Automated Workflow for Engineering Services [1,2]

Ruth Bergman, Chester S. Borden and Silvino Zendejas
Jet Propulsion Laboratory/California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91009
818-354-1112
ruth@alum.mit.edu, Chester.Borden@jpl.nasa.gov, silvino.c.zendejas@jpl.nasa.gov

Abstract— In the future, users will request engineering services automatically much like we request stock quotes and weather reports today. Rather than requesting an engineering service directly from an engineer, a user will request the service from an automated server, which can be accessed from a web browser or other computer applications. The engineer's duties will shift to maintaining and enhancing the automated services, and performing expert-level analysis. This paradigm shift is beneficial to the engineer who is currently burdened with providing repetitive services. It is also beneficial to the users, who will receive a faster response from an automated server than a person can provide. Unfortunately, it is vastly more difficult to expose engineering services than services such as stock quotes and weather services. This paper argues that service technology is sufficiently mature to provide many engineering services reliably and securely. It also illustrates a methodology for creating engineering services using the engineering applications that today's engineers use to provide their services. Once automated engineering services are available, they can be used in conjunction with a workflow management system to transform engineering processes in the same way business processes have been transformed in the last decade. This paper describes a prototype of an automated workflow system, developed by the Jet Propulsion Laboratory, that provides navigation and telecommunication services.

TABLE OF CONTENTS

1. INTRODUCTION
2. RELATED WORK
3. ENGINEERING SERVICES TODAY
4. ENGINEERING SERVICES TOMORROW
5. HOW TO PROVIDE ENGINEERING SERVICES
6. THE TELECOMMUNICATIONS PREDICTS SERVER PROTOTYPE
7. CONCLUSIONS

1. INTRODUCTION

Workflow management systems have become prevalent in the enterprise. Large organizations provide many of their administrative tasks through workflow management systems [1,2,3]. Some types of enterprises use workflow management in their business processes, for example on-line retailers and brokerage houses. While converting the processes of large engineering enterprises to take advantage of automatic workflow is nontrivial, this paper describes how it can be done. As a proof of concept, we have implemented a prototype of an automated workflow system for preparation of engineering data for telecommunication between the ground antennas of NASA's Deep Space Network (DSN) and the Jet Propulsion Laboratory's (JPL) spacecrafts.

At the heart of the solution is a paradigm shift from present day processes where engineering services are provided directly by the engineer to a process that relies on automated, computerized engineering services. Consider the daily activities of a stockbroker in the 1970's. He spent a large portion of his day informing clients of the current status of their portfolios. Today clients get such information from automated telephone or web services. This brokerage service is fully automated with the information available anytime, anywhere. The broker can spend his time in his field of expertise - performing stock analysis and portfolio management. We have the same goal in mind for the engineering experts. Today, a navigation expert at JPL spends too much time on the large number of requests for routine navigational computations he receives. In the future, we want to see these requests submitted to an automated system. The navigation expert should spend his day analyzing complex navigational problems, or enhancing the capabilities of the automated service.

The engineering domain is too broad to admit a single automation solution. There are many engineering activities at JPL with which engineers are involved. Among these are mission design, spacecraft design, communication, performance analysis, verification, etc. Many of these activities require creativity and problem-solving skills, which we are unlikely to automate even in the distant future. Other activities, however, involve well-understood computations that already exist in the form of an algorithm or a series executable program. It is this second set of activities that we would like to provide as automated

0-7803-7231-X/01/$10.00/© 2002 IEEE

[2] This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

services. Engineers perform these computations routinely, usually by invoking existing programs. The primary advantage in the engineer's direct control of the computation is verification of correctness - an engineer can look over the results quickly and determine if there is an obvious problem. This ability is extremely important in this environment where the cost of a mistake is overwhelming. We compensate for this capability by validating the software and using the software routinely. The advantages of providing the computation as an automated service include relieving the engineer from such routine, repetitive activities, improving response time to the user and simplifying organizational procedures.

We believe, from research of automated services in other domains and prototyping experience in our engineering domain, that the state of the art in Internet/Intranet based service technology provides sufficient reliability, availability and security. There are, however, technical challenges that are unique to large engineering enterprises. Engineering enterprises typically use heterogeneous computing platforms, including Unix workstations, PC's, Macintoshes. Interoperability is one of the key issues in groupware, with few offerings supporting multiple computing platforms [4,5]. There is great variation in the software tools that engineers use. They have confidence in their tool set and are not typically willing to switch tools to maintain enterprise-wide consistency. It is unusual for engineering organizations to mandate use of specific engineering tools, except, perhaps, for tools with prohibitive cost, such as CAD tools. Therefore, it is difficult to support the interoperability required by workflow systems. We overcome the issues of heterogeneous and distributed computing environments through the use of a distributed component communications protocol. There are several choices for remote object communications: CORBA [6], Java RMI[7] and SOAP[8] are the most notable. We elected to use the Simple Object Access Protocol (SOAP).

Our strategy, toward enabling a service-oriented paradigm in engineering, is to provide routine services through Application Programming Interfaces (API's). The main contribution of this paper is an in-depth analysis of what comprises an automated service, and how to create a service and expose an API from an existing engineering application. These existing engineering applications, unfortunately, do not naturally lend themselves to re-use, but, on the other hand, they are not traditional "legacy" applications. These programs are like legacy applications in that they do not adhere to modern software development practices. They are unlike legacy applications in that they are not static, they are dynamic and continue to be developed and improved today. These programs typically embody a great deal of engineering know-how and are too costly to re-write. They require quite a bit of environment setup to operate. Often, they are not adequately documented, provide many tunable parameters and, as such, require an expert operator to produce useful results. We overcome these issues with wrapper code that exposes the functionality of the application in a modern, modular, re-usable API. These services, then, become available through a browser interface or even to another program.

With this vision of service-oriented workflow for engineering in mind, we come to the practical question of how we make these services available. We identify general requirements for an application service that we can integrate into a workflow management system. These requirements are:

- Modularity
- Concise API
- Data availability
- Location independence
- Monitor status and exception handling
- Reliability
- Documentation
- Asynchronicity
- Scalability

This paper discusses these requirements at length, especially as they pertain to the integration of existing applications. Having identified these requirements, we describe the process of providing a service given an existing application. The process is comprised of five steps

(1) Define the services
(2) Analyze input and output data
(3) Create database support
(4) Write the service proxy, which invokes the legacy application
(5) Write the service monitor, which controls and monitors the execution of the service

In the paper we describe each of these steps. The ideal set of solutions is presented, but implementing this ideal often implies a complete re-write of the application. We then present a set of compromise solutions that allow us to attain the requirements above in a reasonable time frame and cost.

We have developed a prototype that follows this strategy. The prototype provides data services as well as application services. Data services include information about missions, schedules, ground antennas and more. The data to support the prototype resides in an Oracle database, but the data source is transparent to the user. At the time of writing, application services are provided for view period generation and some telecommunications link predictions. (A view period is the time span that a ground antenna can view an object in space and telecommunications link predictions provide configuration information for equipment to support communication between ground antenna and spacecraft.) This paper describes the methodology used to provide these services, the process of exposing an API from a legacy application, and the workflow management for executing requested services.

This paper is organized as follows. Section 2 provides a (non-comprehensive) overview of fields of work related to the work presented in this paper. Section 3 discusses the state of engineering practices today and the technical challenges for automation. Section 4 provides a vision of the engineering services of tomorrow. Section 5 delves into the issues related to providing application services, in particular details pertaining to providing these services using legacy, engineering applications. Section 6 describes our Telecommunication Predicts Server Prototype. The intention of this paper is to provide a comprehensive discussion of the service-oriented vision for engineering. Thus some of the detail regarding the workflow management system implemented for the prototype is omitted from this paper and can be found in [9]. We conclude and present lessons learned in section 7.

2. RELATED WORK

This paper describes a workflow management system. Many workflow management packages exist and more are in development [1, 2, 3, 10, 11]. Whereas most currently available workflow management systems are tied to a specific computing platform, with most systems running on PC's, our system is platform independent utilizing Java, SQL and XML [12]. Today's workflow management systems are monolithic centralized programs. Our workflow management is data-driven with decentralized control. Much of the workflow logic is embedded in a database as data, procedures, functions and triggers. This approach has the advantage that we can modify the most aspects of the workflow logic while the system is online and without affecting service availability. For an extensive discussion of this approach to workflow see [9].

Workflow management systems are often discussed in the literature under the umbrella of Computer Assisted Cooperative Working and Groupware [4,5]. These resources provide excellent discussion of issues the enterprise faces when adopting collaboration tools.

Our system uses a distributed component architecture. In particular we are providing distributed services; an area that has received considerable attention in the last decade [13]. Two main issues are frequently cited: security and communication. Security is very important for our system, and we use standard solutions such as firewall, authentication and access control. See [14] to learn more about security related concerns.

Several excellent solutions exist to the distributed object communication problem. The most notable solutions are the Common Object Request Broker Architecture (CORBA) [6], Java's Remote Method Invocation [7], Microsoft's .NET Architecture [15] (which replaces Distributed COM (DCOM)), XML Remote Procedure Call (XML-RPC) [16], and the Simple Object Access Protocol (SOAP) [8]. A comprehensive comparison among these architectures is beyond the scope of this paper, see [17] for a very good analysis. We select SOAP which we feel suits our requirements for platform independence, firewall penetration and evolution into the future.

Last, our problem includes legacy application integration, which provides some unique challenges. This area of work is prevalent in the e-Business community. See [18, 19] for some discussion of application integration.

3. ENGINEERING SERVICES TODAY

Engineers are doing mission critical work. The effect of an error in their calculation can be catastrophic to the tune of as much as hundreds of millions of dollars. The organization places a great deal of trust in its engineers, as well as a great deal of blame when failure occurs. It is of no wonder, therefore, that the work environment in the organization is cautious. Engineering organizations are cautious about changing processes, cautious about taking a new approach to a problem, and especially cautious about software.

The engineering services required in the organization are, therefore, provided as they have been for several decades directly by the engineer. At JPL, for example, we have several categories of engineering experts. We have mechanical engineers who design instruments and spacecraft, navigation engineers who plan mission trajectories, telecommunication engineers who analyze requirements for communication between spacecraft and ground antenna, etc. Every mission at JPL uses experts in each of these areas. Some missions have a dedicated expert in each area, but many smaller missions cannot afford to keep an expert for every category and use the services of a pool of experts.

When planning a mission, one person, typically, handles all the requests in his category of expertise. The expert almost always uses tools to perform the necessary computations. Generally speaking his expertise is essential for setting up the problem. For example, quite a bit of expert telecommunication knowledge and mission-specific knowledge is required to set up the parameters for a telecommunication computation. After the initial setup most computations become routine because the analysis of the problem has been done and the all but a few problem parameters are unchanged. Consider, for example, a change in mission trajectory. This change necessitates re-computing telecommunication capabilities, which really implies re-running the tool with the new trajectory. Despite the fact that this re-computation amounts to changing an input and hitting a go button, today it is the expert that performs this function.

The primary technical reason this duty falls to the expert is

that he is not only an expert in his field; he is an expert at running the legacy software tool. The majority of the tools we use to do engineering computation are tools that evolved in-house by expert engineers. These tools compute very sophisticated mathematical models, but lack many of the usability features common in today's software tools. They may be complicated to install, have complex command line interfaces, are subject to core dumps, and provide little or no error information. In short, one needs to be an expert to use the tool. The bottom line is that, today, experts are providing all engineering services manually.

understood and are not handled by existing tools. For example, the Mars program is comprised of several spacecraft, which will be on Mars and in orbit around Mars simultaneously. This scenario creates a new challenge for our communications network, which until now communicated with one spacecraft at a time (except occasionally in a manually intensive setting). The engineering tools we use must be enhanced with such new capabilities. Only the engineering expert, who is already over-extended, can do this work. Thus, changing the paradigm by which we currently provide engineering services is critical to the long-term success of our missions.

What is the	**Local Temp**	**Temp in Mumbai**	**Add Temp to my app**
1970's	Weather report	Call a friend	Manual update
1980's	Telephone Service	Call a friend	Manual update
1990's	Check weather site	Check weather site	Manual update
2001	Updated in portal	Updated in portal	Use Temp RPC service

Table 1: Temperature services

What is the	**Price of MSFT**	**Value of my portfolio**	**Add Stock price to my app**
1970's	Call stock broker	Call stock broker	Manual update
1980's	Television stock- ticker	Call stock broker	Manual update
1990's	Check stock site	Check brokerage house site	Manual update
2001	Updated in portal	Updated in portal	Use Stock RPC service

Table 2: Brokerage services

Where is Galileo?	**JPL user**	**NASA user**	**Add Galileo location to my app**
1970's	Call Galileo NAV or DSN	Call Galileo NAV or DSN	Manual update
1980's	E-mail Galileo NAV or DSN	E-mail Galileo NAV or DSN	Manual update
1990's	E-mail Galileo NAV or DSN	E-mail Galileo NAV or DSN	Manual update
2001	E-mail Galileo NAV or DSN	E-mail Galileo NAV or DSN	Manual update

Table 3: Navigation Services at JPL

4. Engineering Services Tomorrow

Unfortunately, organizations cannot afford to continue this engineering paradigm. At JPL, the number of missions is increasing and the engineers cannot keep up with the workload. They are spending a significant percentage of their time providing these routine services, and, consequently, less time on expert analysis and tool enhancements. It is vital to JPL that engineers utilize their expertise for these activities, because future missions will identify new problems that are not currently well understood and are not handled by existing tools.

To illustrate the style of services we are advocating let us examine several services most people use daily, namely stock brokerage services and weather services. Tables 1 and 2 describe the progress of temperature and brokerage services over the last several decades. Each of these services has undergone a metamorphosis. Where we once had to catch the weather report on television to find temperature information, we can now see up to the minute updates in our portal application. We can even get temperature information for most cities around the world

right in our portal application. Lastly, if we need to add temperature information to our own application, where we once had to maintain our own database with this information and manually update it, we can now use a temperature Remote Procedure Call (RPC) service such as [20]. Likewise stock brokerage services have progressed. Where our primary source for up to date stock quotes and portfolio summaries was the stockbroker, we can now see that information in our portal application. Likewise an RPC service exists, for example [21]. These services are not typically real-time, but delayed some minutes. This information fidelity is, however, sufficient for, perhaps, 95 percent of users. Not on the table but also available are stock trading services, which also suffice for most users. Thus these services have supplanted the frequent calls to the stockbroker. The stockbroker now deals with the problems where his expertise is best utilized: stock analysis and portfolio management.

There are, on the other hand, some issues that today's technology solves. The engineering community does not yet perceive Information Technology as providing secure, reliable and robust solutions. While these problems are continually under scrutiny, with better and better solutions available, we consider today's technology in this area sufficient for most applications. The largest unknown in providing engineering services in a secure and reliable way is how reliable the underlying computational tool is.

Table 3 describes the progress of a simple navigation service, namely the identification of the current location of the spacecraft Galileo, over the last several decades. As you see, whether a user is on the Galileo team, at JPL or outside of JPL, they must contact an engineer to obtain this information - even today! Adding the location of Galileo to an application still requires manual update (unless one chooses to add the complete suite of navigation programs to their application, which, in itself, requires expert understanding.

We believe that we can take advantage of the technology used to provide temperature and brokerage service to provide some engineering services. Our aim is not to replace the engineer, nor to provide a complete set of engineering services. Rather we want to substitute software, when applicable, for expert labor. In particular, where such services are already performed by manually invoking a software tool. We will consider ourselves successful if we can avoid manual intervention in 80 percent of the requests for service. These engineering services are extremely oversubscribed, and the engineer's workload is such that a response delay may be as high as two weeks. Our goal is to provide these automated services in a matter of minutes or hours, at most. The difference in procedure should be significant both to the user, in terms of faster response to the requests, and to the engineer, in terms of time available for analysis.

We do not delude ourselves. Providing engineering services is considerably more complex than providing temperature or stock brokerage services. Engineering services rely on more volumes of data and contain a great deal of specialized logic. In addition, the tools currently used to perform these computations are, at best, prototype implementations. They do not provide sufficient reliability or monitor and control capability. Providing these services is a *challenge*. We therefore cannot expose the full capability of the application as services, and we must work to improve reliability. These issues are discussed in detail in the next section.

5. How To Provide An Engineering Service

For the purpose of this paper, let us define an automated service is a service that is available over a network, such as the Internet or Intranet. A service operates on messages that contain requests, and include either document-oriented or procedure-oriented information.

This section describes how to expose an engineering service. We define the requirements for an automated service, problem areas for exposing services using existing applications and how we solve these problems. Section 5.2 discusses ideal solutions, which require too much effort for most applications. To reduce the level of effort, we have adopted with non-ideal but workable solutions, which we describe in section 5.3. These solutions allow us to attain faster results and avoid re-writing and re-validating the existing tools.

Requirements for Application Services

This section outlines requirements for services and contrasts these requirements with the state of engineering applications. The requirements we identify for application services are: modularity, a concise Application Programming Interface (API), seamless access to application data, location transparency for both the user location and application location, ability to monitor and control the application, reliability with respect to abnormal program termination or communication fault and, lastly, comprehensive documentation. The remainder of this section discusses these requirements and where these engineering applications do not adhere to the requirements.

Modularity— According to the Cambridge Dictionaries Online a service is a system or organization that provides for a basic public need [22]. Our interpretation is one service, one need. We want to provide basic functionalities as services. These services can be combined to provide complex functionality (as we do with our workflow system). The services should be modular so services can be added to the overall system or replaced by new services without disruption to the user. This modularity will allow

our system to remain up-to-date with the latest formats and technologies. This paradigm is very much the extension of a modular programming paradigm to a large distributed system, and appears to be the direction software development is headed.

The engineering applications we are working with do not support this modular service paradigm well. These applications evolved from historical software. In many cases, the application started as a Fortran program or Matlab model, which the engineer used to do some calculations. Over decades, more capabilities were added, the toolset grew, and the tool fell into multiple hands until it became an application that performs a critical function for the organization. Unfortunately, this evolutionary software development resulted in monolithic applications. Typically an application bundles multiple services in one package or provides a partial service so it must be used in conjunction with several other applications. These applications need some cleaning-up to support our one service, one need requirement.

Concise API— A successful service must be intuitive to the user. A concise API is the key to making the service intuitive. The service must provide exactly the "right" set of methods, which take the natural input parameters and return the desired output parameters. For this reason, temperature and stock quote services are very successful. The API is indeed intuitive: method *stockquote*, input *ticker symbol*, output *dollar value.*

The engineering applications we use to provide services, on the other hand, typically accept many parameters, options and file inputs. The entire "data base" that the application requires is passed in as input. This comprehensive set of inputs is akin to an application that provides temperature service and has two inputs. The first input is the name of the city for the location whose temperature the user would like to find out. The second input is a file containing the temperature information for every zip code. The temperature application converts the city name to a zip code and looks up the temperature in the input file. For the engineering application, separating those inputs that might be considered part of the database from those that are salient for the service is difficult. Even more difficult is analyzing the data to determine an appropriate model for storing this data. If one succeeds in defining the data model, the application has to be adapted to use the new form of the data.

An additional challenge arises from the use patterns for these applications. Discussions with users typically indicate that the automation and ease-of-use we are suggesting sounds great, but users still want to be able to run special cases manually. The API that supports complete manual functionality for these applications is far from concise and user-friendly. On the other hand, the engineering expert has a legitimate need for manual invocation. We describe our solution to this dual challenge in the following sections.

Data Availability— Whatever data the application uses should be readily available. This requirement seems obvious in modern applications. Naturally, a stock quote service must have a database with stock prices to support it. The service provider is tasked with ensuring that the data is up to date and accessible. As we have already discussed, it is not so with many engineering applications. The application data is passed in with other parameters. The task of managing the application data, at present, falls to the user.

Location Independence— The service should be available to a user at any time and from anywhere (subject to security constraints). The API should make all details of the service transparent to the user. These details include the application platform, the directory location of the application and the location of application data.

Deploying engineering services in a location independent manner is difficult because the applications are platform dependent and often contain some file system directory structure dependence as well. Most of these applications have no chance of running on any platform other than their development platform (usually some flavor of Unix). The applications may be in Fortran, they may include Matlab models or they may combine multiple programming languages. Directory dependence may stem from including a Matlab model, which the program uses as part of the calculation, or from some hard-wired application data input. This practice, unfortunately, is common in these applications. We tackle this challenge by setting up a distributed architecture. The user makes requests. The request is handled by a workflow management system which invokes services that execute elsewhere using SOAP RPC[17].

Monitor Status and Exception Handling— We require services to make status information available. This status information includes both progress information and termination status. We want to monitor the progress of the application. The preferred progress information is in the form of percent of work completed. We only require a flag indicating that the application has started, is running or has completed. Upon termination the application should report whether it has succeeded or failed. The service also should maintain a log, which the service administrator can browse when a failure occurs. We require this log as opposed to comprehensive exception information for simplicity. To extract detailed exception information or progress in the form of percent of work completed from an engineering application would require significant re-writing of the application. The weaker requirements for progress and termination status indicators constitute an acceptable

compromise, between information requirements and development time, which enables seamless operation.

Reliability— We require reliability at many different levels in our services. First, the application must result in correct outputs. Second, the workflow must be reliable. Third, the system must recover from hardware and network failures. Last, the service must be available 24 hours, 7 days a week.

For engineering applications these reliability requirements translate to a level of testing to which these applications have never been subjected. The current mode of operation for these applications has assumed an engineer looks over the results. To remove this check, we must validate the application and its inputs. Achieving reliable workflow with these applications presents a further challenge, since they were not designed for an automated environment or for multiple users. This reliability requirement drives some of the requirements we have already discussed, such as location independence and status monitoring. The other two reliability requirements are handled with the standard industry solutions to the problem of service availability by using redundant systems and failure recovery. See [23] for an excellent overview of the state of the art in this area.

Documentation— Naturally, the service must be documented comprehensively so that a user can make use of the service with ease. We, furthermore, require API documentation so that developers of other applications or services can reuse this service in their development.

These requirements suffice to expose a service on the network using existing service protocols, such as SOAP or CORBA. The messages correspond to the requested service and include the methods and parameters defined by the API. Requirements such as modularity, location independence, data availability, etc. allow the service to exist as a stand-alone entity on the network. Last, a variety of clients can use the service due to the API and documentation.

Asynchronicity— Services should support an asynchronous delivery of results. Some of the calculations can take many minutes or even hours to run. We can not expect the user to be waiting at the web browser for the request to complete.

Scalability— A useful and responsive service is likely to attract use. Clearly the service must be able to handle multiple concurrent requests and be able to scale in performance and capacity depending on the needs.

We have made some allusions to problem areas for creating an engineering service. Let us now discuss in detail how to convert an engineering application into a service.

Providing an Engineering Service the Right Way

This section describes the preferred way to provide a service. We describe the steps one must go through to expose an application as a service. Here we outline how to do everything "the right way". If we follow these steps we will have an ideal service. Namely, it will be a service that is easy to use, easy to add to a workflow system and easy to adapt to future needs.

Define the Services— In the ideal case, the application we use would provide a service has a well-defined API. The services then correspond to this API. That is, each of the top-level methods provided by the application is exposed as a service and the parameters correspond to the method parameters. In such cases where fulfilling the function that the service provides requires performing an additional function, the additional function is itself used as a service. For example, compute spacecraft view periods requires trajectory computation, so we should provide a trajectory service as well as a view period service and the view period service should use the trajectory service. As discussed in the previous section, the engineering applications we use, unfortunately, never provide a programmatic API. We invoke these applications through a cumbersome manual interface. Providing services the right way, implies a significant re-write of, at least, the input and output portions of the application. We need to define the services and API's that provide these services. Applications that invoke other applications internally should be broken up to modular pieces that use and provide services. This is a significant undertaking.

Analyze Input and Output Data— As part of providing the API we must define the input and output parameters to the services the application provides. Since we use an existing application the input and output parameters for the application are already defined. Ideally, we can simply propagate these inputs and outputs to the service API. If we had, for example, a view period generation program that has as input a *spacecraft object, a list of antenna objects, start time and end time,* and as output a *list of view period objects*, we could define the service API with ease. It would simply correspond to the application input and output. The view period generation application we actually have has as input a spacecraft identifier, a start time, an end time and about twenty files. The following is a partial list of the files inputs

- Spacecraft trajectory file
- Planetary trajectory files
- Ground antenna trajectory file
- Planetary constants, such as masses
- Ground antenna parameters, such as a horizon mask
- Model of the Earth
- Parameter list of view period generation events
- Parameter list for view period post processing

The outputs of this application are

- View period file

- Output status and log files

The problem then is how to expose the intuitive API suggested above, when the underlying application requires so many more inputs.

The "right" solution, is to

(1) Thoroughly analyze the application inputs.
(2) Identify the supporting application data, in contrast with the invocation specific data.
(3) Modify the application interface to expose an object oriented API instead of input files containing parameter lists. This API contains the invocation specific data.
(4) Organize the supporting data in a database.
(5) Modify the application to use the data services from the database rather than an input file base for supporting data.
(6) Last, but not least, modify the organizational processes to maintain the data in the database, i.e. keep it up to date. In the view period example, most of the data is static, such as the planetary constants. Other data changes on a daily, weekly or monthly basis. There is a process for updating the data, but the current process is comprised of updating files and directories.

Create Database Support— In our design the workflow is closely tied to a database. We maintain the queue of work orders including status information and input values in the database. Likewise, the description of the tasks our system is capable of completing resides in the database for persistent storage. When we define a new service we need to add information about the tasks performed by this service to the database. This information includes the name of the tasks, and the full set of inputs required to complete each task.

At first glance adding database support appears straightforward, especially after we have thoroughly analyzed the parameters above. It is, indeed, straightforward if the API is simple and the input parameters are simple. If, on the other hand, the types of the parameters are complex we must define every type of parameter the service uses.

In practice, doing this right is likely to drive quite a bit of database development. For (a simple) example we may have a parameter input that is integral, and may take a value within some known range. We may have many similar parameters that take values in different ranges. We may have a string, such as uplink frequency band, that can take one of an enumerated set of values, e.g. "X", "Ka", etc. View period events are an example of complex types. All this domain specific information has to be represented in the database to support the service.

The advantage of domain knowledge in the database is that we can do quite a bit of validation on requests before they are submitted to the server. From the user's perspective, early input verifiation is obviously useful, because submitting a request to the workflow system does not provide immediate response. With up-front validation, the request can be corrected immediately, rather than after the user has waited for the workflow system. In addition, we can use this domain knowledge to drive automation of the system. If we have good information about the parameters a task requires we can automatically generate valid work requests and increase the reliability of the service.

Write the Service Proxy— The service proxy is the back end provider of the service. We call this server a proxy because it is a stand-in for the application that provides the service in the workflow design. This application can be a Java program or a legacy program, or any other program. The only requirement is that it provide the following interface methods

- execute - The execute method is called to for a new request. All input parameters are passed in to this method.
- getWorkerEvent - request the status of the request
- abort - the abort method is called to stop the processing of the request
- testProxy - this method can be used to verify that the proxy server is up and running

In our design the proxy is a stand-alone application. It does not know about other architectural components, including the database. The application may, however, use other services, such as application services or data services. In addition to making a request, via the execute method, the proxy can be queried for status of the request using the getWorkerEvent method. We can stop execution of the request using the abort method. Lastly, we can check if the proxy is running using the testProxy method.

When the application exists a priori and provides a clean API, the service proxy simply invokes the appropriate methods of the API. Since we are doing things the "right" way, we have already provided such a clean API for our application. The proxy must contend with multiple requests. If the application can handle multiple requests internally we have no problem. Otherwise, the proxy has to manage multiple instances of the application.

Our design exposes this service proxy as a SOAP service on a web server. Thus, the above API methods are remotely accessible to any SOAP enabled client.

Write the Service Monitor— The service monitor is the middle layer component that fulfils a work order. It prepares the request for the service proxy, invokes the service proxy, monitors the progress of request execution on

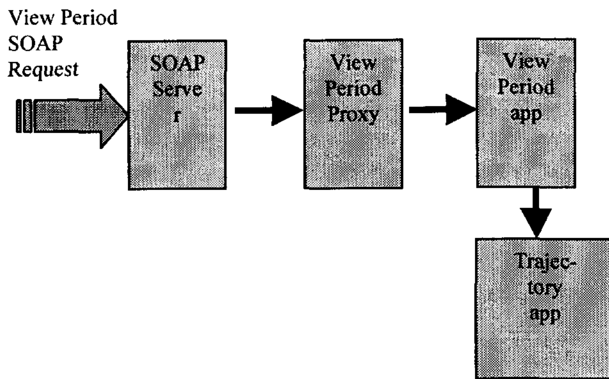

Figure 1a: The diagram above shows the design of the view period server. In this design, the trajectory application is invoked directly by the view period application.

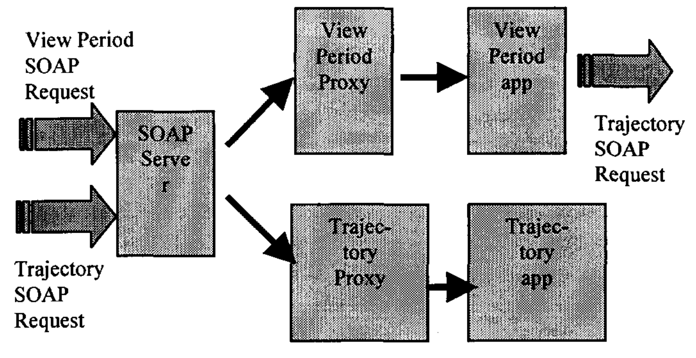

Figure 1b: The diagram above shows a modular design for the view period and trajectory applications. In this design, the view period application obtains trajectory information via a SOAP request to the SOAP server.

the proxy, updates the status of the work order and the worker proxy in the database, and manages the output data. The monitor is fully database aware. In fact, in our design, the monitor runs in the Java server inside the Oracle database we use. The monitor would typically contain most of our application specific business logic as well as some workflow logic, specifically with respect to monitoring and status.

Providing An Engineering Service the Practical Way

The most common phrase we utter is "for now". Unfortunately, the applications we are working with are so complex, use so much data, and involve so much domain knowledge that it is impossible to provide services "the right way". Instead, we make compromises and provide these services "the practical way". The practical path to providing a service given an engineering application is as follows:

Define the Services— Since the engineering application is not modular and does not provide a programmatic API, providing the service the right way becomes impossible. What we do instead is make sure we provide the functionality of the application as a service. That is, we need to do a good job of defining the services. This allows us to define the API that the application should provide. We then wrap the application, or possible set of applications to provide this API. The intent is that we can provide this functionality to the user. In addition, the functionality from the user's perspective will never change. That is, the API we expose to the user will remain consistent. The underlying application and the wrapper implementation and even the support data can change, but these changes will be transparent to the user.

For example, we now provide a view period generation service. The view period application needs quite a bit of trajectory information, which it obtains by using, internally, a set of trajectory tools. In the future we would like to provide trajectory services as one entity. We would then replace the current view period service with a new service based or an application that uses these trajectory services, instead of the current application that uses the trajectory tools directly. The modular design of our service system will enable us to make this change transparently to users and other services. Figure 1 depicts a pictorial comparison of these two scenarios.

Defining the right set of services requires quite a bit of domain understanding. The crux of the problem is how users prefer to invoke the application. It is not the case that a single command line application corresponds to one service. Some services may be made up of multiple applications. For example, generating some telecommunication link predictions is a two-step process beginning with generating a mission configuration from the desired sequence of events and equipment and then generating the predictions for this configuration. Other applications may be broken up into multiple services. For example, the functionality provided by the trajectory toolkit we use (SPICE Toolkit) provides many services including locations of planets, moons and spacecrafts, ranges between bodies and more. Lastly, we do not change the applications to use other services "for now".

Our system provides a manual interface as well as an automated one. We, therefore, provide a two-layer API for these applications. The lower layer API exposes all the details of the application, the full set of input parameters, file inputs and options. This provides the capability for manual invocation (for example through a detailed user-interface). The higher level API, which we call the Application Services API exposes a subset of the functionality according to an analysis of use patterns. This API exposes only a handful of parameters. Other

application parameters and options are either defaulted or retrieved from a database. This API is the commonly used one. It is also the API that is exposed as a RPC service. The engineering expert uses the lower-level API for advanced analysis.

Analyze Input and Output Data— Modifying the input and output data as described above leads to significant rewriting of the application in addition to a major process change. These changes will not happen in a timely fashion. Again, we compromise. While we keep in mind our goal of database integration, we build a service that is still file based, but supporting files are stored in the database. We provide an object API, but underneath this API is an API comprised of a lengthy parameter list. Thus we are able to support manual invocation as well as typical uses.

The "practical" solution follows the same sequence as the "right" solution, but each step is different

(1) Thoroughly analyze the application inputs.
(2) Identify the supporting application data files, in contrast with the invocation specific data files.
(3) a. Expose the manual , parameter based interface
    a. Define the file inputs (some of these will be supporting files) and output.
    b. Define the parameter inputs. All the invocation specific data should be provided as parameter inputs, even if the underlying application uses a file input for those parameters.

    b. Expose the application service, object based interface
    - Define the object inputs and outputs.
    - Identify the supporting files, which will be in the database.
    - Identify the invocation specific parameters that are not part of the object inputs. Provide defaults for these parameters.
(4) Organize the supporting data in a database. The practical solution for data management is to store the supporting data files in the database with reasonable organization. (For example, for view period generation the spacecraft trajectory is stored with project data, the planetary trajectory is stored with astrodynamic data, etc.)
(5) Since we cannot re-write the application to use supporting data directly from the database we provide supporting data to the two application interfaces as follows
    a. For the manual interface
    - The user specifies file inputs. These files must be in the database.

    b. For the application service interface
    - this interface uses data services to find appropriate supporting files in the database given the object inputs and required file types.

    A middle layer, which we call the work monitor, takes care of copying the supporting files from the database onto the application server for use by the manual interface.
(6) The process change for updating the data is unlikely to change in the near future. Instead we put utilities in place to grab the latest supporting data files from their release location and insert them in the database.

Create Database Support— We do not, at present, feel equal to the task of analyzing the full set of parameter types for the task. For the telecommunications domain, there is vast domain knowledge that should be in the database. We compromise greatly on the parameter type definition, and postpone most of this hard analysis and development until later. We currently support simple types: integer, float, string , url, date, boolean and file. Of the above types only the file type has been analyzed to the point that we specify various file types. Files are critical because, as we have already discussed, they are the primary data source for these applications.

The result of these shortcuts is that we can check a priori whether an input file is of the correct type. This level of type checking for files is limited, because a file may have the correct type, but may not, for example, be for the time period of interest. We can also check that the input data is of the right type, but not that it falls into an appropriate range or that it is one of the enumerated set values. For strings, we cannot check proper format. These limitations imply that validating correctness of requests falls to the user, and most input errors will not be caught until the application fails.

These shortcuts also limit our ability to automate our system. For most parameters we do not have enough information to populate a work request. The remaining options are to get the input data from a user, or to assign defaults for most parameters. Thus, the services we provide in an automated setting expose partial functionality of the application.

Write the Service Proxy— Our applications do not provide a clean API. In addition, they use files for both inputs and outputs. Thus, a large part of the proxy's work is wrapping the existing application to provide the desired API. The application's use of local directories and files complicates the API; the files must be copied from the database to the local file system. In addition, supporting multiple threads is complicated because we have to ensure that one thread does not use or overwrite files for another thread. Lastly, the output of the application is usually a file, which is written to

the file system. We must grab that output file and save it.

The solution to these problems in our design is that the service monitor loads the input file onto the service proxy's file system. After application execution, the monitor grabs the resulting output file from the proxy's file system and inserts it to the database or provides it to the user in the manner the user requested it (we support mail and ftp). Cleaning up the local file system must wait until the output file has been assimilated. We, therefore, add a cleanup method to the proxy interface. The proxy does not clean up the local environment until this cleanup request is made by the monitor.

The service proxy, then implements the following interface

- execute - The execute method is called to for a new request. All input parameters are passed in to this method.
- getWorkerEvent - requests the worker status
- cleanup - the cleanup method signals the proxy to clean its local environment from the respective program execution.
- abort - the abort method is called to stop the worker application
- testProxy - this method can be used to verify that the proxy server is up and running

The wrapper implementation that the proxy uses to invoke the application is implemented in a thread aware class. The implementation of the wrapper class deals with the many domain specific and application specific details involved in wrapping. Some of the issues the wrapper must address follow.

- Converting parameters from user specified type and format to those the application expects.
- Creating parameter input files.
- Creating and removing temporary files.
- Creating and removing temporary directories.
- Invoking the legacy application from the wrapper.
- Providing status updates.
- Maintaining an accurate and comprehensive log on the part of the proxy.
- Aborting work that is currently executing.
- Cleaning up when the request is completed.

While a great deal of the logic for invoking the application resides in the back-end wrapper, some logic resides in the service monitor that invokes the wrapper. The decision as to where to put some domain or application logic must be made by the implementer on a case-by-case basis. Some of the decisions are guided by the parameter analysis. Some issues are practical, such as separating the back-end application from the database.

Write the Service Monitor— Our underlying application is still file based, and, at the same time, we want the proxy to be location independent and ignorant of the database. It, therefore, falls on the service monitor to "magically" place the input files in the proper location for the proxy and to fetch the output file from its location on the proxy server. In addition, since the monitor is between the user, on the front end, and the proxy, on the back end, we can use it to do some of the parameter conversion. As stated above, it is up to the implementer to decide how to divide the logic between the middle layer (monitor) and the back layer (proxy).

Of the other functions performed by the service monitor: invoking the service proxy, monitoring the progress of request execution on the proxy and updating the status of the work order and the worker proxy in the database, only monitoring progress depends on the application. Our existing applications do not provide progress information while the application is executing. We would like to be able to report progress in terms of time to completion or percent done. We do not have this capability at present. Thus, we focus more on status than progress. Status information is more limited. We can tell if the application has been initiated, is in progress, has succeeded or has failed. For now, we provide this limited progress information.

A service created using the above steps is not perfect. We would expect to revisit some issues on this service, to review the shortcuts we have made one by one and redesign them the right way. This service may be more difficult to reuse, integrate and adapt than a service created the right way. This service is, however, functional, and functionality is our first goal.

6. The Telecommunications Predicts Server Prototype

This section describes a prototype that provides telecommunication predicts services. We briefly discuss the architecture of the prototype, the services provided by the prototype, the workflow management, and how we automate processes using known services.

The Architecture

A long list of requirements influences our architectural design. We have a requirement for 7/24 availability. Our system must be fault tolerant for hardware, networking and software failures. We must support users on any computing platform. (Unix, Windows and Macintoshes are all pervasive at JPL.) The legacy applications we use to

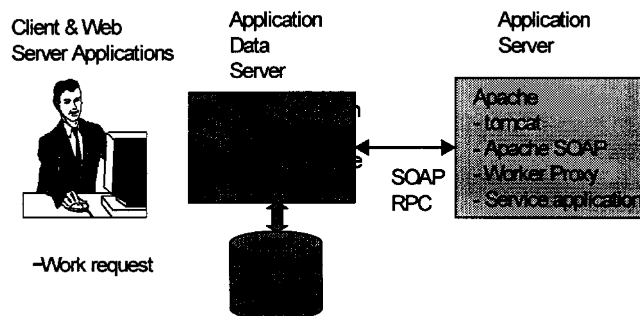

Figure 2: High-level architecture of the telecommunications

provide services typically execute on the Unix platform. Our services, both data services and application services, must be available both within and outside of a firewall. Naturally, some of the data our system carries is sensitive and proper security is required. Based on all these requirements, the architecture for our prototype is distributed, multi-platform and multi-tiered.

The architecture for the prototype is presented graphically in figure 2. On the back end is the application server, which invokes the server application. From the architectural point of view, for this server, any computing platform is acceptable. We are currently using Apache SOAP, which runs under the Apache Tomcat servlet container. As this software is all Java, we can use any platform where Java is supported. Usually we are restricted by the legacy application, most of which are native for some flavor of Unix. In the prototype, the back end server is a Sun Solaris machine. Security issues are fairly simple for the back-end server since it is behind a firewall and is only accessed by the workflow manager, which is also behind the firewall.

The middle tier in our architecture contains the bulk of our data and business logic. We are using an Oracle database, which provides the reliability, availability and security we need for our data. Our application server also resides in the middle tier. In fact, we are using the Oracle provided Java server and application server, which run in the database. (Note that the term application server is overloaded here. This application server does not refer to one of the services we provide. Rather, it is the industry standard name for a server that can execute some software upon request.) This combination of tools should provide good reliability and security infrastructure, and keep our efforts in this area to a minimum. The middle tier in our architecture is behind the firewall. The data and application services it provides are accessible by users (and other applications) within and outside the firewall. Thus, the bulk of emphasis on security in our system is in the middle tier. As part of our business logic the middle tier hosts the workflow manager. This module contains the SOAP client that communicates with the SOAP server to request a service.

The data and application services we provide in the middle tier are available as either web applications or SOAP RPC. Thus we support two types of clients on the front end: human user with a browser and any SOAP enabled client.

VGR2 Spacecraft

Astrodynamics Run Control Station Elevation Events Station Parameters

Spacecraft/Object Ephemeris: vgr2_v2.bsp(0.000) OR upload Browse...

Antenna Name: (Non Selected) DSS13 DSS15 DSS16
Complex: SPC 10 SPC 40 SPC 60 D GARCIA
Antenna Type: 0 10 26 34
Body ID: 10 Sun 199 Mercury 299 Venus 399 Earth 301 Moon

Events

Earth Apoapsis · Occ Moon · Occ Sun · Earth Shadow
Wrap Limit · Elevation Limit · Azimuth · SI Rt Axis 1
SI Rt Axis 2 · Periapsis · Umbra · Rise · Set

Mask Angle: 6.0 · Save results into Database

Viewperiod Output Location: e-Mail

(note: if selecting FTP, use "servername/ftppath/yourusername:yourpassword";
if selecting e-Mail, use "youremailaddress";
if selecting smdb, use "smdbdirecty")

Generate Viewperiod

Since web browsers are available for all platforms, and SOAP is a platform independent protocol this architecture does not restrict the platform of the client.

The Services

At present the prototype provides view period generation and some telecommunication link prediction services. View periods are the spans of time that a spacecraft is "visible" for a particular antenna. View period generation is a service that is particularly well suited to automation. This service is used during mission design to ascertain the level of telecommunication service the mission can expect. The mission designer needs the spacecraft view periods to forecast ground antenna availability for this mission. It is often the case that the mission can be modified to obtain better service. For example, the launch date can change. Some trajectory changes are possible. The mission designer will typically explore many scenarios before committing to a design. For each scenario the designer must obtain view periods, and the fidelity of these view periods need not be very high. (Five minute accuracy suffices, typically, for scheduling.) The convenience and fast service paradigm we provide are a perfect fit with these requirements.
The low level manual interface to these services is accessible via a web browser. Figure 3 shows the main screen for the view period server interface. As you can see the user has considerable control in selecting the parameters for view period generation, including selection of events, occulting bodies and supporting data. Most of the input information is defaulted based on database information after the spacecraft is selected and start and end dates are set. The user may then override any of the database default values. The outputs are in the form of a file. The user can request the output location of the resulting file as a Uniform Resource Locator (URL). Both FTP and mail URLs are available.

There is also a high level API available programmatically as a SOAP RPC. The generate view periods function of this API takes as input the spacecraft object, an optional trajectory file, start and end dates, optional ground antenna objects, and an optional output URL. Like the manual interface, the result of this method will be to place the output file in the requested URL. This API uses the same infrastructure to create a complete work order for the view period generation service using the database defaults for the given inputs. It also uses the same infrastructure to control the workflow for this work order once it has been queued. Thus we have achieved the two-level API we desired. The naïve user is able to get results with minimal inputs and little domain understanding. The expert user has fine-grained control over the execution of the application.

Other services available in our prototype generate the telecommunication link predictions. These predictions are specific equipment settings for a known telecommunication

link. For example, we know that Mars Global Surveyor will downlink telemetry to the 70m antenna in Goldstone on December 21, 2001 between 7pm and 3am, what equipment setting are necessary to support this activity. For these telecommunications link predictions we have similar manual and high-level APIs available programmatically in the prototype. Work to design the browser Graphical User Interface (GUI) is ongoing.

We are extending the services provided by the prototypes to include additional radiometric predictions. The DSN needs these predictions to enable acquisition and tracking of the spacecraft and to specify the equipment configuration necessary for telecommunication with a spacecraft. For example, for a given spacecraft, trajectory and antenna view period, where should the antenna be pointing to lock onto the spacecraft and what is the optimal frequency for downlink communication.

In addition, we have identified trajectory functions to be well suited to exposing as services. There is a large community of people, in and outside of JPL, who use trajectory information in a variety of ways. There is a toolkit, called the SPICE toolkit, which is often used to obtain trajectory information. This toolkit provides excellent functionality. The drawbacks of using it are that every user must install a copy on his machine. Every user is then responsible for maintenance and upgrades. Although the toolkit is well documented, it requires considerable investment in time to learn. We believe we can expose many of the functions provided by the toolkit as a trajectory service, and this service will have considerable impact for many JPL processes (from mission design to mission operations to educational outreach). This service remains, however, future work.

The Workflow Module

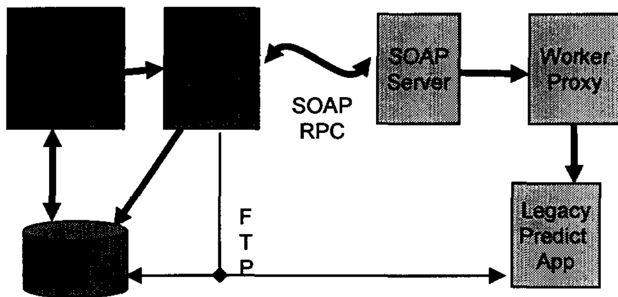

Figure 4: Design of the workflow management module.

The design of the workflow management module is depicted in figure 4. Let us briefly overview this design. Very few of the design details are included in this discussion. For a detailed discussion on the workflow management system see [9].

The workflow management in the prototype manages the requests for services, which we call work orders. A work order is queued in a work order queue, which we store in the database. When the work order is ready for execution the work dispatcher attempts to find a worker proxy for the requested work. If the work dispatcher finds a worker proxy it instantiates a work monitor to monitor the progress of the work. The work monitor prepares the data for the proxy, transfers files if necessary, and makes the SOAP call to the proxy. The proxy receives the request and invokes the legacy application. The monitor repeatedly checks the progress of the work, using SOAP calls, and updates the status of the work order in the database. The monitor is also in charge of retrieving the output file from the worker proxy and putting it in the user-requested location (database, email or ftp).

In our design the workflow module resides in the database server. We take advantage of this collocation to create a new breed of workflow management system. The work dispatcher, although it is drawn in Figure 4 as one unit, is actually a distributed collection of Java methods, stored procedures and triggers. In addition, information about the available services and worker proxies is stored in the database. The dynamic nature of this design allows us to update the workflow logic in a localized manner. We need not make a grand re-release when we change a component or add a new one. We do not need to interrupt the workflow to make a change to the system. In practical terms, this implies improved usability. The disadvantage of this type of distributed logic is that the workflow is difficult to analyze and debug. We have attempted to reduce this challenge with a carefully designed state transition system for work order and worker status.

7. Conclusions

This paper explores the feasibility of an automated engineering service paradigm. Under this paradigm, engineering services are analogous to today's web services. The service can be invoked directly by a user using a browser. The service can also be re-used in another program through an API. We assert that the technology enabling such automated services is sufficiently mature to adapt to the engineering domain.

The bulk of the paper addresses the challenge of exposing engineering services in practice. We immediately see that engineering applications are inherently more complex than typical web-enabled applications. These applications require considerable domain specific input. Furthermore, the only reasonable path to exposing automated engineering services is to adapt existing applications to the new paradigm. These programs, unfortunately, are ill suited for an automated setting. They have strong dependencies on the environment, they expect a large number of parameter inputs, and they operate on files. This paper provides a

methodology for exposing such existing engineering applications as automated services. This methodology converts the existing application to an application with the following requirements: modularity, location independence, concise API, data availability, reliability, documentation and monitor status and exception handling.

We have implemented the methodology described in this paper in the telecommunications domain. We built a prototype which exposes several applications as web services, including view period generation and telecommunication link predictions. This prototype, in addition to providing useful functionality in its own right, demonstrates the use of automated services in a web setting and application re-use as part of a workflow management system.

REFERENCES

[1] MQSeries Family. IBM. http://www-4.ibm.com/software/ts/mqseries/about/.

[2] ActiveEnterprise. TIBCO. http://www.tibco.com/products/enterprise.html.

[3] Oracle Workflow. Oracle. http://otn.oracle.com/products/integration/workflow/workflow_fov.html.

[4] Usability First. Web site. http://www.usabilityfirst.com/groupware/.

[5] Michel Beaudouin-Lafon. Computer Supported Co-Operative Work. Trends in Software, 7. , John Wiley & Son, 1999.

[6] Ben-Natan, Ron. CORBA: A Guide to Common Object Request Broker Architecture. McGraw-Hill, 1995.

[7] Sun Corporation. Java™ Remote Method Invocation (RMI). http://java.sun.com/j2se/1.4/docs/guide/rmi/.

[8] Skonnar, Aaron. SOAP: The Simple Object Access Protocol. MSDN Magazine. January, 2000. http://www.microsoft.com/mind/0100/soap/soap.asp.

[9] Bergman, Ruth and Zendejas, Silvino C. A Decentralized Approach to Automatic Workflow in Dynamic and Distributed Environments. In preparation.

[10] The Workflow Management Coalition. http://www.wfmc.org/.

[11] Fraunhofer-Institute for Software Engineering and Systems Engineering. http://www.do.isst.fhg.de/workflow/produkte/index_e.html.

[12] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml/.

[13] Matti Aarno Hiltunen. Configurable Fault-Tolerant Distributed Services. PhD Dissertation. University of Arizona. 1996.

[14] Lincoln D. Stein and John N. Stewart. The World Wide Web Security FAQ. http://www.w3.org/Security/Faq/www-security-faq.html.

[15] Microsoft Corporation. Microsoft .NET. http://www.microsoft.com/net/.

[16] Userland. XML-RPC. http://www.xmlrpc.com/.

[17] Kenn Scribner and Mark Stiver. Understanding SOAP: The Authoritative Solution. Sams, 2000.

[18] Attachmate Corporation. Approaches to Enterprise Application Integration Involving Legacy Applications. http://www.attachmate.com/article/0,1012,3163_1_3857,00.html. 2000.

[19] David S. Linthicum. EAI Application Integration *Exposed.* Software Magazine, February/March 2000. http://www.softwaremag.com/archive/2000feb/EAI.html.

[20] Xmethods. Weather - Temperature Service. http://xmethods.com/detail.html?id=8.

[21] Xmethods. Delayed Stock Quote Service. http://xmethods.com/detail.html?id=2.

[22] Cambridge Dictionaries Online. http://dictionary.cambridge.org/define.asp.

[23] IMEX Research. High Availability Overview. http://www.highavailabilitycenter.com/overview.html.

Ruth Bergman is a Senior Computer and Information Scientist in the Mission and Systems Architecture section at JPL. She is working on the TMOD network simplification project among others. From 1996 through 1998, Dr. Bergman was a member of the technical staff at the MIT Lincoln Laboratory in the Advanced Systems and Sensors Group, where she developed artificial intelligence approaches for Infra-Red seeker technology in ballistic missile defense. She holds a Ph.D. in Computer Science from MIT, where she was a member of the Artificial

Intelligence Laboratory and performed research in the area of autonomous agents and machine learning.

Chester Borden is the Supervisor of the Information and Mission Operations Architecture Group in the Mission and Systems Architecture Section at the Jet Propulsion Laboratory. He manages the software development effort for the Service Preparation Subsystem for the Interplanetary Network and Information Services Directorate. Mr. Borden has worked on and managed numerous technology and software development tasks at JPL. He has an MS in Operations Research from Cal State Northridge and a BS in Mathematics from UCLA.

Silvino Zendejas is a Senior Computer and Information Scientist in the Mission and Systems Architecture section at JPL. He is the Cognizant Design Engineer for the Service Data Management Assembly of the Service Preparation Subsystem. He holds an MS in Computer Engineering from the University of Southern California and a BS in Civil Engineering from California Polytechnic University in San Luis Obispo.