it is not clear when and where tools should be integrated. To develop an integration strategy supporting the interaction of these tools would require a detailed analysis of the overall process involved. However, with the integration service approach, this analysis is not required. Instead, services supporting known interactions can be provided initially and new services developed as the demand arises.

This flexibility is especially important to the Framework Programmable Platform. The premise of the FPP is that the platform can be configured based on the development practices and policies of the enterprise. As a result, the process by which available services can be used is configured by the site specific framework. The flexibility of this service approach allows the access to services to be configured when the site specific framework is installed. Another critical problem with the vertical approaches is their lack of facilities for managing the evolution of the system (product) definition within a large team development activity.

3 FPP Preliminary Design

Preliminary work on the design of the overall FPP was required before work could begin on the design of the Integration Mechanism, . This platform design was required to bound and scope the Integration Mechanism. Figure 1 shows the original, conceptual architecture for the FPP. This architecture identifies the major components of the FPP and their relationship to each other.

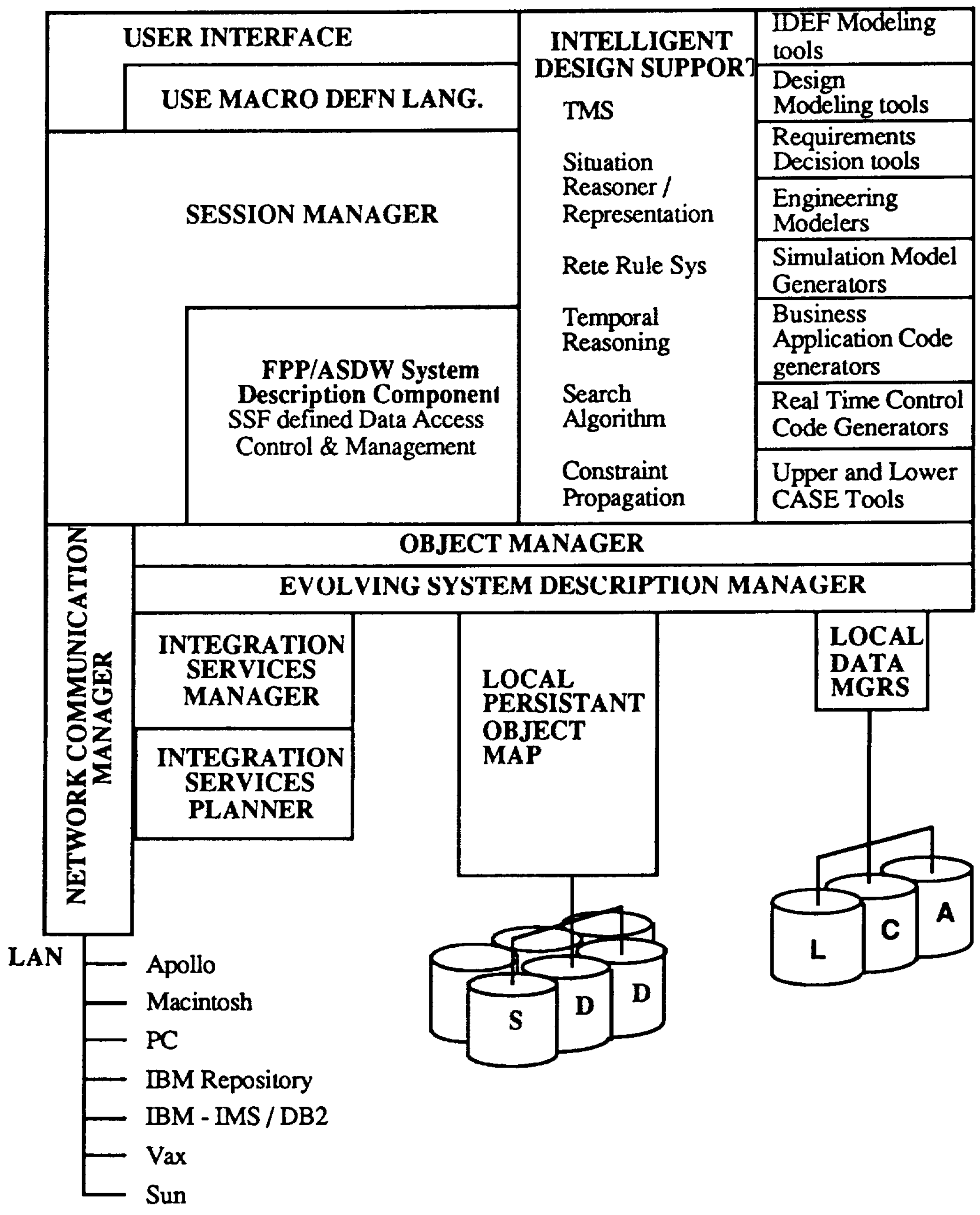

USER INTERFACE
USE MACRO DEFN LANG.
SESSION MANAGER
FPP/ASDW System Description Component
SSF defined Data Access Control & Management
INTELLIGENT DESIGN SUPPORT
TMS
Situation Reasoner / Representation
Rete Rule Sys
Temporal Reasoning
Search Algorithm
Constraint Propagation
IDEF Modeling tools
Design Modeling tools
Requirements Decision tools
Engineering Modelers
Simulation Model Generators
Business Application Code generators
Real Time Control Code Generators
Upper and Lower CASE Tools
NETWORK COMMUNICATION MANAGER
OBJECT MANAGER
EVOLVING SYSTEM DESCRIPTION MANAGER
INTEGRATION SERVICES MANAGER
INTEGRATION SERVICES PLANNER
LOCAL PERSISTANT OBJECT MAP
LOCAL DATA MGRS
S
D
D
L
C
A
LAN
Apollo
Macintosh
PC
IBM Repository
IBM - IMS / DB2
Vax
Sun

Figure 1. FPP Conceptual Architecture

The design process began with an examination of this conceptual architecture. While this diagram presents an overall view of the system operation, it is difficult to work with when attempting to formalize a design for the FPP. Closer examination of the conceptual architecture resulted in a separation of applications from underlying platform functionality. This mostly involved removing the upper-right quadrant of the diagram and examining what was left. Conceptually, this was an important distinction. This forced the design team to treat applications generically, which is especially important to the service integration strategy.

The result of this examination and separation is shown in Figure 2. The diagram represents the functional architecture where each box in the diagram represents a functional unit and the links between the boxes represent communication between those functional units. It should be noted that this architecture is a derivative of the Design Knowledge Management System platform architecture that KBSI is currently developing for the Air Force [DKMS 90]. The approach in the FPP project has been to leverage off of the DKMS effort by first adopting the integration strategy of the DKMS and then layering the framework programmability on top. A discussion of this architecture is required to set the context for how the Integration Mechanism fits in with other components of the FPP. The remainder of this section will be dedicated to this discussion.

The *FPP Session* and *Application* represent the external interfaces to the FPP. The FPP Session serves as the users direct link to the FPP environment while the Application component represents the interface through which applications would gain access to the FPP functionality. Through this module the user could browse the database, services, users, hosts, tools, etc. The user could also set up protection for groups working on data sets, etc. The user could invoke the Integration Mechanism to generate a service plan on a data set. Application programmers, can incorporate this plan in their code. Plans embedded in application programs would be checked for feasibility at the time the application is launched and when the plan is executed. Infeasible plan segments are trapped and the user is allowed to suggest an alternative action which could include replan, ignore, abort, etc.

The *Integration Mechanism* is responsible for monitoring and controlling the generation and execution of integration service plans. Conceptually, the Integration Mechanism can be partitioned into an Integration Services Manager and an Integration Services Planner (the design actually partitions the Integration Mechanism into several different components, see Section 4.2.3). The Integration Services Manager is responsible for receiving and executing integration service requests. In doing this, the ISM spawns background plan management processes for each request, monitors the progress, and collates the results of the request. Part of the process of providing a service is the generation of general service and executable plans. This task is delegated by the Integration Mechanism to the Plan Builder. During plan generation, two planning processes take place. The first is the generation of a service plan, whereby the Plan Builder determines whether advertised services exist. If this first process completes successfully, the more detailed executable plan generation process begins, where the actual machines, resources, and utilities needed to provide the service are determined.

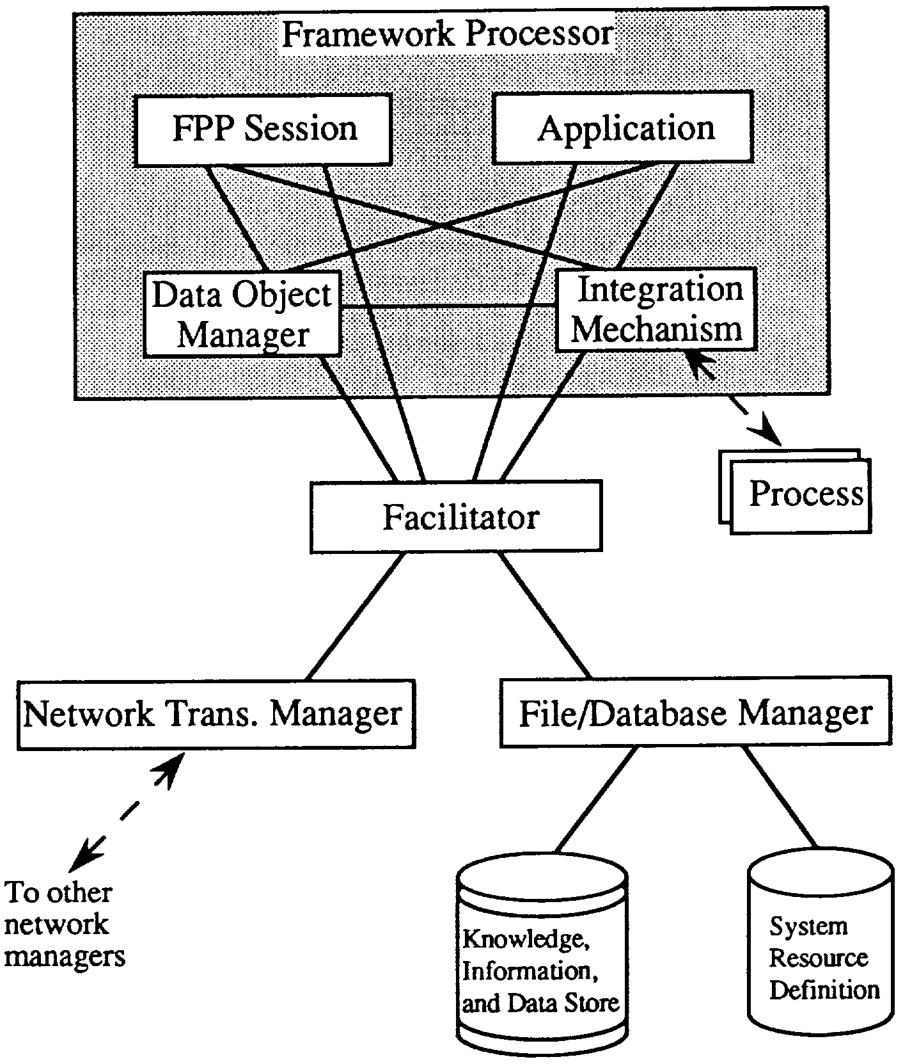


Figure 2. Current FPP Functional Architecture

The *Data Object Manager* (DOM) is responsible for the management of life cycle data artifacts. In managing life cycle artifacts, the DOM will provide functionality for registering data artifacts in the repository and to maintain access control over the artifacts. The controlled nature of these artifacts managed by the DOM allow relationships between artifacts to be maintained. One of the most important relationships to be supported by the DOM will be dependency relationships between artifacts. Other relationships include configuration and versioning relationships. To maintain consistency in the constraints and relationships defined on data artifacts, an inference engine will be required to support the DOM. This inference engine supports both truth maintenance and constraint propagation.

The DOM will also be used to manage system resource artifacts. The need to do this stems from the fact that the FPP operates in a distributed environment. The namespace for the DOM, as a result, will be distributed across all nodes of the platform. To take advantage of this single repository and eliminate the need to store system resource data on every machine, the DOM will manage these artifacts. However, to improve performance, FPP components may cache this information at system startup.

It is difficult to call the *Framework Processor* a distinct component of the FPP. While it is true that some framework specific functionality will be required to load a framework and to configure the platform based on that framework, the burden will be on the other components of the FPP to adhere to the requirements of the framework. 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. In light of this role, the framework processor would simply respond to queries from the various components about the framework contents.

The *Facilitator* serves as a dispatcher of messages between higher level components (DOM and Integration Mechanism) and the lower level components (Data Managers 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 DOM 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 distributed nature of the FPP also makes the Facilitator necessary. When accessing data, whether that data resides on the local machine or on a remote machine should be transparent to the DOM. To hide the location of the data from the DOM 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 also serves as the interface between the FPP and *File/Database Managers* running on the machine. This architecture allows the host file system and different database managers to be used for FPP data storage. The location of the data will be encapsulated in the data ID. The Facilitator will extract the location and formulate a query based on the query structure of the database manager. The query will then be passed to the appropriate manager where the data will be collected and returned to the Facilitator. The FPP components will be taking advantage of objectoriented databases to maintain knowledge about data artifacts and system resources. KBSI has initially targeted the Itasca database for the Sun workstation and the Statice database for the Symbolics workstation.

The *Network Transaction Manager* is responsible for sending and receiving network operations for 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 NTM is only accessible through the Facilitator. From the message ID, the Facilitator determines that the message requires a remote transaction. The message is passed on to the NTM where the message is encapsulated into a network packet and sent to the appropriate machine. When the remote machine receives the message, the network packet is decoded by the remote NTM, the contents are passed to the remote Facilitator, and the message is sent to the appropriate FPP component on the remote machine.

It is expected that the Network Transaction Manager will be built on top of the CRONUS distributed operating system. The advantages of using CRONUS include the fact that CRONUS is object based, making it easily integratable into our system design. CRONUS also provides a common network protocol across heterogeneous workstations (including Symbolics, Sun, and DEC) that will make the generation of portable Network Transaction Managers easier. Finally, CRONUS provides a higher level of abstraction from the low level network protocols that will allow more powerful network operations to be performed more easily.

The *Knowledge, Information and Data Stores* will store the data artifacts being maintained and controlled by the FPP. These stores will not only contain the data artifacts themselves but will also include data and knowledge necessary to manage those artifacts. The management information will include access control information, audit trail information, configuration and version control information, as well as dependency relationship information. Rules and constraints for the manipulation and management of these data artifacts will be established by the Site Specific Framework.

The *System Resources Definition* repository will contain information required by the FPP to operate. This would include knowledge about tools, applications, services, hosts, and database servers operating under the FPP. These resources would also contain information about FPP users and groups. The distribution of this information has not been established yet. It would be advantageous to have a single repository of information to eliminate inconsistency. However, central location of data can serve as a bottleneck in system performance. For one case, it is expected that the integration service information will be maintained on the machine that the service actually executes on. But since the service knowledge is maintained by the DOM, access to the service information will be available to all machines and therefore to the Integration Mechanism running on that machine.

There are basically three sources for the information contained in the System Resources. The first is the Site Specific Framework. This framework will define the user roles, methods, and system configurations that will be operable at the particular site. The second major source is the system installation procedure. When the FPP is installed, it must be configured. Part of this configuration process will be the definition of the hardware and software resources available to the FPP. Finally, the third source of information will be the maintenance activities. This will include adding new machines to the platform or adding and defining new services for the platform.

4 Integration Mechanism Design

Once the general structure and functional breakdown for the components of the FPP had been established, work on the formal design of the Integration Mechanism began. Section 2.2 describes the integration strategy to be implemented by the Integration Mechanism. Designing a system to support the integration services approach requires that a definition of what a service is and how a service can be represented be derived. Once the required knowledge has been identified and a structure for representing that knowledge has been defined, the design of a system to generate and execute plans based on those structures can proceed.

The discussion of the design of the Integration Mechanism will follow closely this two phased approach. The discussion will begin with the definition of the structure and content of the information maintained about integration services. This discussion will be followed by a description of the functional components of the Integration Mechanism and how these components will use service knowledge to support the integration services strategy.

4.1 Service Information Representation

For the Integration Mechanism to generate and execute service plans, it must have access to data about the service utilities managed by the FPP. The Integration Mechanism must be able to determine what each utility does, what arguments it requires to execute, the format for its inputs, and the format of its outputs. A structure is needed to capture this information and to support the efficient generation of service plans.

The structure that has been developed for capturing service information is displayed in the service representation schema shown in Figure 3. The schema consists of six interrelated objects: format class, format, service advertisement, service protocol, service contract, and utility. Each and every service registered with the FPP will be required to provide this information in order for the service to be accessible to users. However, this information will only have to be provided once.

It is important to note that, at present, this schema represents an internal representation of the service knowledge. The form by which this information will be presented to the FPP has not yet been established, though a language has been defined for the representation of Service Contracts (see Section 4.1 and Appendix A). It might be possible that the service registration format could simply be a form to fill in, similar to the structure shown in Figure 3. More than likely, however, automated tools for registering and entering this information will be provided. See Section 4.2.3.4 for more information about the service registration process.

Format Class

name:	string
formats:	set of <format>
key:	type

Format

name:	string
version:	string
primary class:	<format class>
filter:	predicate
can be treated as:	set of <format>

Service Advertisement

source format class:	<format class>
destination format class:	<format class>
protocols:	set of <service protocol>

Service Protocol

source format:	<format>
destination format:	<format>
contracts:	set of <service contract>

Service Contract

utility:	<utility>
argument specification:	set of argument abstractions
data specification:	set of data abstractions
invocation structure:	set of argument labels

Utility

name:	string
version:	string
resources:	hardware/software specs
host:	hostname
location:	pathname
environment specification:	set of environment specs
termination codes:	set of exit codes

Figure 3. Service Representation Schema

4.1.1 Formats and Format Classes

From an abstract point of view, the execution of a function or use of an application can simply be viewed as a transformation of information from one (input) format to another (output) format. In many situations, this view is not important. For example, Microsoft Word was used to prepare this design document. When the document is opened for editing, the input format is Microsoft Word and when the document is closed, it is still in Microsoft Word format. So, actually, no translation has taken place.

It is only when data needs to be moved across tools that this view becomes particularly important. Using a similar example, let us assume that the same Microsoft Word document needs to be included as part of a TeX document being prepared on a Unix Workstation. Since the document is currently in Word format, it is not usable as part of the TeX document. So, a possible option is to open the Word document and then save the file out in an ASCII text format that could be incorporated into TeX. In this instance, the design document has undergone a transformation from Microsoft Word format to ASCII text format.

Using this idea, a format-based service representation system has been developed for the Integration Mechanism. This representation of integration services revolves around *Formats* and *Format Classes*. A Format is a name for a specific data representation and a Format Class is simply a group of closely related Formats. For example, format classes might include IDEFØ, Postscript, IGES, and Text. However, within the IDEFØ format class, formats might include AIØ v2.3 Btrieve, IDEFINE 1.6, and Modeler 1.0. Within the Postscript format class, formats could be Adobe PS 4.3 and TGPS 1.1.

A Format Class is represented by three pieces of information: a name, a set of formats, and a key. The name simply serves as an identifier for the format class and will be used for planning purposes. The set of formats maintains the set of formats that are part of the format class. Finally, the key captures the type of object that the format represents. For instance, in our Word example above, the key would be "pathname" since the Microsoft Word format represents a file.

The representation of a Format is somewhat more complex than for a Format Class. Like a Format Class, a Format also has a name identifying the format, but, unlike the Format Class, a Format may have a version number associated with it. This version is important when different versions of the same tool are running under the FPP. The primary class slot of a Format captures the Format Class to which the format belongs. Semantically, a format may belong to only one class. However, the format may be syntactically equivalent to many other formats. This is what the "can be treated as" slot of the format structure captures. For example, an AIØ v2.3 Btrieve file is a specific representation of an IDEFØ model and its primary class is IDEFØ. However, the file is stored in a Btrieve database format, and the AIØ v2.3 Btrieve file can be used in the same utilities that can use Btrieve database files. Therefore, the AIØ v2.3 Btrieve file can be treated (syntactically) as a Btrieve database file. Finally, the "filter" slot captures any information about constraints on the format. For instance, a utility may produce output in a specific format, but may have only produced certain pieces of information. The filter would specify what data had actually be produced.

One issue that has not been completely resolved involves formats and format classes. It is not always the case that a format or a format class will have a recognizable name. The names of formats are critical for service planning as the names determine whether two services manipulate the same format and therefore can be executed in sequence. For industry standard formats like PostScript, this is not a problem. It is possible, though, that different services could use formats that have no standard name and may be equivalent. The service planning requires that some means for determining equivalence between formats is provided. With the current representation scheme, the equivalence test is simply equality of format or format class names. It remains to be seen whether more robust equivalency procedures are required for formats and format classes.

4.1.2 Service Advertisements

Service Advertisements describe sets of service protocols that have been defined between format classes. Each service advertisement serves as a bridge between two Format Classes and captures all operations that can occur between the two Format Classes. A Service Advertisement maintains three things:

1) source format class - the format class that input to services adhering to this advertisement must be a member of;
2) destination format class - the format class that output of services adhering to this advertisement must be a member of; and
3) set of protocols - a list of all service protocols that adhere to this advertisement.

When the Integration Platform attempts to generate a plan, it scans the list of service advertisements to determine if there is a path from the source format class to the destination format class. If a path is found, the Integration Mechanism then examines the service protocols (as described in the next subsection) associated with the service advertisement to determine which service to invoke.

4.1.3 Service Protocols

Service Protocols describe sets of service contracts that exist between two formats. Each protocol is a member of the set of protocols defined for a service advertisement. As a result, the protocol captures services that transform data maintained in the source format class of the parent service advertisement to data maintained in the destination format class of the parent service advertisement. A service protocol is represented by the following pieces of information.

1) Source Format: The format that input to services adhering to this protocol must be. This format will be an instance of the source format class of the parent service advertisement.

2) Destination Format: The format that output of services adhering to this protocol must be. This format will be an instance of the source format class of the parent service advertisement.
3) Set of Contracts: A list of all service contracts that adhere to this protocol.

The Integration Mechanism searches through the service protocols for a given service advertisement to find which contracts might be invoked to satisfy a service request. If it finds a service protocol that represents the desired service (eg. text to Adobe Postscript 2.1), it then examines the contracts (see the next subsection) associated with the service protocol to determine which utility to invoke to perform the needed service.

This "hierarchical" organization of Formats and Format Classes and of Service Advertisements and Services Protocols is designed to narrow the search required when attempting to generate a service plan. Planning involves searching through the integration services to determine if an operation using the appropriate formats exists. By organizing service protocols that link two formats under a more general service advertisement that links two format classes allows a two-tiered search strategy. At the first level of search, only the service advertisements are searched. It is only when an advertisement linking the two format classes has been found that the search moves to examining the service protocols. Through this approach, a large number of services can be eliminated from the search space and faster integration plans can be generated.

4.1.4 Service Contracts

While the previous structures have focused on representing services for planning purposes, the Service Contract is concerned more with capturing information necessary to actually execute a service. These contracts specify to the Integration Mechanism what information must be collected in order to invoke the utility and how to actually perform the invocation once the information is collected. These service contracts represent actual executable utilities and organize information about these utilities into four units of information: the utility, the argument specifications, the data specifications, and the invocation structure. The four units are discussed in the following subsections.

4.1.4.1 Utilities

The Utility is a service representation structure that captures hardware and software information about specific utilities that provide service to the FPP. The information required to register a utility is relatively straightforward and is enumerated below.

1) Name: The name of the utility or tool.
2) Version: The version of the utility or tool. Different versions of the same tool may exist in the environment.

3) Resources: The prerequisite hardware and software resources needed for the utility to execute properly (i.e., memory, etc...).
4) Host: The machine on which the utility resides and runs.
5) Location: The pathname of the utility on the host machine.
6) Environment Specification: These specifications define the state that the host machine must be in to execute this utility properly.
7) Termination Codes: These codes represent values that will be returned from the execution of the utility so that the Integration Mechanism can determine the context of the termination (i.e., did the utility terminate normally).

This information would be used by the Integration Mechanism to first determine if the service can be run locally or must be run remotely. Having established that, the information would be used to configure the environment so that the spawned service execution process will execute properly.

Note: The following three subsections discuss the argument specification, data specification, and invocation structure components of a Service Contract. The structures for defining this information are a modified form of the CAD Framework Initiative Tool Encapsulation Specifications [CFI 91]. The grammar for the modified language to be used by the FPP is presented in Appendix A of this document.

4.1.4.2 Argument Specification

In order for the Integration Mechanism to invoke utilities, it must understand their invocation protocol (command line syntax or program call interface) and be able to supply the appropriate arguments as needed. The argument specifications of a service contract define what, if any, arguments are required by the utility. The service contract would contain a single argument specification for each argument to the utility.

4.1.4.3 Data Specification

The data specifications define the input and output parameters for the utility along with the type of those parameters. These data parameters must correspond directly with the keys in the format class specifications for this utility. Failure to match these data specifications with the key of the format class will result in a registration error, as the two formats will be incompatible. The service contract would contain a single data specification for each input and output parameter of the utility.

4.1.4.4 Invocation Structure

When the argument specifications are defined, the arguments can be entered in any order. Nevertheless, it may be the case that the utility requires that arguments be placed in a specific order. The argument sequence for the utility is specified with this invocation structure. There will be a single invocation structure for each service contract.

Format Class: <fc1>

name:	text
formats:	(<f1>)
key:	pathname

Format Class: <fc2>

name:	postscript
formats:	(<f2>)
key:	pathname

Format: <f1>

name:	ASCII text
version:	n/a
primary class:	<fc1>
filter:	none
can be treated as:	n/a

Format: <f2>

name:	Adobe Postscript
version:	2.1
primary class:	<fc2>
filter:	none
can be treated as:	n/a

Service Advertisement: <sa1>

source format class:	text
destination format class:	postscript
protocols:	(<sp1>)

Service Protocol: <sp1>

source format:	<f1>
destination format:	<f2>
contracts:	(<sc1>)

Service Contract: <sc1>

utility:	<u1>
argument specification:	[specified elsewhere]
data specification:	[specified elsewhere]
invocation structure:	[specified elsewhere]

Utility: <u1>

name:	text2ps
version:	1.04
resources:	IBM PC compatible, MSDOS 3.0 or higher
host:	MTF/KAA PC
location:	c:\bin\text2ps.exe
environmental specification:	none
termination codes:	none

Figure 4. Service Representation for Text2ps

4.1.5 Example Service Representation: Text2ps

The representation of service information is quite complex. In light of the complexity, an example may describe the representation better than the previous discussion. This section presents portions of service representation image for a PC-based text to postscript conversion utility called text2ps. This example is intended to show how the structures described in the previous sections will appear when actual integration services are registered with the FPP.

Service Representation

Figure 4 shows the service representation image for text2ps. Text2ps takes an input format of *ASCII text*, which is part of the format class *text*. It produces output in *Adobe Postscript 2.1* format, which is part of the format class *postscript*. The advertisement for the Text to Postscript service currently states that only one service protocol has been registered for this service. That protocol, in turn, states that only one contract has been registered to provide the ASCII Text to Adobe Postscript format translation service. That contract is the text2ps utility we are describing. The text2ps, version 1.04 runs on an IBM PC compatible, resides on the MTF/KAA PC, is located a c:\bin\text2ps.exe, and has no environmental specifications or termination codes.

Argument Specification

The text2ps utility has twelve arguments that may be specified on the command line. For our purposes, we will show the argument specifications for four of these arguments. Each argument specification refers to a specific command line argument. The order in which the argument specifications appear is the order in which the user (or framework) will be queried.

Partial set of Argument Specifications for text2ps:

```
;;; the -v argument tells the utility to print version information.
;;; no other options may be specified with -v.

(arg_boolean version
        (if_true "-v")
        (if_false "")
        (default false)
        (label "show tool version"))


;;; the -? argument is used to print a list of the command
;;; arguments for the utility.

(arg_boolean listargs
        (if_true "-?")
        (if_false "")
        (default false)
        (constraint (string_equal (value version) ""))
        (label "show command arguments"))
```

```
;;; the rotation angle specifies page rotation for the output

(arg_integer rotation_angle
        (condition (clause  (not (equal rotation_angle 0))
                            (concat "-r" rotation_angle)))
        (default 0)
        (range (between (at_least 0) (at_most 90)))
        (repeat (exactly 1))
        (constraint (or     (string_equal (value version) "")
                            (string_equal (value listargs) ""))
        (label "page rotation angle (in degrees)"))


;;; the -f option specifies the font for the output.  Notice that "Times
;;; Roman"is the default value.

(arg_choice font_name
        (choice
                (if_true "-f Courier")
                (default false)
                (label "Courier"))
        (choice
                (if_true "-f Helvetica-BoldOblique")
                (default false)
                (label "Helvetica Bold Oblique"))
        . . .
        (choice
                (iftrue "-f Times-Roman")
                (default true)
                (label "Times Roman"))
        (repeat (exactly 1))
        (label "desired font")
        (constraint (or     (string_equal (value version) "")
                            (string_equal (value listargs) "")))


;;; the input file name.

(arg_string input_name
        (default "")
        (repeat (exactly 1))
        (label "input file"))


;;; the output file name.

(arg_string output_name
        (condition   (clause (string-equal (input) "")
                              "> output.ps")
                    (clause (not (string_equal (input) ""))
                              (concatenate "> " (input))))
          (default "")
          (repeat (exactly 1)
          (label "output file"))
      etc ...
```

Data Specification

This section describes the data required for the text2ps utility. This data corresponds to the keys in the format class specifications for the utility. In this case, both keys are pathnames. The first is the pathname of the input file(s), and the second is the pathname of the output file.

```
      ;;; this is the data definition for the input data file

      (datadef input_file
              (label "input file")
              (direction input))


      ;; this is the data definition for the output data file

      (datadef output_file
              (label "output file")
              (direction output)
              (existsif (not (string_equal (value output) ""))))
```

Invocation Specification for text2ps

The argument specifications determine the sequence in which the user (or framework) is queried for information. However, the utility may expect a different argument sequence. The argument sequence for the tool is specified in the invocation specification. The invocation specification for the text2ps tool (and the subset of arguments we have discussed) is as follows:

```
      (command_args
              (value version)
              (value listargs)
              (value font_name)
              (value rotation_angle)
              (value input_file)
              (value output_file))
```

This example has shown how a service would be represented to the Integration Mechanism. While the service advertisement and protocol specifications occur at a relatively abstract level, the service contract specification is very detailed and complex as the parameters and requirements of the service utility must be explicitly defined. This separation of abstract representation from utility description is required to support a two stage planning process described in Section 4.2.3.1 that makes service planning more efficient.

4.2 Integration Mechanism Design Description

With an understanding of how services are represented within the FPP environment and of why they are represented that way, it is now possible to discuss the design of the Integration Mechanism. The Integration Mechanism has been broken down into four components: the Integration Message Encoder/Decoder, the Plan Builder, the Plan Executor, and the Service Registration Tool. Each of these components will be discussed in further detail below.

4.2.1 Design Considerations

Before presenting the design of these components, however, the reader should understand some of the factors that had to be considered when producing the design of the Integration Mechanism. An understanding of these concepts will allow a better feel for why the Integration Mechanism is designed the way it is. The following paragraphs categorize the primary design considerations.

1. The working of the platform is controlled by the programmable framework. The framework influences the operation of the Integration Mechanism in many ways. Among the most significant are:

    - Access control: user and application access to data, services and tools can be defined with the framework.
    - Enforce critical paths: many project plans contain certain critical paths that define which activities must be completed before certain other can be started. This constrains the services that can be accessed.
    - Constraint propagation: inter- and intra- project constraints often are complex enough that their side effects will be implicit in other parts of the project. This again constrains what the Integration Mechanism can do for a service request.
    - Log design history: based on the requirement set in the framework, the detail of the ISM log can be controlled.

2. Archiving successful service plans for reuse eliminates the overhead of replanning each time that request is made. Logging plans that fail during execution aids in generation of an error report to debug that service. Keeping track of service requests that could not be satisfied by existing services helps identify new services that need to be designed and built.

3. Detailed monitoring of the Integration Mechanism while providing a service will not only aid in documenting development history, but also serve to log illegal accesses, identify bottlenecks in service requests, identify points of frequent failure, etc.

These considerations were a driving force in the development of the Integration Mechanism design.

4.2.2 Integration Service Plans

The discussion of the Integration Mechanism Design to be presented in the next section is driven by the process supported by the Integration Mechanism. The operation of the IM revolves around the generation, management, and execution of service plans. However, a discussion of a system that produces and manipulates service plans would be difficult to follow without some understanding of what a service plan actually is. This section will describe the concept of a service plan.

In Section 2.2, the Integration Services strategy to providing an integrated environment was presented and the Integration Mechanism has adopted this strategy. The main concept in the integration services approach is the service. A service is just a functional unit that performs a service to the user. Normally, services are intended to be some operation that integrates two or more existing CASE tools, though this is not a requirement. For example, a service might be to translate an IDEFØ model produced by the AIØ into an IDEFØ model readable by IDEFINE.

Given, that these services exist, there must be some means for telling the Integration Mechanism that the services exist. In Section 4.1, the structures by which a service can presented to the Integration Mechanism was defined. This representation alone, however, is not enough. The Integration Mechanism must have a strategy for manipulating these service representations.

This is where the "service plan" enters the picture. A service plan is simply a sequence of operations that, when performed, result in the provision of the requested service. However, a service plan can exist on two levels: the functional plan and the executable plan. The functional plan is a sequence of service identifiers that when sequenced together provide the requested service and is independent of any hardware or software implementation details. It is in the generation of the functional plan that true "planning" occurs. The object is to find a sequence of services that will provide the requested service and this process is accomplished by examining the service advertisements and protocols. It is only when a functional plan is generated successfully that an executable plan is produced. An executable plan is a sequence of machine operations derived from the service contracts that will actually be executed to provide the requested service.

These service plans, both functional and executable, are the mechanism by which the integration services strategy will be implemented. The following section on the design of the Integration Mechanism will detail exactly how these service plans are produced, managed, and executed.

4.2.3 Integration Mechanism Design

The following description details the current design of the Integration Mechanism. Operationally, every machine that is part of the FPP will have an implementation of the Integration Mechanism running. Each Integration Mechanism will have access to information about all available services and will have the ability to interpret service requests, build executable service plans based on those requests, execute the service plans, and finally collect and return the results of the service plans. Control over this process will be defined by the Integration Mechanism and the Site Specific Framework installed at the particular site.

Figure 5 shows the operation of the Integration Mechanism and the interaction of the key components of the Integration Mechanism. The key components include the service registration tool, the plan builder, the plan executor/monitor, and the message decoder/ encoder. Notice, however, that all service requests, even single step services, initially pass through the Plan Builder. Originally, the design incorporated another component that would determine if a particular service were advertised and if so would route the service request around the Plan Builder. However, after further study, this other component tended to duplicate some of the functionality of the Plan Builder (i.e., service advertisement lookup). To reduce this duplication of effort, this extra component was combined with the Plan Builder.

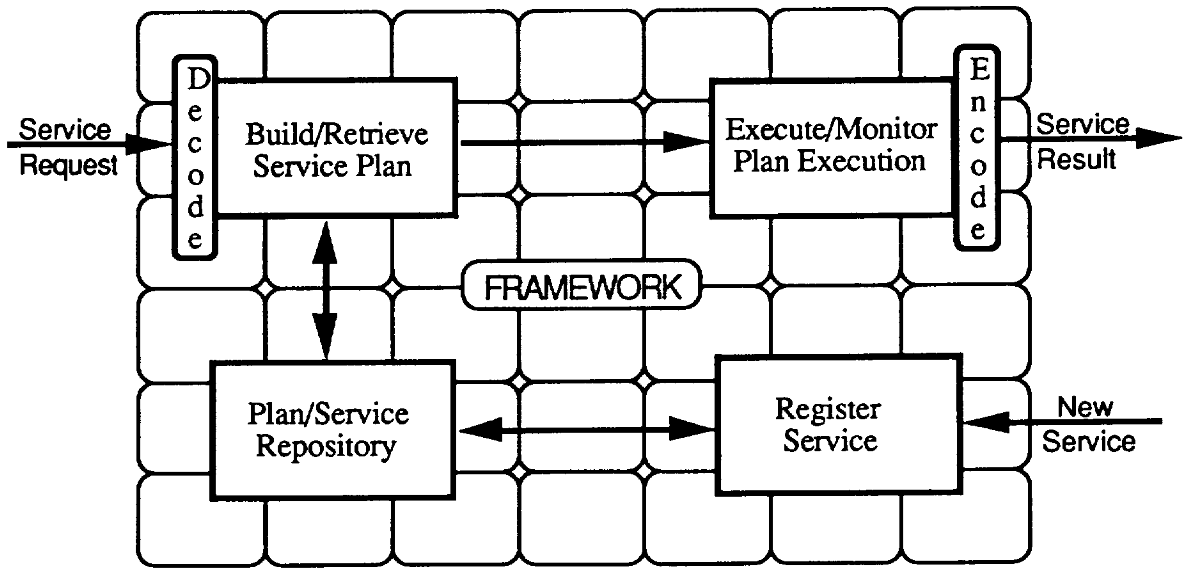


Figure 5. Operation of the Integration Mechanism

The operation of each of these components and the interaction between the components will be discussed in the following sections. Also, an IDEFØ model of the Integration Mechanism Design can be found in Appendix B of this document.

4.2.3.1 Plan Builder Component

The Plan Builder component of the Integration Mechanism is responsible for generating the functional and executable plans required to satisfy service requests received by the Integration Mechanism. Figure 6 shows the operation of the Plan Builder. When the plan builder receives a service request, it first checks to see if any planning is required. If the request is for a simple service (i.e., a service plan of only one step) and that service is advertised, the service request is simply passed on to the executable plan generator.

If, however, the requested service is not advertised, the Plan Builder will attempt to devise a plan to support this service through a two step process. First, service plan databases are searched to see if a functional plan for this service had been generated previously. This step implies that a library of generated plans is maintained by the Plan Builder and, in fact, this will be the case. Every successful plan generated by the Plan Builder will be "checked in" to this library. The storage of these service plans will reduce the requirements for planning when the same service is requested many times.

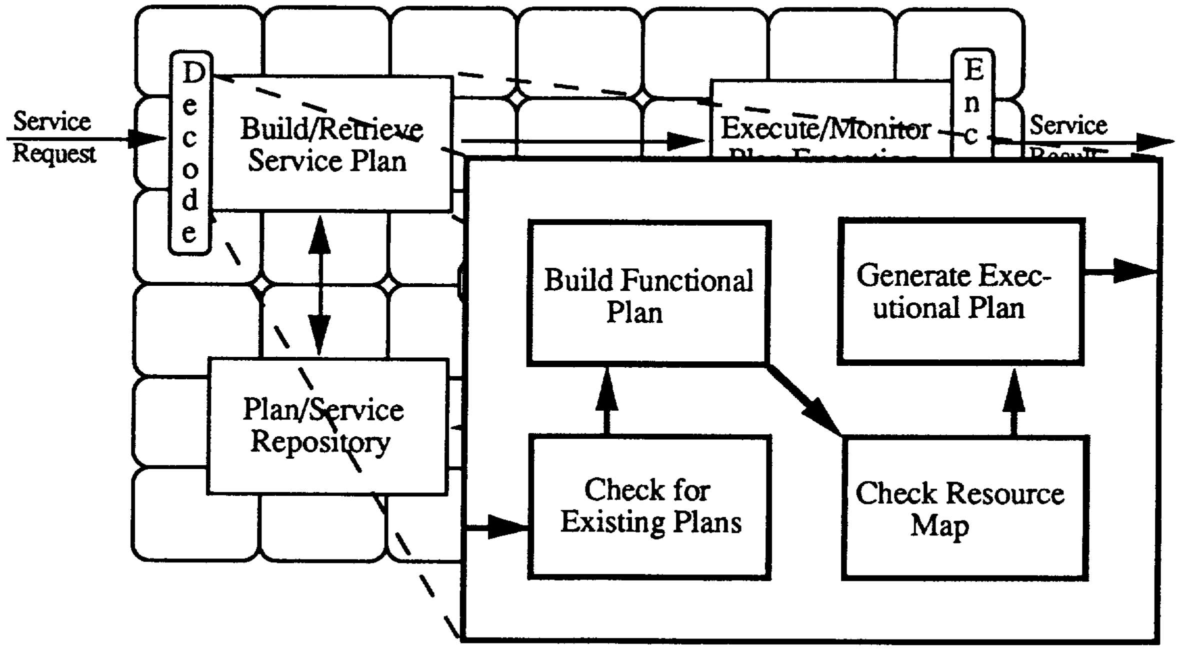


Figure 6. Operation of the Plan Builder

However, in the absence of a pregenerated plan, the planner attempts to build a plan based on the knowledge stored about existing services and the information provided in the service request. An attempt is first made at building a functional plan. In doing this, the Plan Builder first examines the service request to determine its source and destination formats. Second, the format classes of the formats are determined, and a high level (and rapid) search of the the service advertisements is done to determine whether a plan can be built between the format classes. Next, a more detailed search is done of the service protocols for the chosen service advertisements. This search will discover whether or not there is a sequence of services available that will go from the source format to the destination format. If this sequence of services is discovered, it is recorded as the functional plan and this plan will then be used to generate executable plans.

An executable plan also takes several steps to produce. First, the service protocols in the functional plan are examined to determine the service contracts that are available to satisfy the service request. These service contracts determine which utilities on which machines must be invoked to provide the necessary services. The contracts also specify the types and structure of the arguments necessary to invoke the utilities. The arguments needed are collected from the framework and/or the user and combined with the argument specification information to generate an executable plan that is then passed to the plan executor.

It is possible that several executable plans may be generated from a single functional plan. Which executable plans are generated and/or executed will be controlled by user access privileges defined by the framework. In order for the user to be able to use a service plan, the user must have the authority, based on user roles defined by the Site Specific Framework, to perform every step in the service plan. In the event that two or more executable plans (for which the user has complete access) are generated for a service request, factors that may be considered in choosing which executable plan to execute are: the number of plan steps, the estimated execution time of the plan, or user preference.

The Plan Builder also has the capability of plan validation before the plan is submitted to the Plan Executor for execution. The validation check can include data checks, service availability checks, and authorization checks. Should the validation process fail, the Plan Builder can attempt to construct a new service plan. Another aspect of the plan validation process is the identification of steps in the service plan that require manual execution. This situation might come about when dealing with legacy tools that do not support invocation by the Integration Mechanism. While these legacy tools may not support the remote execution capability, the tools can still provide integration services. The only difference is that the steps in performing that service will have to be performed manually. Indication as to whether a service requires manual intervention should be provided with the service registration, and, should be recognized by the plan builder during the plan generation process. When a manual operation is detected, notification should be sent to the user specifying exactly what operations need to be performed to perform the integration service.

4.2.3.2 Plan Executor/Monitor Component

The Plan Executor/Monitor (PEM) component of the Integration Mechanism is responsible for executing the plans passed to it from the Plan Builder and then monitoring the execution of these plans.

The executable plan details the services, programs, and data sets that need to be invoked to fulfill an integration service. The PEM spawns processes that govern each plan execution, thus preventing bottlenecks caused by a plan waiting for an earlier one to complete and allowing multiple plans to run concurrently. The plan is stepped through and appropriate local and remote service calls (remote Integration Mechanism requests) are made. It must be understood that the services are independent modules that could be a simple operation on a data set, an interactive session, or even a full-blown application (Figure 7). The piecemeal results of each step are collated or passed on to the following execution step. The final results are passed through the encoder before being returned to the caller to ensure that the information being returned is in an understandable format.

If, at any stage of this process, the plan execution were to fail, the PEM takes appropriate action according to information stored in framework or specified by the service request. These actions include user intervention or a request to the Plan Builder to replan either the whole plan or just the failed section. Except for these possible interactions with the user, the entire process of plan execution is completely transparent to the user.

Another important aspect of the PEM involves the issue of security and access privileges. It must ensure that, during the execution of a service plan, the processes are performed with the same access privileges of the user requesting the service. This verification can occur on many different levels. Certain access privileges, defined by the framework, deal with access to certain integration services and processes and are detected by the Plan Builder during the plan generation process. Other privileges deal with access to remote machines. It may be the case that a user is not authorized to log on to a certain machine and yet a service plan generated for the user that requires execution of a service on that machine. It is important that the PEM notice this constraint and not allow it to be violated. In a situation where an access violation would occur, the PEM will notify the user and indicate why the service is not being provided.

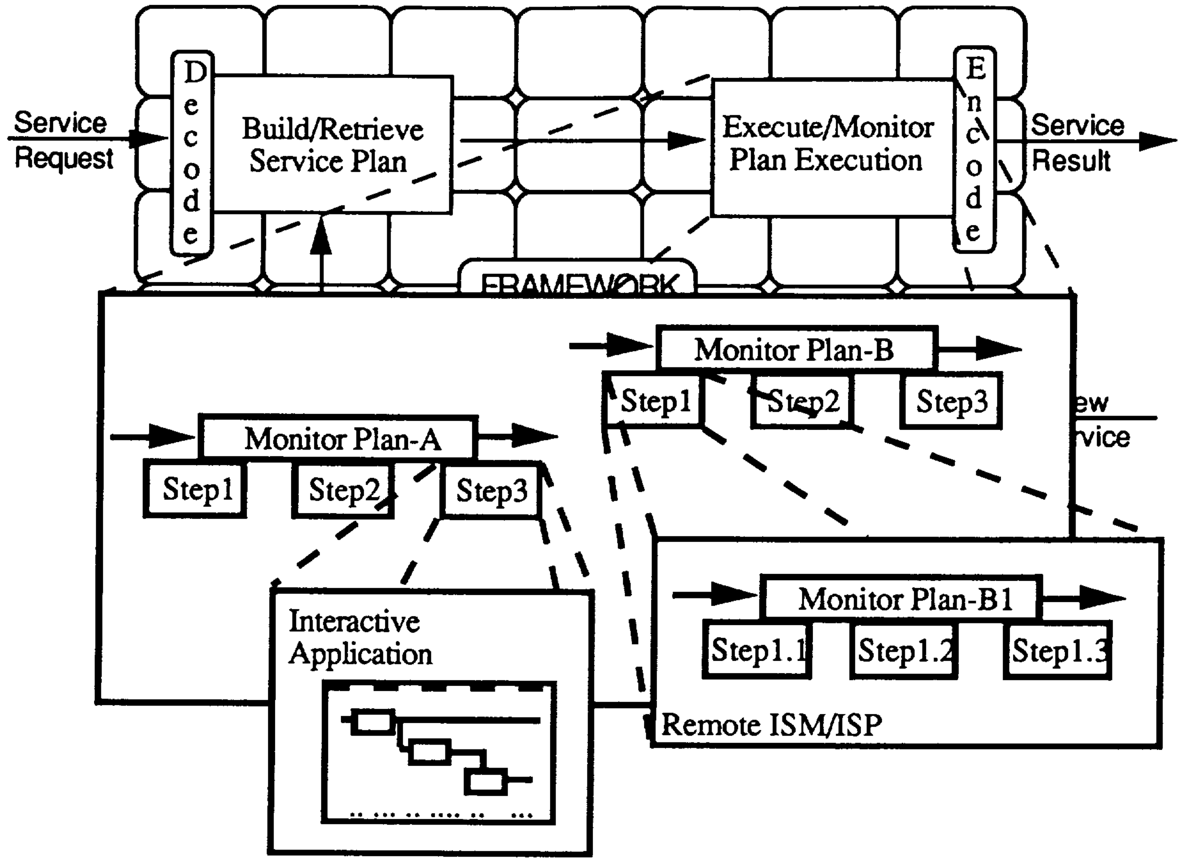


Figure 7. Operation of the Plan Executor

4.2.3.3 Decoder/Encoder Component

The Decoder/Encoder component of the Integration Mechanism is the interface between the Integration Mechanism and the other components of the FPP. All communications to and from the Integration Mechanism are coded in the same format, called an *external format* in this discussion. This is necessary to alleviate the need to reprogram the Integration Mechanism to understand a new format each time a new component is added to the FPP. This will also allow the Integration Mechanism to evolve and change, as long as the communications interface standard is kept intact.

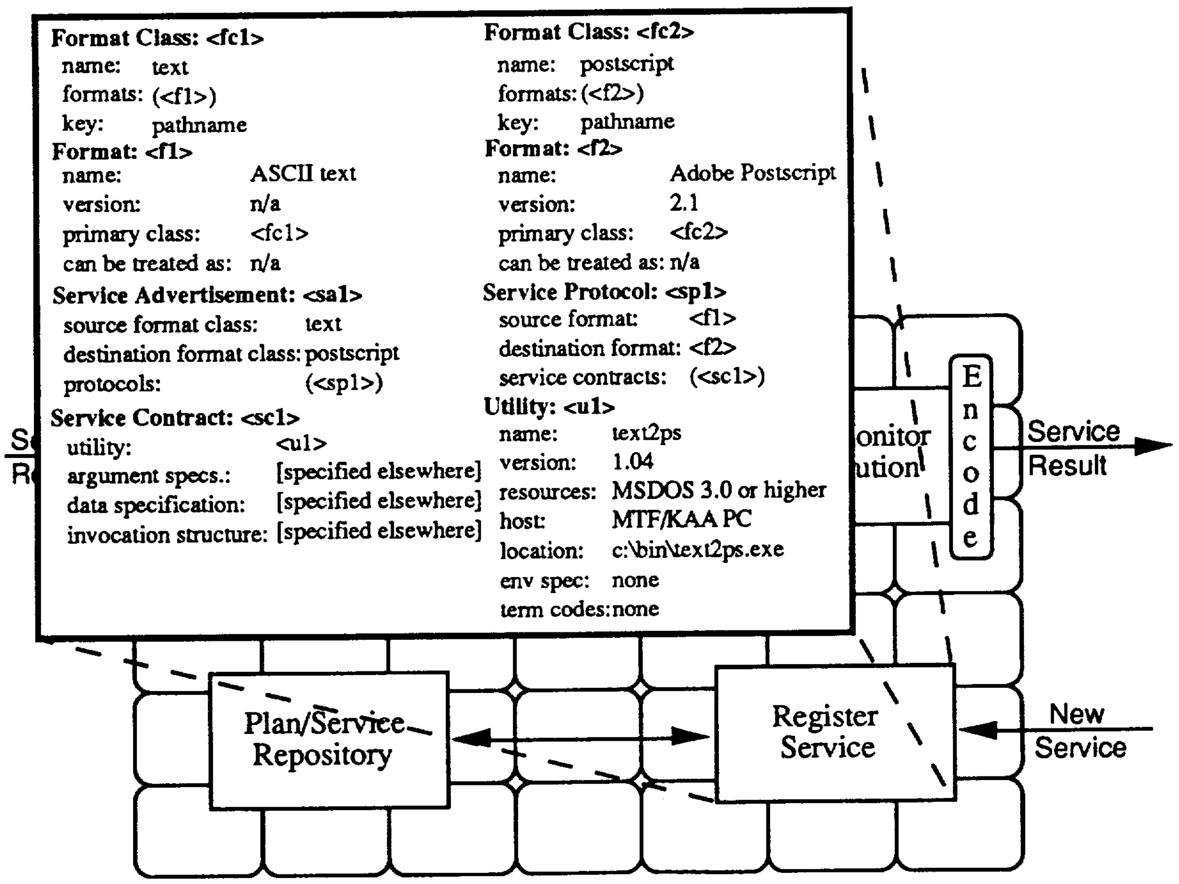
Incoming messages, including service requests, data objects, pregenerated plans, constraint sets to be used while planning, and replies from remote Integration Mechanisms, are decoded from the external format and routed to the appropriate Integration Mechanism component. Outgoing messages, including requests for service definitions and predefined plans, messages to the DOM for data objects, messages to an application or user for intervention, and messages to a remote Integration Mechanism for a service, are encoded into the external format and routed to the appropriate FPP component.

4.2.3.4 Service Registration Tool

The Service Registration Tool (SRT) is an interface designed to facilitate the registration of services with the Integration Mechanism. It also allows for the logging of service advertisements that it receives from remote machines.

As discussed in Section 4.1, the Integration Mechanism requires that certain information be specified about a service before it can be registered with the Integration Mechanism. Figure 8 shows a template of the type of information that will have to be captured by the Service Registration Tool.

Using the interface provided by the SRT, the user can browse existing services in the service database, and build new services that use the existing format and data types, or existing services. The user can also define and register new formats and build new format classes incorporating existing formats. Browsing existing services formats allows the user to determine if there exists formats that may have a different names but having the same structure and thus be included in the same format class.



Format Class: <fc1>
name: text
formats: (<f1>)
key: pathname
Format: <f1>
name: ASCII text
version: n/a
primary class: <fc1>
can be treated as: n/a
Service Advertisement: <sa1>
source format class: text
destination format class: postscript
protocols: (<sp1>)
Service Contract: <sc1>
utility: <u1>
argument specs.: [specified elsewhere]
data specification: [specified elsewhere]
invocation structure: [specified elsewhere]

Format Class: <fc2>
name: postscript
formats: (<f2>)
key: pathname
Format: <f2>
name: Adobe Postscript
version: 2.1
primary class: <fc2>
can be treated as: n/a
Service Protocol: <sp1>
source format: <f1>
destination format: <f2>
service contracts: (<sc1>)
Utility: <u1>
name: text2ps
version: 1.04
resources: MSDOS 3.0 or higher
host: MTF/KAA PC
location: c:\bin\text2ps.exe
env spec: none
term codes: none

Figure 8. Service Registration

Potential users of the tool would be vendors defining new services for their tools, FPP Site Administrators defining new format classes, and users registering new services for utilities they have produced.

While the other three components of the Integration Mechanism are closely related, the Service Registration Tool stands on its own, and is used separately from the other components. This tool interfaces with the other components of the Integration Mechanism through the Plan/Service Repository. This repository is just the collection of all the registered services available to the FPP and is a partition of the System Resource Definition Knowledge Base.

Note: This module may also provide an interface to the database of service plans maintained by the Plan Builder and allow the user to browse and edit those plans. Using such an interface the user could instruct the Plan Builder to generate all possible plans for a particular transaction and then choose the one that he would like to have executed. The facility to pre-can plans is necessary as the required plans could be generated as the need for them arises, thus eliminate planning on the fly.

5 Status and Future Directions

As the design of the Integration Mechanism has been the first step in the design of the overall FPP, the previous sections have presented a snapshot of the current FPP design. Future work will be directed first towards the design of the Framework Processor and then towards the design of the remaining components of the FPP.

5.1 Outstanding Design Issues

While the near term work on the FPP is to be directed towards the design of the Framework Processor, there are still open issues concerning the design and operation of the Integration Mechanism. This section is meant to highlight areas of the current design that have potential deficiencies or areas where possibility for change have been identified.

Separation of the Data Object Manager and the Integration Mechanism. In the FPP Preliminary Design described in Section 3, the DOM and the Integration Mechanism were shown to be distinct components. However, in the later stages of the Integration Mechanism design, these components seem to be grow closer together. They were initially separated in the design to distinguish between data operations and function operations. It now appears that, while the operations may be different, the operations of the DOM and the Integration Mechanism might be handled in a single, consistent fashion. This would allow the two components to be logically combined, perhaps having the Integration Mechanism layered on top of the DOM. The effect that this combination would have on the Integration Mechanism is to require that all data operations be routed through the Integration Mechanism.

Related to this point is the role the DOM will play in managing the service information. The current design calls out for the service information to be managed by the DOM so that a single representation can be maintained across the entire platform. However, it is possible that performance issues could require a change in this operational philosophy. An alternative strategy might be to have service information maintained by the specific Integration Mechanism with which the service is originally registered. This Integration Mechanism could then "publish" the service information to the other Integration Mechanisms. This option would also reduce the amount of message traffic between the Service Registrar and the DOM.

Integration Mechanism Component Interface Protocols. The design of the Integration Mechanism has partitioned the IM into four functional components. However, protocols for communicating between these components has not been established. These components may or may not need formal protocols as the protocol may just be a function call. For example, the plan builder may simply call the plan executor with a call like:

```
(execute-plan <plan-object>).
```

Regardless of the degree of formality, as the design becomes more detailed, these interfaces should be defined due to the distributed nature of the FPP. Since the Integration Mechanism will run on many different platforms, portability issues are very important to the definition of the Integration Mechanism. Definition of these external component interfaces will make portability easier by allowing components to be redesign, re-implemented or replaced without impact on the other components.

External Interface for Service Registration. Section 4.1 detailed the format in which service information will be represented to the Integration Mechanism. It was also stated that this representation was an internal format. In other words, the description detailed how the information would appear to the Integration Mechanism. The format by which the service information will be transmitted from the FPP administrator to the Integration Mechanism has not been established. It is quite possible that a combination of methods and interfaces will be required to support the service registration process.

Service Query Language. No service query language for requesting the execution of a service has been established. As the format of service plans, both functional and executable, become more clear, it is expected that a structure for the query language will be established.

Framework / Integration Mechanism Interaction. While the design of the Integration Mechanism has made reference to the interaction between the Integration Mechanism and the Site Specific Framework, the formal mechanisms by which the two components will interact have not been established. It is expected that the design of the Framework Processor, the next step in the FPP design, will contribute to the definition of these mechanism.

5.2 Requirements Matrix

It is important to recognize that the design of the Integration Mechanism has been driven by the requirements established for the FPP. The matrix in Figure 9 relates the requirements [FPP 90b] satisfied by the design of the Integration Mechanism to the actual component of the Integration Mechanism that satisfies the requirement. In many cases, a requirement is satisfied through a combination of several Integration Mechanism components.

	Encoder / Decoder	Plan Builder	Plan Executor	Registrar
2.4.2	√	√	√	
2.9.1.1	√	√	√	
2.9.4.1	√	√	√	√
2.9.4.2				√
2.9.4.3	√	√	√	
2.9.4.4				√
2.9.4.5	√	√		√
2.9.4.6	√	√	√	
3.1.3.3	√	√	√	
3.2.1				√
3.2.2				√
3.2.3				√
3.2.4				√
3.3.1.1.2	√	√	√	√
3.3.2.1	√	√	√	
3.3.2.2				√
3.3.2.3	√			
3.3.2.4				√
3.3.2.5	√	√		√
4.4.1.1	√	√		
4.4.1.2	√			
4.4.1.2.1	√	√	√	
4.4.1.3	√			
4.4.2.1		√		
4.4.2.2		√		
4.4.4.3	√		√	√
4.4.4.4			√	
4.4.4.5	√		√	
4.4.4.6	√		√	
4.4.4.6.1	√		√	
4.4.4.7			√	
4.4.4.7.1	√		√	
4.4.5.1				√
4.4.5.1.1				√
4.4.5.2				√

Figure 9. Integration Mechanism Requirements Matrix

As would be expected, the requirements satisfied deal directly with the provision of a integrated development environment. However, not all integration requirements are being satisfied entirely by the Integration Mechanism since certain integration requirements extend past the scope of the Integration Mechanism. The Integration Mechanism is responsible primarily for the support of the integration services approach to integration. Those integration aspects of the FPP that deal more with the control of data and processes will be dealt with by other components of the FPP, though the Integration Mechanism may play a contributing role.

Also, as the design of the Integration Mechanism will be open to modifications, this requirements matrix will also be subject to changes. We matrix shown above indicates those requirements that are either completely or partially satisfied by the Integration Mechanism. As the designs of the other FPP components are completed, the role the Integration Mechanism plays in partially satisfying these requirements will become more clear. As a result, the final requirements for the Integration Mechanism will not be established until the design of the other components is completed.

6 References

[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.

[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.

[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.

[I^{2}S^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.

A Appendix A - Service 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].

Lexical Conventions

This section describes the lexical conventions used in the definition of the specifications. 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.

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.

3) Within an argument specification the form (**get_input**) accesses the value specified by the user for this argument. The type of the value depends on the defining argument specification class. For example, in **arg_boolean** argument specification (**get_input**) would return either **true** or **false**.

```
argument-specification ::= ( {  arg-boolean-decl |
                                arg-choice-decl |
                                arg-integer-decl |
                                arg-real-decl |
                                arg-string-decl }* )
```

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 |
                        ( get_input ) |         ;; reference to a string argument
                        ( value identifier ) |
                        ( concatenate string-value+ )

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

boolean-expression ::=          true |
                              false |
                              ( get_input ) |  ;; reference to a boolean argument
                              ( 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

data-definition ::= ( { ( data_def
                              identifier
                              ( direction { input | output | inout } )
                              ( arg_ref identifier )
                              { ( required_if boolean-expression ) }?
                              { ( exists_if boolean-expression ) }? ) }* )

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

exit-code ::= ( { ( result    integer
                              { success | warning | error | failure }
                              { ( label string ) }? ) }* )
```

B Appendix B - Integration Mechanism Functional Model

The following pages show the IDEFØ Functional Model of the current FPP Integration Mechanism design. These diagrams show activities that will be performed by the Integration Mechanism, the relationships between those activities, and the controls on and the mechanisms for performing those activities.

When building or examining an IDEFØ model, it is important to establish and understand the point of view from which the model is defined. This point of view is defined in the following statements.

Purpose:

To define the major functions of the Integration Mechanism and to show the relationships among these functions.

Viewpoint:

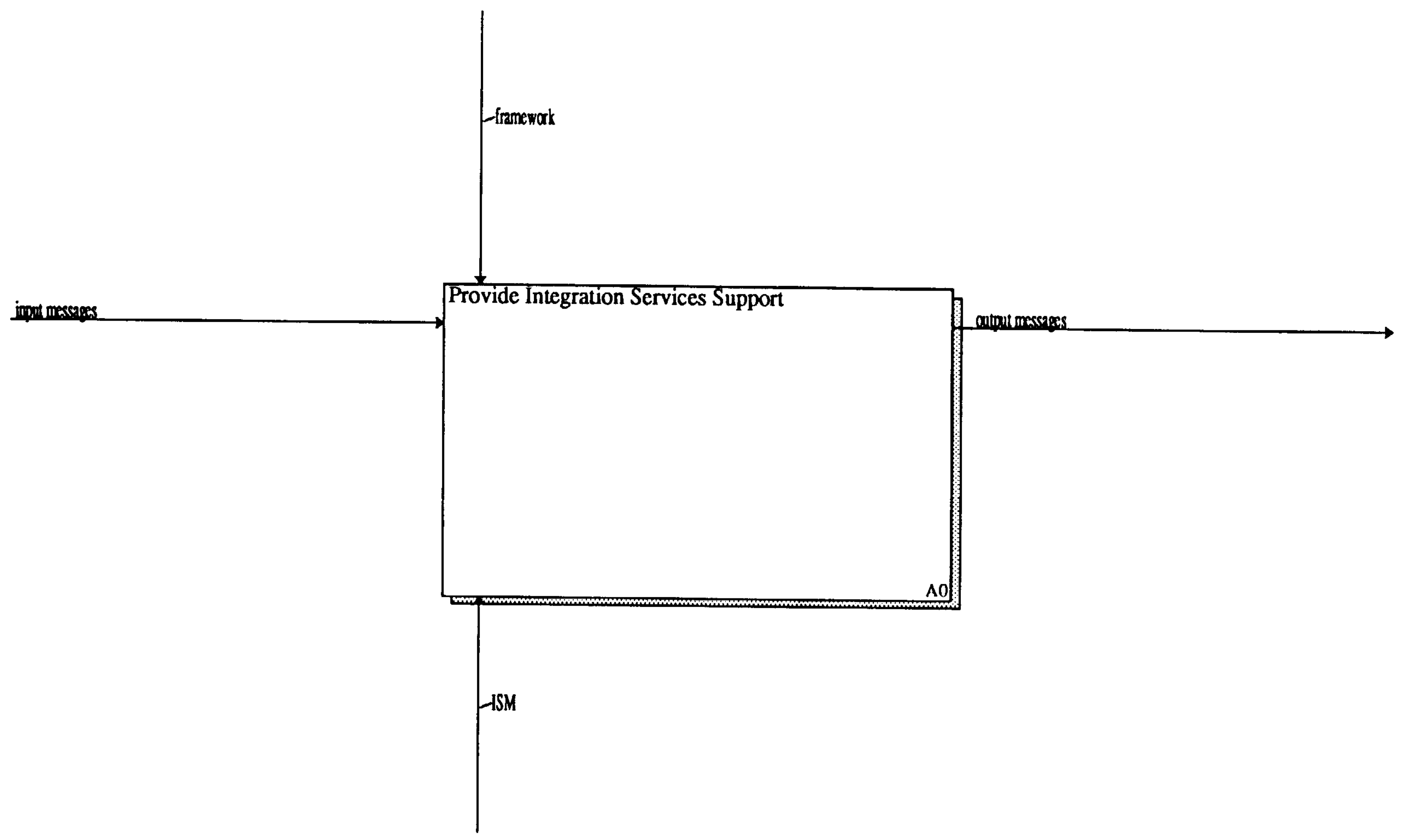
This modeled is presented from the viewpoint of the Integration Mechanism of the FPP.

Context:

This model exists in the context of providing integration services to the users of the FPP.

Because of the point of view established for this model, the following model does not incorporate the Service Registration component. The reason for this is that the Registration utility operates separately from the more common operations of the other Integration Mechanism components.

Used At: KBSI	Author: IDSE Level 0 Design Team	Date: 6/17/1991		Working	READER	DATE	Context
	Project: IICE Experimental Tools	Rev:		Draft			
	Notes: 1 2 3 4 5 6 7 8 9 10			Recommended			
				Publication			



Node:A-0	Title:Provide Integration Services Support (Context)	Number: 1

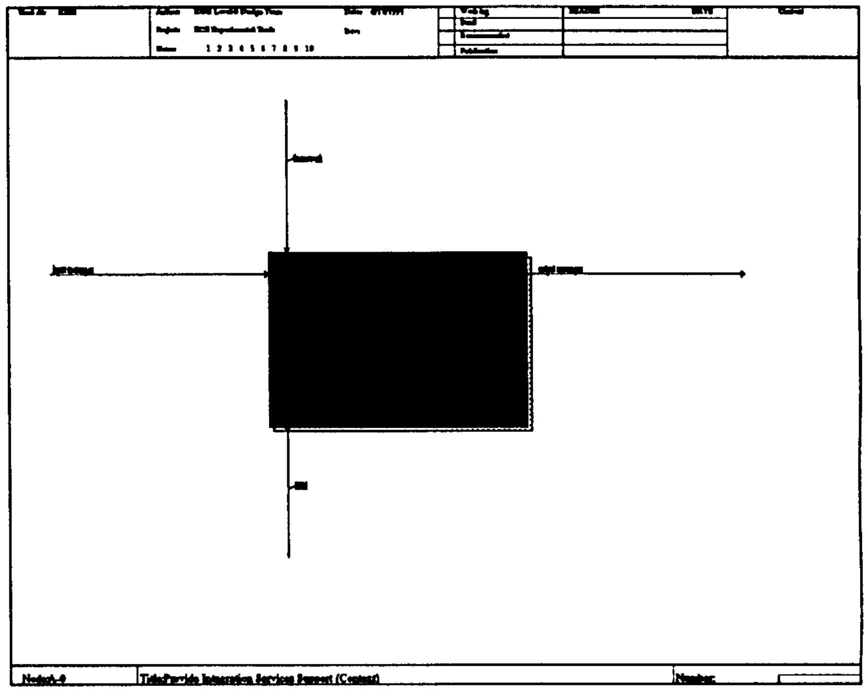
Provide Integration Services Support

These are the major components of the Integration Services Manager (ISM) of the Framework Programmable Platform for the Advanced Software Development Workstation.

The ISM is an expert system that maintains information about all of the software utilities available to the enterprise that aid in the software system development process. Using this information, the ISM automates the application/invocation of these utilities to/on various data stored across a hetergenous collection of platforms.

The operation of the ISM is as follows:

1) All external messages received by the ISM are received through the decoder. The messages are decoded and routed to the appropriate ISM component. For example, a service request is received and decoded
by the decoder and routed to the plan builder.
2) The decoded request is given to the plan builder. The plan builder checks to see if a plan that would satisfy the request is available from the library. If so, that plan is sent to the plan executor. Otherwise, a new plan is generated, stored in the library, and sent to the plan executor. If a plan cannot be found or generated, the service request is recorded in a log. This log would serve as a reference for future service development.
3) The plan is passed to the plan executor. The plan executor executes each instruction in the plan. It is possible that an instruction cannot be executed. Depending on the controlling information provided with the service request and in the framework, and the reason
that the instruction failed to execute, several actions may be taken. These include: a) trying to execute the instruction again, b) generating an alternate instruction, c) requesting that an alternate plan be generated, or d) asking the service requestor for assistance. Upon successful completion of the plan, any results are returned to the service requestor.
4) All outgoing messages generated by the components of the ISM are sent
to the encoder. The messages are encoded into the proper format and routed to the appropriate FPP/ASDW component. For example, a Domain
Object Manager request from the plan builder would be encoded into a format understandable by the Domain Object Manager and then sent to it.



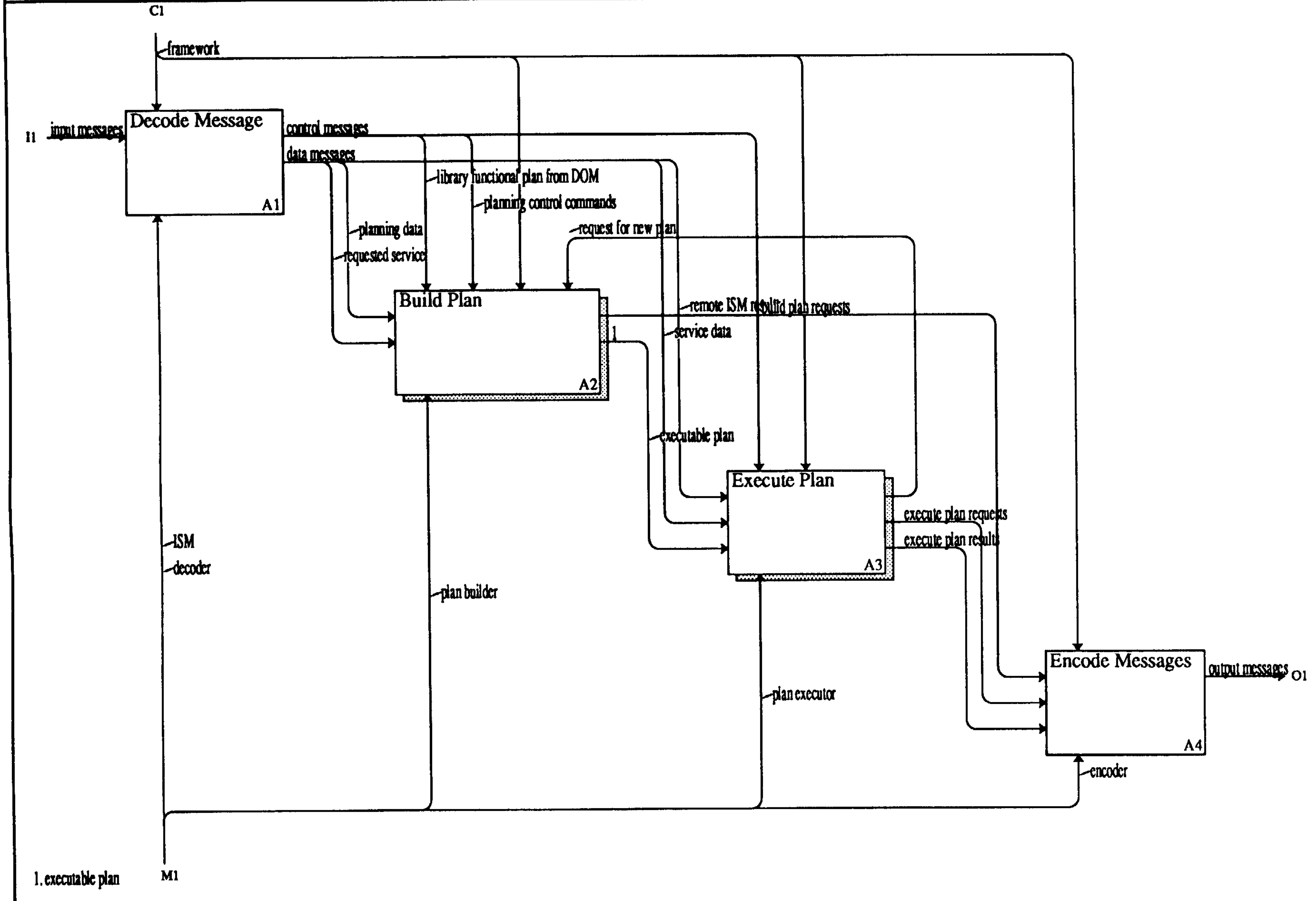
Important Note:
All activities produce notifications about the status of the process they are performing.

Purpose: Determine the functions/interactions of the ISM
The purpose of this model is to define the major functions of the ISM and to show the relationships among these functions.

Viewpoint: Integration Services Manager
This is modeled from the viewpoint of the Integration Services Manager for the FPP/ASDW.

Context: Providing Integration Services
This model occurs in the context of providing integration services between the users of the FPP/ASDW (both personnel and applications), the Domain Object Manager, and the Facilitator. These services include coordinating the activities between the user and his/her local applications with remote applications and the development and execution of integration services plans.

Used At: KBSI	Author: IDSE Level 0 Design Team	Date: 6/17/1991	Working / Draft / Recommended / Publication
	Project: IICE Experimental Tools	Rev:	READER / DATE
	Notes: 1 2 3 4 5 6 7 8 9 10		Context



Node:A0 | Title:Provide Integration Services Support | Number: 4

Decode Message

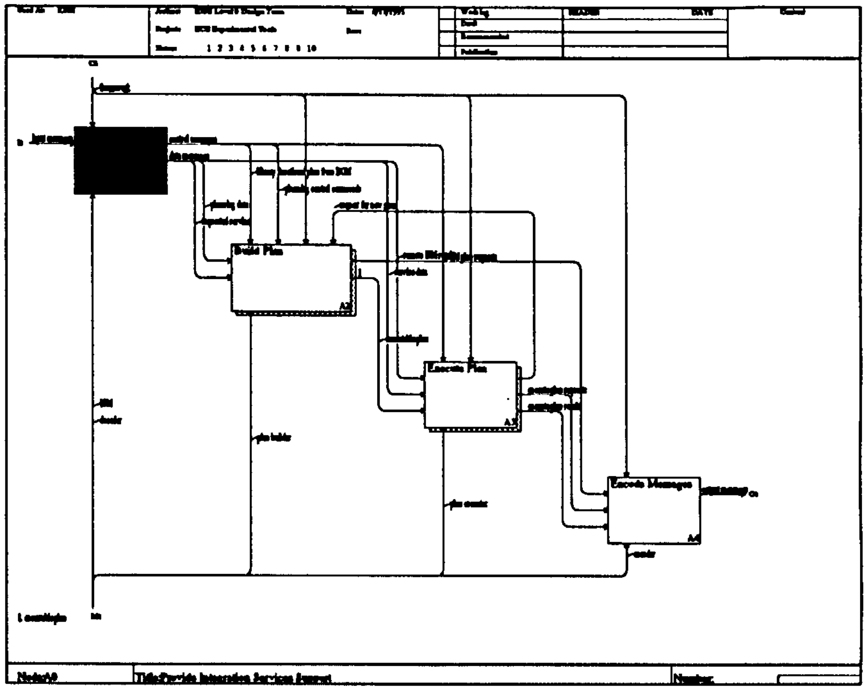
All messages sent to the ISM pass through this activity. It decodes the messages and passes them to the appropriate components. Messages sent to the ISM include:

1) Service Requests

These are requests from an application for some action to be taken. For example, "translate this IDEF0 model into an IDEF1 model" is a service request.

2) Responses to queries

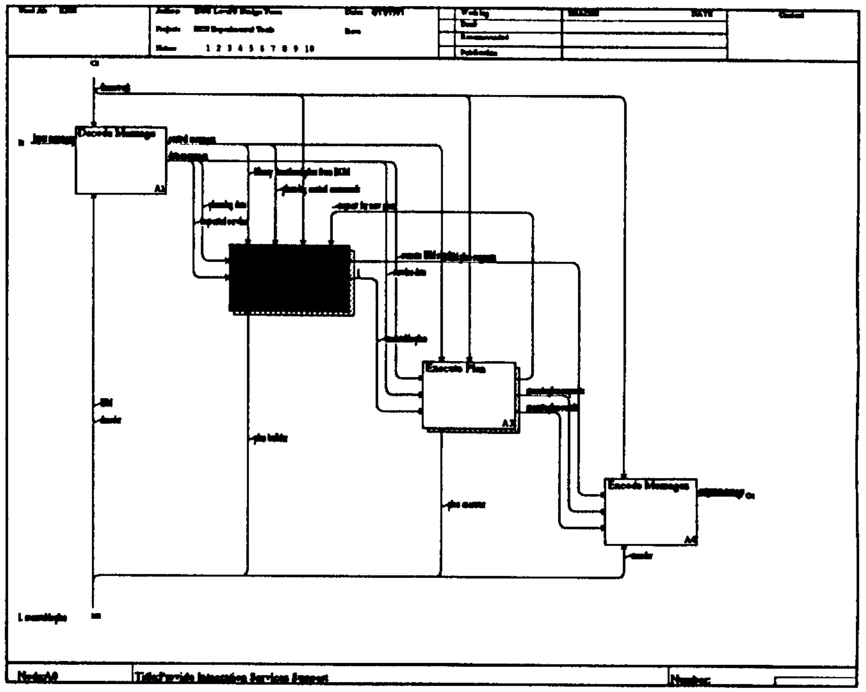
These are responses from applications to queries generated by the components of the ISM. For example, the plan builder might request (through the encoder) that the DOM send it a functional plan from the library. The DOM would send the requested plan through the decoder to
the plan builder.



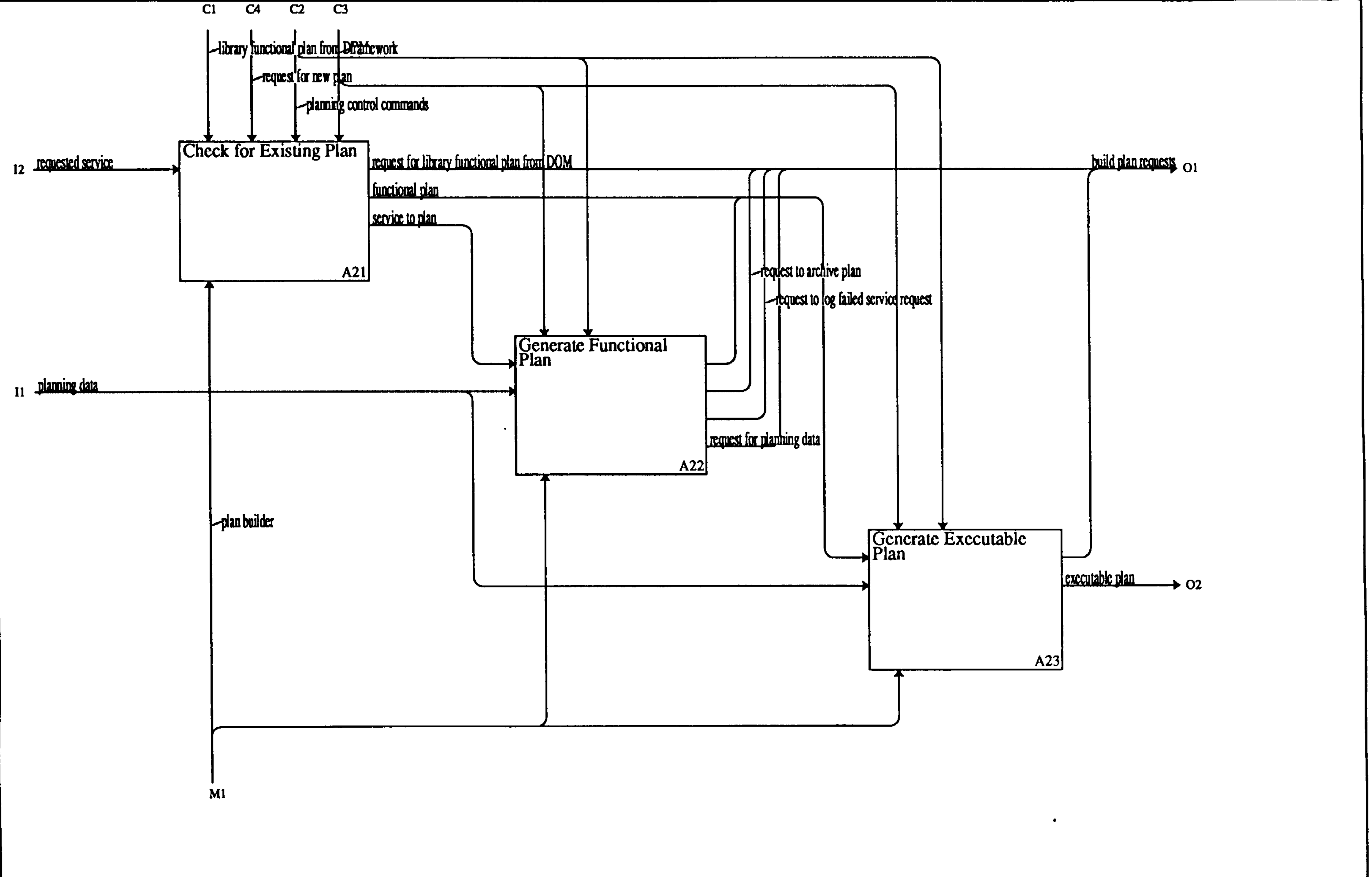
Build Plan

This activity describes the plan builder. The plan builder is responsible for generating plans to satisfy service requests. After receiving a service request, the following occurs:

1) The plan library is checked to see if a functional plan that will satisfy the request already exists. If so, that plan is used, and the process continues at step 3.
2) If no functional plan is found in the library, a functional plan is generated from the service planning data maintained in the DOM. If a functional plan cannot be generated, the service request is recorded in a log of unfulfillable service requests.
3) The functional plan is then converted into an executable plan. This is
done by changing the service advertisements in the functional plan into hardware/software specific instructions to be executed by the plan executor.



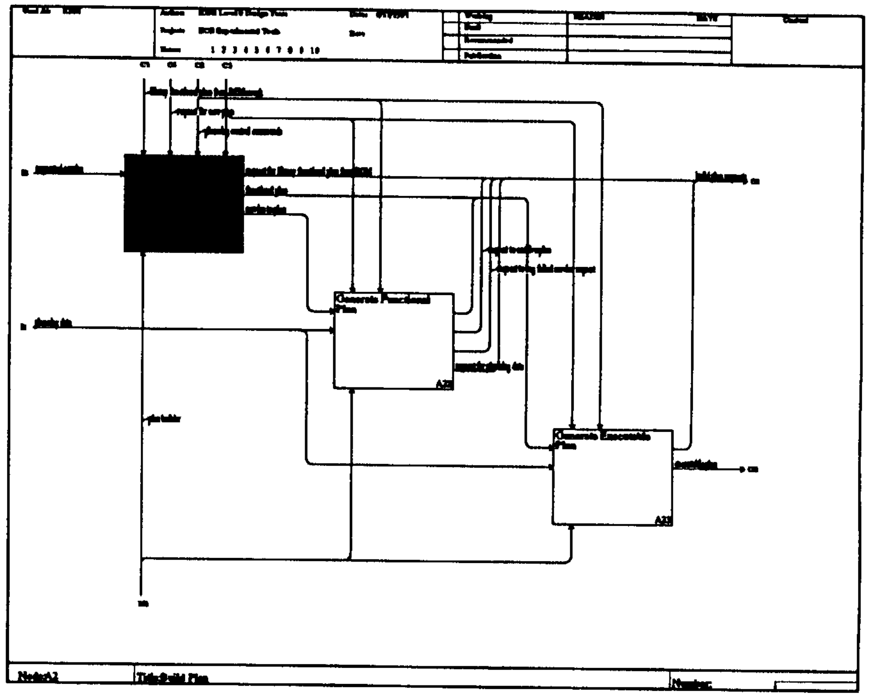
Used At: KBSI	Author: IDSE Level 0 Design Team	Date: 6/17/1991	Working	READER	DATE	Context
	Project: IICE Experimental Tools	Rev:	Draft			
			Recommended			
	Notes: 1 2 3 4 5 6 7 8 9 10		Publication			



Node:A2	Title:Build Plan	Number: 7

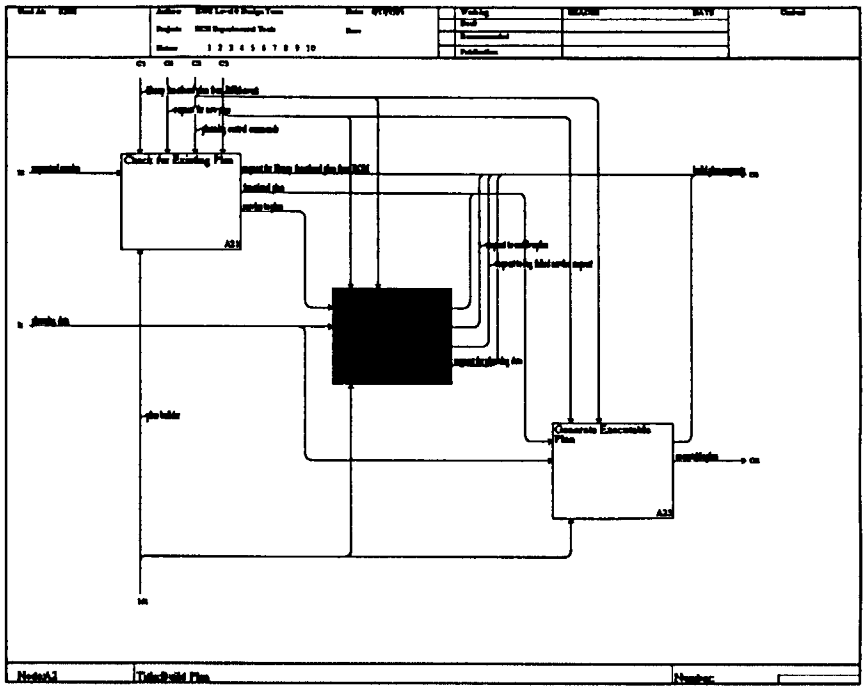
Check for Existing Plan

This portion of the plan builder is responsible for querying the library of functional plans to see if one exists that will satisfy the service request. If a functional plan is found, it is sent to the executable plan generation portion of the plan builder. Otherwise, the service to be planned is passed on to the functional plan generating component.



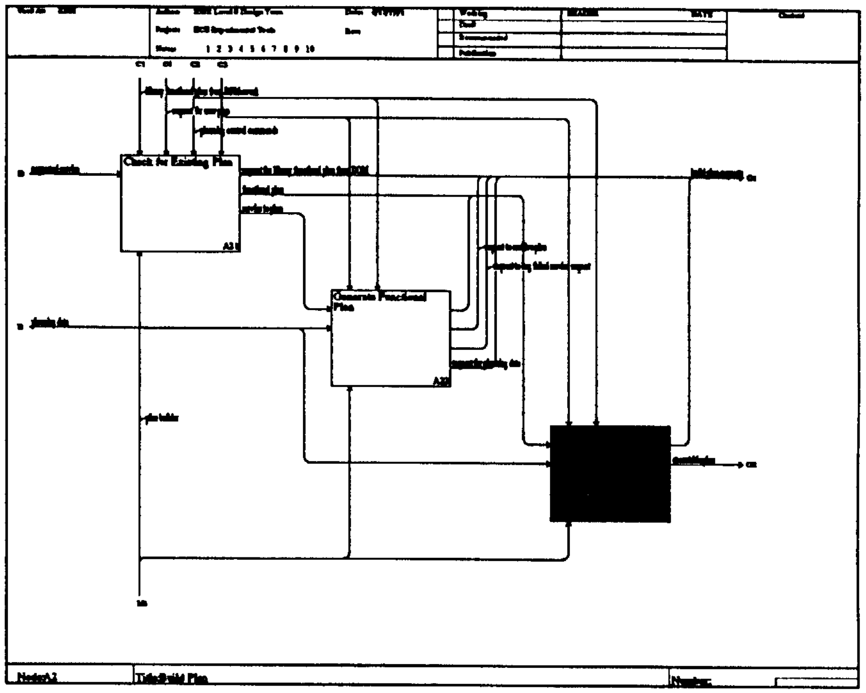
Generate Functional Plan

This portion of the plan builder is responsible for generating new functional plans. Given a service to plan, it consults the repository of tool artifact information and generates a service plan devoid of hardware/software specific information (ie, a functional plan) that should satisfy the request.



Generate Executable Plan

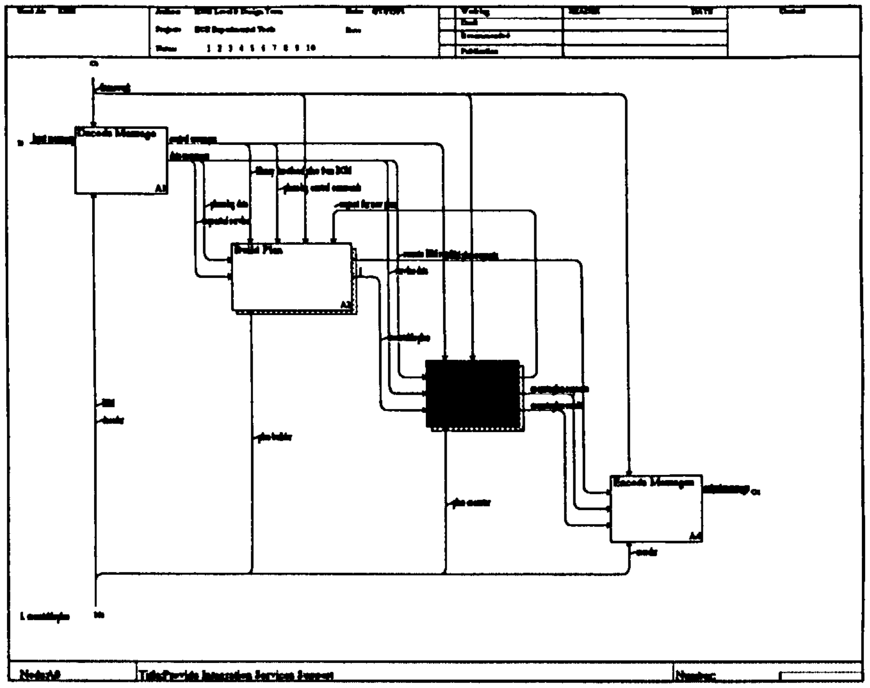
This portion of the plan builder is responsible for translating a given functional plan into a hardware/software specific sequence of instructions that can be executed by the plan executor.



Execute Plan

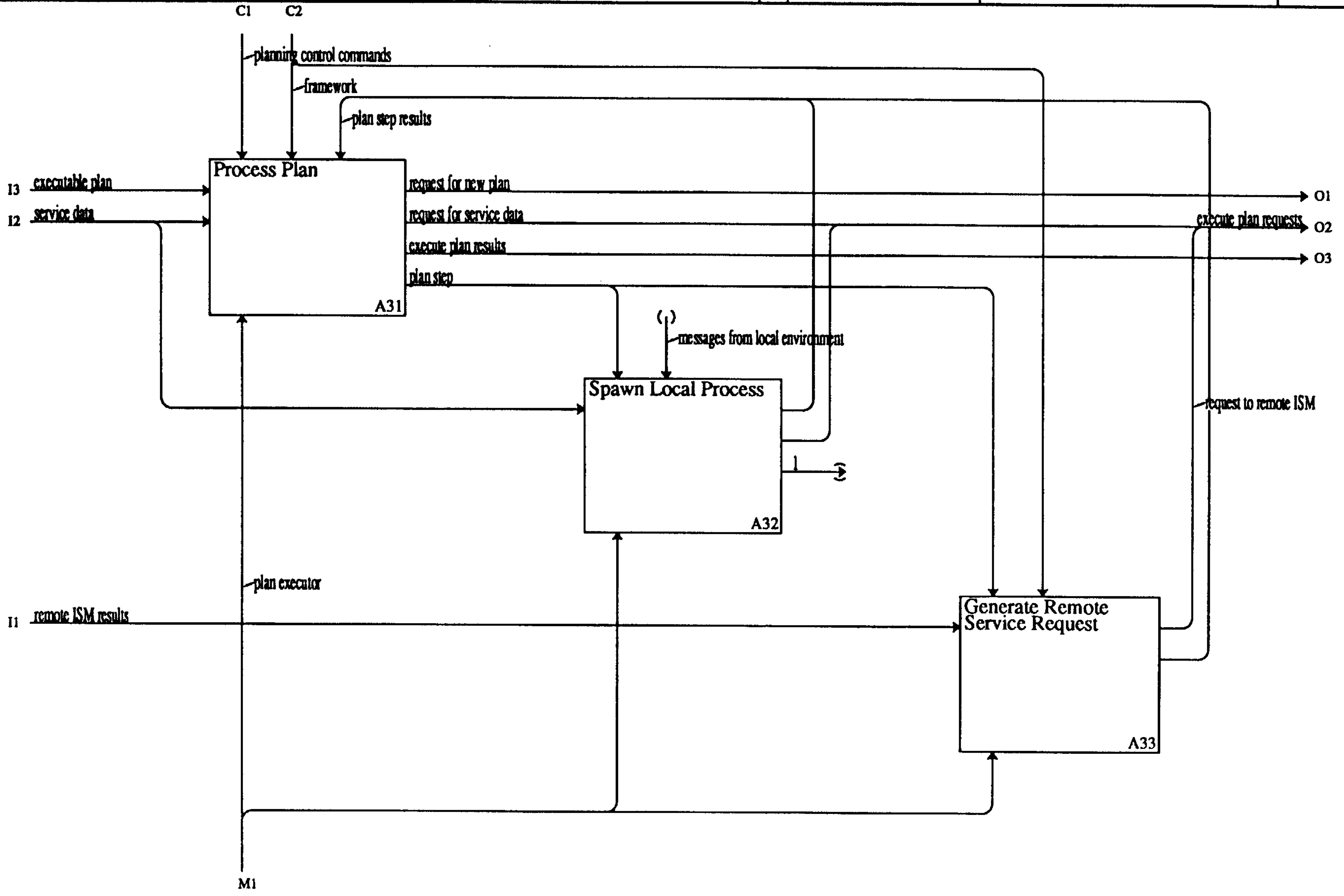
This is the plan execution component of the ISM. It is responsible for executing the executable plan. Given an executable plan, the following steps occur:

1) The plan processor breaks up the plan into individual instructions.
2) For each instruction, the plan processor determines whether the the instruction can be performed locally (step 3) or must be parceled out to a remote ISM (step 4).
3) The local instruction is passed to the local process controller. It spawns the process necessary to carry out the instruction, monitors the progress of the process, and returns the results of the process execution to the plan processor (go to step 5).
4) Remote instructions are passed to the remote ISM request generator.
This will generate the remote ISM request, send it (through the encoder), wait for results from the remote ISM, and return the results to the plan processor.
5) If all of the steps in the plan have been executed, the results of the plan are returned. Otherwise, execution continues at step 2.



Used At: KBSI	Author: IDSE Level 0 Design Team	Date: 6/17/1991	Working	READER / DATE
	Project: IICE Experimental Tools	Rev:	Draft	
			Recommended	
	Notes: 1 2 3 4 5 6 7 8 9 10		Publication	

Context



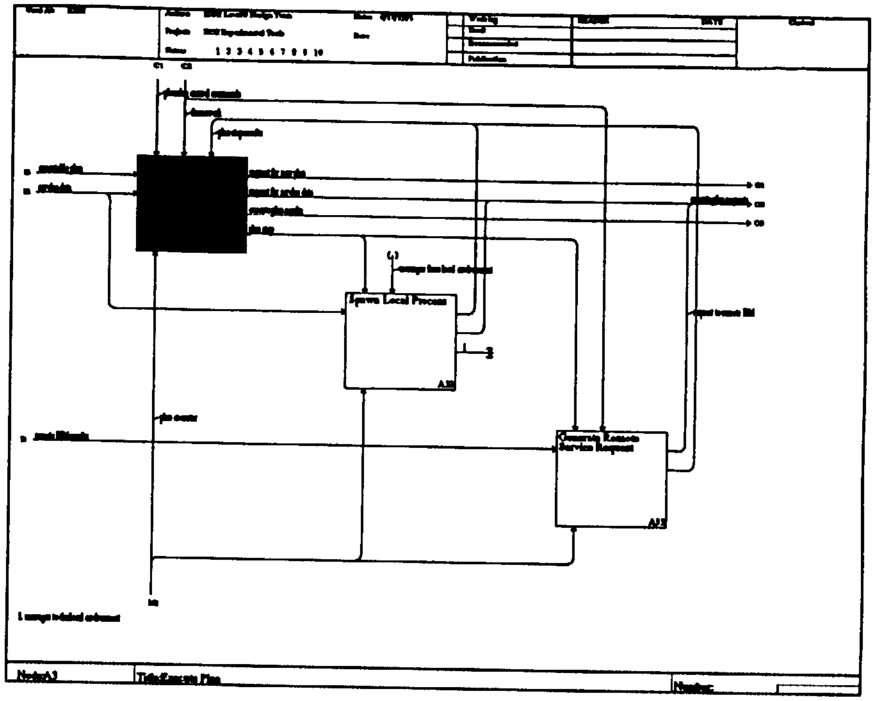
1. messages to the local environment

Node:A3	Title:Execute Plan	Number: 12

Process Plan

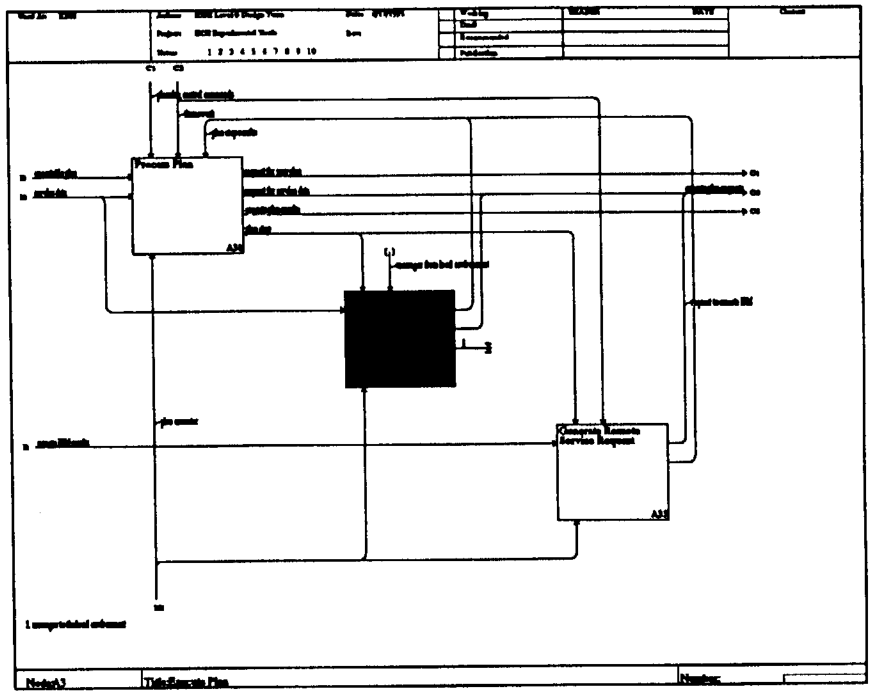
This is the plan processor portion of the plan executor. It takes an executable plan, executes it, and returns the result. Given an executable plan, the following steps are performed:

1) The executable plan is broken down into individual instructions.
2) For each instruction, the plan processor determines whether the instruction can be executed locally or must be performed on a remote ISM.
3) If the instruction can be executed locally, the plan processor passes the instruction to the local process controller for execution and waits for results.
4) If the instruction cannot be executed locally, the plan processor passes the instruction to the remote ISM request generator for execution
and waits for results.
5) If the instruction executed successfully and there are more instructions,
control returns to step 2. If an error occured, a request for a new plan is generated.
6) When all instructions have been executed, the plan processor returns the
results of the plan execution.



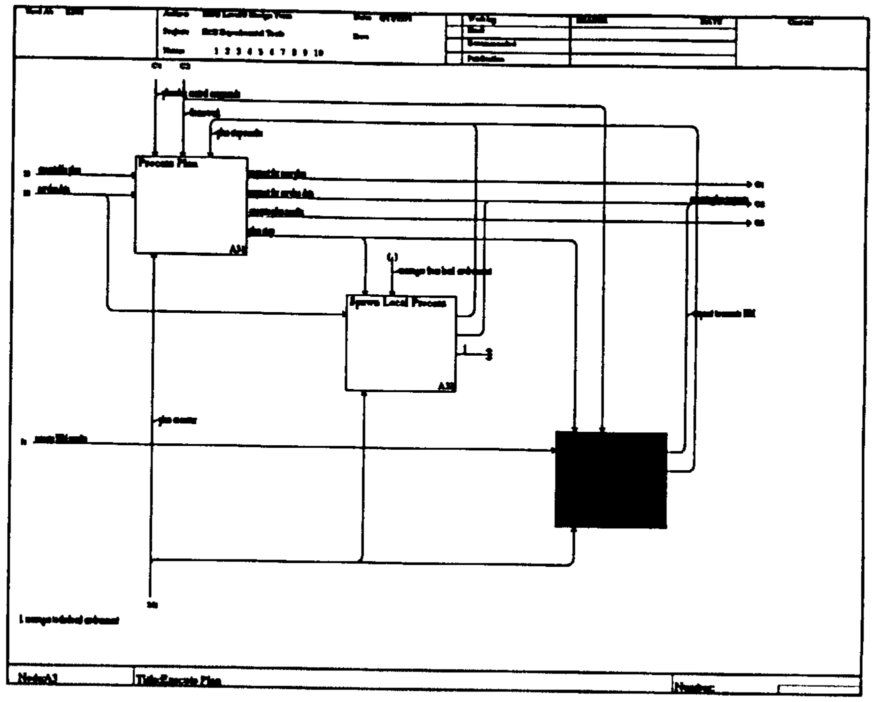
Spawn Local Process

This is the local process controller portion of the plan executor. Given an instruction, it acquires the resources and input data necessary to complete the instruction and spawns a process with this information that will actually carry out the instruction. The controller monitors the spawned process, collecting the results when the process has run to completion and passing them to the process planner. If the spawned process is unable to complete, the local process controller sends a notification to the plan processor and waits for further instructions.



Generate Remote Service Request

This is the remote ISM request generator of the plan executor. Given an instruction, it determines the appropriate ISM to receive the request (based on information in the framework and other system dependant factors), generates the appropriate remote ISM request, and sends the request (through the encoder) to the remote ISM. It forwards the response it receives from the remote ISM to the plan processor.



Encode Messages

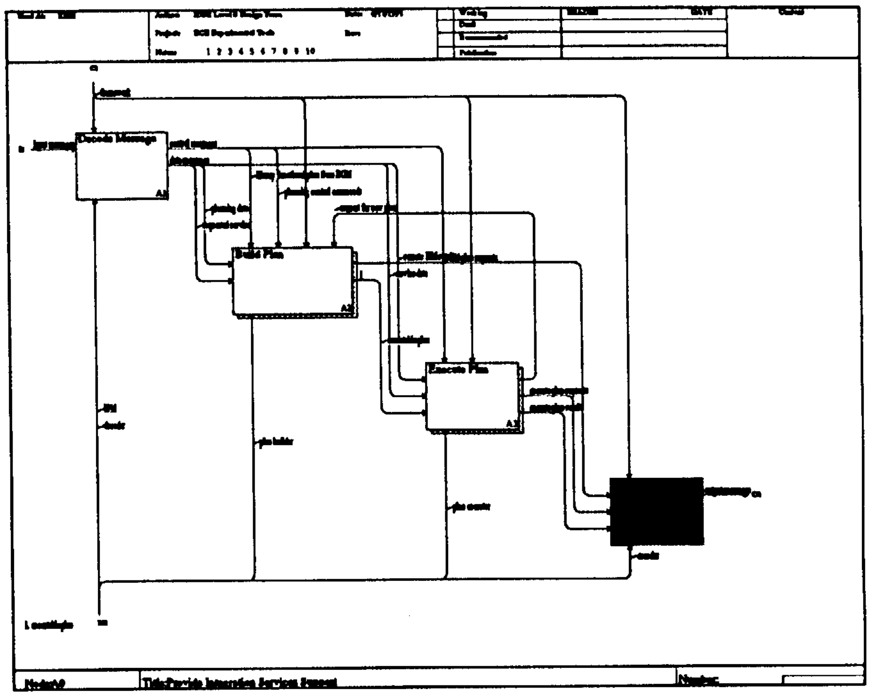
All outgoing messages generated by the ISM components pass through this activity. It encodes the messages and routes them to the appropriate FPP/ASDW component. Messages sent from the ISM include:

1) Data requests

These are the requests for data made by the ISM components. They include
requests for functional plans from the library, requests for tool artifacts, requests for planning data, requests for service input data, etc.

2) Service results

These are the results of the various activities of the ISM. These could take many forms (eg, a file, an integer, a notification, only side effects, etc).



ORIGINAL PAGE IS
OF POOR QUALITY

GLOSSARY

build plan requests

These are information queries generated by the components of the build plan activity. These include:

1) requests to the DOM for a library
2) requests for plan information
3) requests to archive (permanently store) a newly generated plan.

See the glossary entry for each of these items for more information.

control messages

These are messages received by the ISM that contain control information from the application/user, DOM, or remote ISM. These include:

1) planning control commands
2) functional library plan from DOM

See the glossary entry for each of these for more information.

data messages

These are messages received by the ISM that contain data information from the application/user, DOM, or remote ISM. These include:

1) planning data
2) remote ISM results
3) requested service
4) service data

See the glossary entry for each of these for more information.

decoder

This is the portion of the ISM responsible for decoding and routing messages sent to the ISM to the appropriate internal component.

encoder

This is the portion of the ISM responsible for encoding and routing messages from the ISM to the other components of the FPP/ASDW.

executable plan

This is a series of commands to be executed in order to complete a service request. This is somewhat analogous to a BAT file on a PC or a script file on a UNIX machine. Some example command types are:

1) requests to remote ISMs for services
2) data requests
3) tool invocations

execute plan requests

This is a collection of requests from the plan execution components. These may include:

1) requests for service data
2) remote ISM requests

execute plan results

These are the products of the execution of the service plan.

framework

This is a site specific collection of system definition and system development processes, system architecture descriptions, and method classification matrices.

functional plan

This is a hardware/software independent description of a service plan. Several executable plans may be built from a single function plan.

input messages

This is a collection of messages to be processed by the ISM. These may include:

1) DOM requests/replies
2) remote ISM requests/replies
3) application requests/replies

ISM

This is the Integration Services Manager portion of the Framework Programmable Platform for the Advanced Software Development Workstation (FPP/ASDW) system. It is composed of the following components: encoder, plan builder, plan executor, and decoder.

library functional plan from DOM

This is a copy of a functional plan stored in the DOM.

messages from local environment

This is communication to the controlling process from the spawned process or the operating system.

messages to the local environment

This is communication from the controlling process to the spawned process or the operating system.

output messages

This is a collection of messages produced by the ISM. These may include:

1) DOM requests
2) remote ISM requests
3) application requests/replies

plan builder

This is the component of the ISM responsible for generating an executable plan that, when executed, will satisfy a service request.

plan executor

This is the portion of the ISM responsible for executing an executable plan.

plan step

This is a single instruction from an executable plan.

plan step results

These are the results of the execution of a plan step.

planning control commands

These are commands from applications, remote ISMs, or other external entities that influence the workings of the ISM. These may include: 1) commands to halt processing of the current process, 2) commands to view/modify/replan plans, and 3) commands to synchronize activities associated with manual portions of a plan.

planning data

This is data needed in the plan generation activities. This may include:

1) tool service advertisements for the generation of functional plans
2) tool protocol and contract specification information used in the generation of executable plans

remote ISM results

These are the results of remote ISM service executions.

request for library functional plan from DOM

This is a request to the DOM for a particular functional plan that it may have stored i2 its library.

request for new plan

This is a request from the plan executor to the plan builder for a new plan. This is generated when, for some reason, the plan executor is unable to carry out the instructions of an executable plan.

request for planning data

This is a data request that may be sent to the DOM or to the application. Some examples include:

Requests for tool information for building functional and executable plans.

request for service data

This is a request generated by the plan processor for information needed in the execution of a plan step.

request to archive plan

This is a request sent to the DOM to store a newly created functional plan into the library of plans.

request to log failed service request

This is generated when a functional plan could not be developed to satisfy a service request. The unfulfilled service request will be stored in a log for future reference.

request to remote ISM

This request is generated by the plan executor when a step in the plan it is executing cannot be carried out on the local machine. The request is routed via the encoder to the8 appropriate ISM and the results of the request are returned to the plan executor (via th5 decoder).

requested service

This is the decoded request for a service.

service data

This is data used by the plan executor to carry out the execution of a plan.

service to plan

This is a service that needs to be planned. This is generated when a functional plan could not be found in the plan library to fulfill the service request.