A Framework-Based Approach to Science Software Development[1]

Steve Larson
Stephen Watson
Jet Propulsion Laboratory
4800 Oak Grove Dr.
Pasadena, CA 91109
Steve.Larson@jpl.nasa.gov
Stephen.H.Watson@jpl.nasa.gov
Kalyani Rengarajan
Raytheon ITSS
299 N. Euclid Avenue
Pasadena, CA 91101
Krengarajan@sdsio.jpl.nasa.gov

Abstract— The Tropospheric Emission Spectrometer (TES) is a Fourier transform spectrometer slated for launch in December 2002. Its six-year mission to provide a 3-dimensional map of ozone and its precursors is part of NASA's Earth Observation System (EOS). TES is expected to produce approximately 8.4 TB of raw data and an additional 25 TB of processed data each year. The data are to be archived and distributed by NASA's EOS Data and Information System (EOSDIS). Processing this data requires the development of a large, robust software system capable of automated operations, sufficiently maintainable to support ongoing revision of the processing algorithms. The target platform for the TES science software provides a rich set of job control functions. However, the changeable nature of the underlying science algorithms mandates a high degree of maintainability in the science software. The long time period over which the software is developed and maintained suggests a framework-based approach to system development. We describe plans for the development of an application framework to support TES, including requirements, architecture, technical and management issues.

TABLE OF CONTENTS

1. INTRODUCTION

The Tropospheric Emission Spectrometer (TES) is a Fourier Transform Spectrometer scheduled to fly on the Earth Observing System (EOS) Chemistry spacecraft in December 2002. The TES project is managed by NASA's Jet Propulsion Laboratory (JPL). In its six-year mission, the TES instrument will provide the world's first three-dimensional global data set of tropospheric ozone and its precursors. Data processing activities are planned to continue through December 2011.

TES is a first-of-a-kind instrument in terms of its combined performance, resolution and operational capabilities. The instrument will produce over 8TB of raw data, and an additional 25 TB of processed data each year. Evolution of the algorithms and the production software that implements them, is expected to continue throughout the mission.

Data processing will be performed by a production facility located in Pasadena, CA. The TES Data Processing Facility (DPF) will be designed, built and operated by Raytheon Information Technology and Science Systems under contract to JPL. The TES project team is responsible for developing and delivering processing software to the production facility, and for providing operational support for data quality monitoring and anomaly investigation. The data products produced in the DPF will be delivered electronically to the NASA Langley Research Center (LaRC) Distributed Active Archive Center (DAAC), an element of the EOS Data and Information System (EOSDIS). See Figure 1 for a context diagram of the DPF.

Description of Target Environment

The production environment is a highly automated batch-oriented system that traces its heritage back to the Goddard Space Flight Center (GSFC) Version 0 DAAC. This system has been ported to a number of facilities since its original development, adding additional functionality in the process. More recent instances of the system include the Vegetation

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

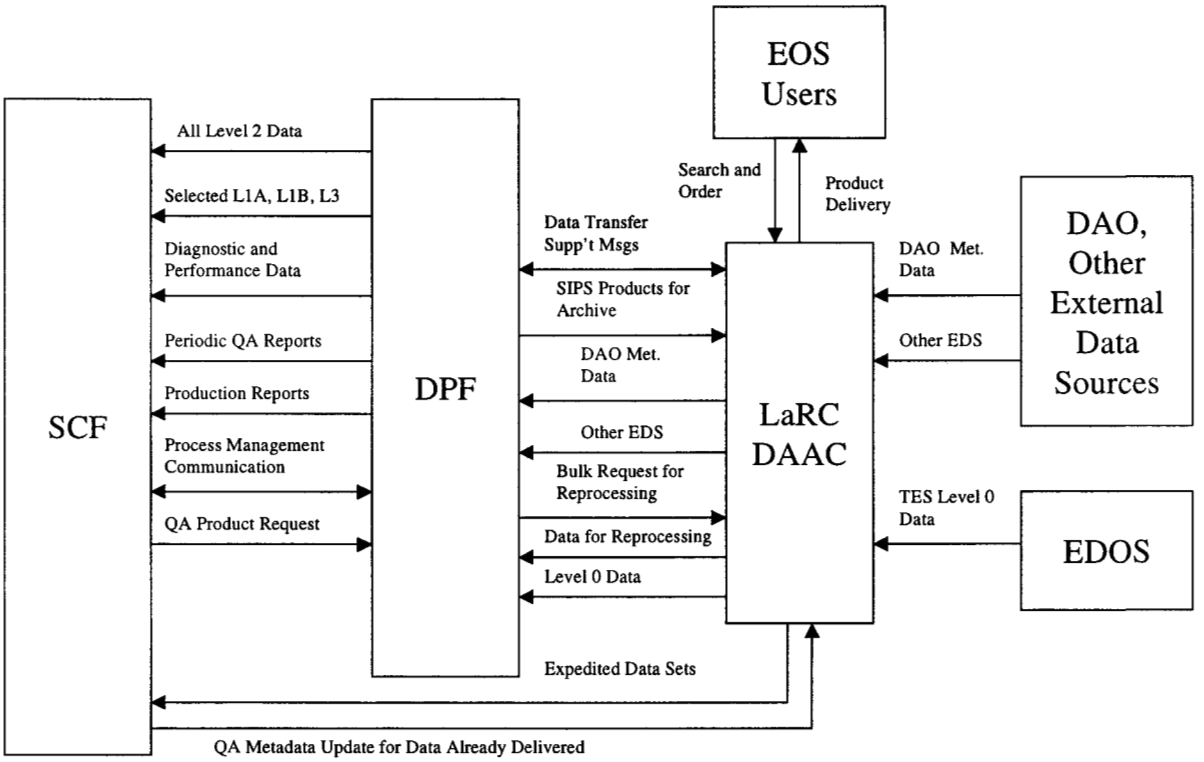

Figure 1. The TES Data Processing Facility Context

Canopy LIDAR (VCL) Data Center, and the IceSat data processing facility.

The production environment is a UNIX-based planning and scheduling system. This system provides all data management, and job planning, scheduling and execution services required to operate the facility. The science software, which is the main subject of this paper, need only be concerned with the science-driven aspects of the processing. The production environment places few code-level restrictions on delivered science software. However, the plan to operate the facility 24x7 with only prime shift staffing places significant requirements for automation and reliability on the science software. This software must fail gracefully when fatal conditions are encountered, and provide a great deal of automated data quality checking and process information capture.

Scientific research, software development, and operational support activities will be based in the Science Computing Facility (SCF) located at JPL. Past experience with remote sensing projects suggests that unforeseen events in instrument calibration and on-orbit algorithm performance will necessitate the rapid development of many new applications. The ability to respond to these events in a timely manner and within budget is a major goal of the system design.

The SCF represents the second major platform the science software is required to run on. The SCF is currently based on the Sun/Solaris platform. Although the DPF is also planned for this platform, changes in technology and market conditions may lead to the use of another hardware and operating system platform. We expect to require the science software to support two separate platforms in the DPF and SCF environments.

As part of the EOS program, TES is required to deliver all data products to the EOSDIS. This places additional requirements on the system in terms of data formats and metadata production. In order to reduce cost, TES is considering the use of a number of third party software packages for inclusion on the delivered system. These include NASA-supplied toolkit routines for data format, metadata, ephemeris, geolocation and mapping functions, and freely available libraries for numerical processing, linear algebra support, and command line processing. The Framework must provide an encapsulation of these third party products.

Motivation

With the current negative pressure on budgets within the federal government expected to continue for the foreseeable future, the TES project was very interested in ways to further reduce development costs, and to ensure the ability to

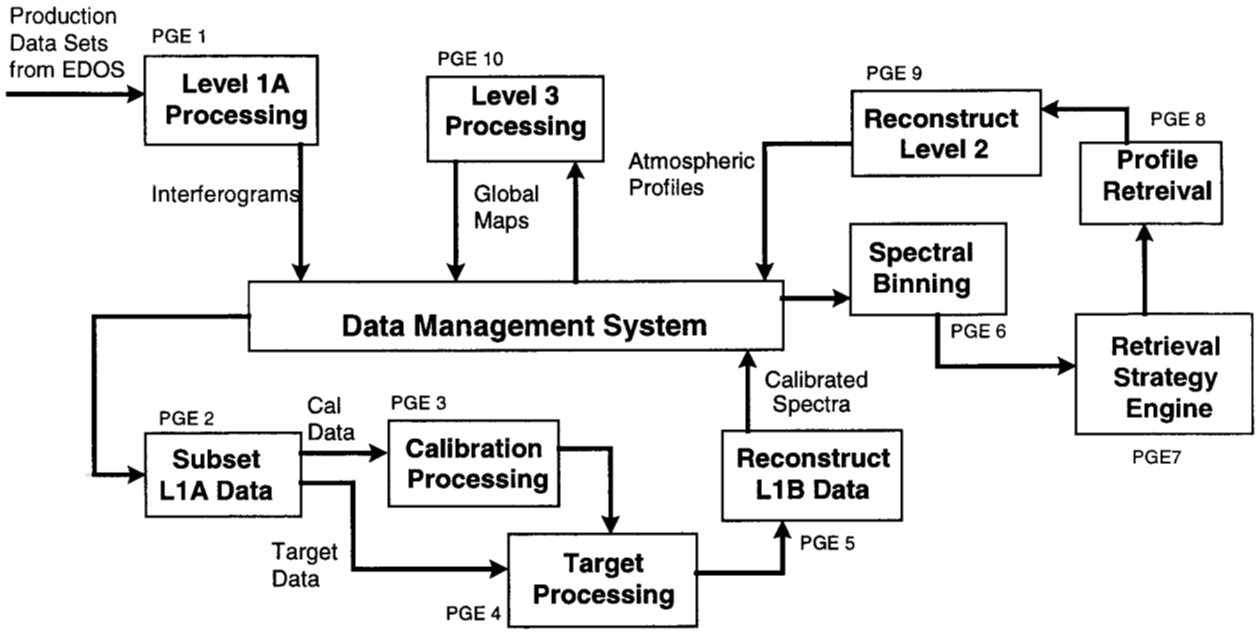

Figure 2. Data Flow Through TES Science Data Processing System

provide new functionality to the science team, especially later in the project when budget pressures are expected to become more severe.

Frameworks have emerged in the last fifteen years as a means of increasing reuse and productivity. The term "framework" has many interpretations in the software engineering community. We base our use of the term on Rogers' definition [4]: A framework is, "a partially completed software application that is intended to be customized to completion." The scope of what we consider a framework includes reusable design and code components.

The decision to adopt a framework-based approach was a strategic one, intended to fulfill the need for reduced cost, and a more robust, maintainable system. Along with the decision to develop a framework, the project adopted an object-oriented (OO) design approach and selected the C++ language. The framework decision was thus part of an overall strategy to leverage OO technology and modern approaches to reuse and development.

The choice of a framework-based approach is consistent with the findings of a 1996 NASA workshop on reuse [5]. One of the important conclusions of the NASA group was that reuse should focus on domain-specific problems. Our effort is narrower in scope than the NASA concept (we focus our efforts on a single project, rather than an organization- or enterprise-wide product line), though the same principles are employed. NASA is currently formulating a large-scale Earth System Modeling Framework (ESMF) development [6] as part of the High Performance Computing and Communications Program [7]. We are considering proposing the framework discussed here as a basis for developing the ESMF.

Considerations

Developing a framework is considerably harder than developing a one-off system. We consider it to be equivalent to a *programming systems product*, as defined by Brooks [8]. Although it has been around for over a decade, the C++ language is still undergoing evolution, especially in the area of compilers and templates. The framework approach, though it has been around for over a decade, is still not well established in the specialized field of science software development. The project team was aware of other science software projects that had developed common subroutine libraries, but we could find no precedent for the magnitude of commitment developing a framework represented.

In the program management area, there were risks associated with recruiting and training staff in the underlying technologies. Risks and uncertainties notwithstanding, adopting a framework-based approach appeared to be the best way to proceed.

Notation

The framework design discussed here is based in part on the design patterns of Gamma, et al [3]. Where we refer to a design pattern we will use the name found in Gamma, et al, with an initial capital. Likewise, we distinguish between discussion of frameworks in general, and the specific Framework proposed for the TES project by using an initial capital where the latter connotation is implied. The UML is used in class and sequence diagrams.

2. SYSTEM OVERVIEW

The current design concept for the science data processing system (SDPS) is shown in Figure 2. Each box labeled "PGE *n*" represents a product generation executable (PGE). A PGE is the smallest unit of processing that is independently planned and scheduled within the production system. A PGE may have subunits of execution that include staging, processing and destaging steps. The processing flow in the SCF (not shown) is more complex, but less formalized. It is expected to comprise a few dozen applications in the year 2002 time frame. As discussed above, the need to develop SCF applications rapidly is a major driver in our decision to develop a framework. However, we will focus much of our discussion on the production system, as it is better defined, and highlights the major features of the Framework satisfactorily.

Table 1. Data Product Levels

Level 1A	Raw instrument data in reconstructed interferogram format, with instrument state data and geolocation data appended.
Level 1B	Calibrated spectra at full spatial and spectral resolution.
Level 2	Vertical temperature and species abundance profiles.
Level 3	Global maps of Level 2 data. One set of maps is created for every four-day global survey cycle.

The total system size is expected to be roughly 500,000 lines of code, including comments. We include comments in system sizing, as we have found that they represent a significant portion of the development effort, as well as comprising one of the most important system documentation records.

The production system design is based on an EOS-mandated demarcation of processing into levels. The basic unit of processing is a four-day data set (also known as a global survey cycle (GSC)) received every eight days (the remaining four days are reserved for special science observations that will not be discussed here).

The four levels of data product are shown in Table 1. Due to the nature of the Level 1B processing, it is necessary to wait until the entire data set is received before processing begins. The data are transferred to the DPF in two-hour chunks and are processed through the Level 1A PGE upon receipt. Once all of the Level 1A inputs for a survey cycle are ready, a subsetting PGE reorganizes the data according to a set of predetermined instrument parameters and geographical locations. Calibration data are processed into an intermediate form and used as input to a set of parallel target processing PGEs, each working on a different subset of the survey cycle data. There is a one-to-one correspondence between calibration PGEs and target processing PGEs. In a survey cycle, we expect 1,536 instances of each to be run. The calibrated target data are reassembled *en masse* into their final product form by a "reconstruction" PGE.

Level 2 processing follows a similar pattern, but instead of reorganizing the data as the first step, the initial PGE creates an index into the calibrated data. The index data are used to define input lists for instances of the retrieval PGE. As with Level 1B, the main processing PGE at this level is designed to operate in data parallelism. Due to the much higher CPU loading at Level 2, the basic unit of input data per PGE is smaller, leading to a total of 4,253 instances of the retrieval PGE per survey cycle. Level 2 processing concludes with a reconstruction PGE. Unlike Level 1B, there is a separate instance of this PGE for every retrieval PGE instance.

A single Level 3 mapping PGE is run after all Level 2 products have been created.

At each processing level the granularity of data is different, and the load in terms of memory, CPU and disk space differs. Memory and CPU are particularly important commodities in the Level 2 processing, which requires more than 100 times the processing power of Levels 1A and 3. Level 2 also uses a large (≈500GB) database of precomputed absorption coefficients to reduce CPU requirements. However, the retrieval PGEs consume this table data at a rate of roughly 30GB every second, placing a severe strain on the I/O capacity of the hardware. To mitigate the situation, it is intended that the retrieval PGE will employ a tunable size (nominally 30GB) shared memory segment. The indexing PGE provides information used to order job execution such that turnover of this memory segment is minimized, thereby reducing the I/O requirements on the hardware.

An important consideration in our Framework design is the preliminary nature of the above described system design. Changes in the scientific algorithms and knowledge gained while in orbit may necessitate significant modifications of this design. The Framework must enable the development team to rapidly reorganize the configuration of functional elements.

The Framework is expected to support development of this system by providing the functionality shown in Table 2. One of the challenges in framework development is the lack of a complete set of requirements at the outset. In the sections to follow, we discuss our understanding of Framework requirements in further detail, as well as an approach to development that we believe will mitigate the risks this lack of knowledge implies.

Expectations for Reuse

The proposed Framework is intended primarily for reuse within the TES project. Although we will strive to make the design as generic as possible, the scope of the effort will be

Table 2. Summary of Framework Requirements

PGE Infrastructure An application skeleton for a PGE executable, including: command line parameter, environment variable, and user parameter processing; high-level input and output collection abstractions;
File I/O Support for I/O to/from all files in the system. Format support for HDF-EOS, HDF, ASCII and native binary file types. I/O support in terms of high-level data types (also framework supplied). Support for specific data product file organizations.
Metadata Support for EOS standard metadata output
Math Library Support Support for linear algebra, mathematical functions, specialized optimization and Fourier transforms, other functions.
Exception Handling Classes and conventions for handling exceptions.

constrained by resources and focus on project-specific requirements.

As mentioned in the previous section, we are considering proposing our Framework design as a basis for the ESMF. We believe our project requirements for file interfaces and application design are sufficiently similar to the more general ESMF case to make them a reasonable starting point for the ESMF design. Our data object model is more tightly constrained by the details of the TES instrument, and would likely require considerable effort to generalize to the ESMF or a similarly scoped system.

In our experience, software reuse is most easily achieved within the context of the originating group and their immediate colleagues. We would hope to be able to reuse our Framework on similar projects undertaken by our group, but as described below, the economic gains expected from the TES Framework are sufficient to justify its development regardless of reuse outside the project.

A reusable subroutine library was developed for the Atmospheric Emissions Spectrometer (AES), a project completed in 1993 as an airborne precursor to the TES instrument. The AES software system was considerably smaller and was not developed to the same level of standards and automation requirements of TES, but was nevertheless a substantial effort. The total system size was approximately 150 KLOC. A considerable portion of this code (roughly 105 KLOC) was already in existence when AES began, or was funded by other sources and not tracked as part of the AES development. Detailed development records were only kept for the low-level processing and utility code (a 26 KLOC portion consisting of two main programs and six utility programs). This portion also included the reusable library.

Of this 26 KLOC code development, about half was part of the library. Roughly three quarters of the entire AES development effort (a one work-year effort) went into producing the library and the two main programs.

Analysis of the final code sizes indicates that all of the library code was used at least twice, and 55% (comprising the data file support, file utilities, and log file code) was used in all programs. Our costs and schedule savings expectations for the TES Framework development (and subsequent rapid application development based on the Framework) have been extracted from this AES data. We found that if the 55% of code used in all programs had been developed from scratch for each use, the total system cost could have grown by as much as 180%. Looking only at the two main programs, we realized that reuse of the code would render cost savings on the order of 32% for that effort alone.

It is expected that the Framework will facilitate the development of test, data quality and other specialized tools for TES. An example of the kind of reuse we expect is the AES data extraction tool. This program required only 876 new lines of code, reusing nearly 7,000 lines of code and cutting the development time fifteen weeks to less than two. TES hopes to realize similar cost and schedule savings. If we succeed in encapsulating our higher level algorithms, we could realize similar cost savings through reuse of code, even for sophisticated applications with more complex algorithms than a simple extraction tool.

Based on the AES experience, we anticipate that developing reusable code from the outset will yield substantial economic benefits to TES.

3. Development Approach

The Framework will support the SDPS development by providing a generic application skeleton that will include classes for parameter handling, message logging, file input/output, exception handling and gathering process history. Applications will extend the generic program design from the Framework for specific algorithm implementations. It is important, therefore, for the Framework design to be sufficiently abstract to allow for a wide range of such extensions or derivations. Thus, the design of and interface to the Framework must be robust and generally agreed upon by all major subsystems, in order to avoid the unpleasant possibility of redesigning the Framework classes at later stages of the project.

Development Lifecycle

Framework development will be iterative and incremental, with requirements having the higher development risk being implemented first. There will be three major deliveries of the SDPS to the production facility before launch, spaced

approximately one year apart. Each external delivery will be built iteratively from a number of internal deliveries.

The Framework development must lead the applications development in order to allow sufficient time for applications developers to integrate the Framework code prior to their deliveries. The Framework API has been developed in draft form, and will be finalized within the next few months. Science applications developers have been heavily involved in the API development, and will be responsible for approving it prior to implementation.

The fundamental Framework components will be delivered in two major increments, spaced six months apart. Subsequent deliveries will add refinements and functionality not required early in the process. The incremental approach will allow us to validate requirements, and catch errors in design and programming earlier in the process than a monolithic development lifecycle. A key consideration is the choice of priorities for implementation within the Framework. This choice must be carefully made to ensure that deliveries provide functionality on a schedule that meets application developer's needs, and which takes into account the need to prototype the highest risk components as early as possible.

It is likely that additional requirements will be levied on the Framework by the various data processing elements, as algorithm development proceeds over a period of several years. Advances in hardware, software, numerical analysis and data convergence (multi-platform sensor data integration) will necessitate more sophisticated Framework support during the mission lifecycle. Consequently, Framework must be able to rapidly integrate new and varied requirements as they arise. By delivering many complete subsystems over a period of several years, usually with a very short (3 month development, 3 month integration and test) period between deliveries, the Framework will be able to accommodate most of the new requirements in a very rapid manner.

The benefits of delivering many complete subsystems extend to other areas as well. For example, the complex nature of the TES SDPS provides fertile ground for subtle errors to occur. By making small, incremental changes to Framework rather than delivering large, monolithic versions at greater intervals, we expect to minimize the number of undetected errors which would might remain hidden, and jeopardize actual processing where large amounts of data are involved. Usually, these are side effects rather than egregious bugs, and they normally result from incomplete or inconsistent interface specifications. Small, concise changes to the subsystem allow for more robust testing of the subsystem and its interactions with other subsystems specifically with regard to any new or modified interfaces. This is considerably more difficult with larger, less frequent deliveries.

The key features of the design will be prototyped to validate design feasibility. Multiple prototypes may be developed for the same algorithms using different techniques and different libraries. Similarly, there may be different algorithms to arrive at the desired result. These prototypes will be evaluated for performance, flexibility, portability etc., and the one that optimizes the goal will be chosen. The prototypes will also be used to derive further requirements that may not be covered under the initial functional or performance requirements.

System Integration and Testing

Framework testing will be driven by the requirements. The formal testing methodology for the developers and test group is currently being investigated, but will include automated tests with known inputs and outputs, demonstration and manual analysis of the data, and test reports. Because system integration and test will occur after each component is completed and passed through peer review, the interfaces between the different subsystems, the operational facility and other external interfaces will be continually evaluated for problems that may be resolved early on. In addition, it will enable us to evaluate the design and demonstrate to the customer that the system meets the standards of the required functionality. This allows the customer frequent opportunities to provide iterative feedback and redefinition of requirements for the next iteration. Besides component testing, each internal and external delivery will be put through a rigorous regression test. The results of this integration and testing will provide criteria not just for evaluating the Framework design, but for evaluating the requirements as well.

Integrated Team of Framework and Application Developers

The Framework team will consist of some core Framework developers involved in design, coding, testing and maintenance. From time to time some science application developers may also be included to augment their understanding of Framework and encourage their commitment to its use. This close collaboration of disciplines will enable Framework designers and developers to gain a solid understanding of the science data processing theories and algorithms necessary to deal with the changing requirements of these applications.

Development Tools

Requisite Pro from Rational Software Corp. is being used to track requirements. The Unified Modeling Language (UML) has been adopted as the modeling language for design. We are using Rational's Rose CASE tool to develop and document the design. In Rose, the subsystems are represented as packages and are broken up into finer level subpackages within each subsystem. Class, state and sequence diagrams will be used to define the data types, applications and their interaction. The roundtrip engineering features of Rose will be used to maintain consistency

between the design model and the developed code. Design and interface documents will be generated automatically from the Rose model using Rose scripts to ensure that documentation remains up to date. Evaluation of commercially available software for test coverage tracking is in progress.

Implementation

The Framework coding effort will be undertaken in a relatively traditional manner. Aside from the use of code skeleton generation from the Rose model, the software will be hand-coded. The use of generators (see, for example, [9]) to instantiate a program, or suite of programs, from a metamodel is a promising development in software engineering, but does not appear to be warranted in the present situation. The benefits of such an approach would include the ability to accommodate changes in the fundamental abstractions without labor-intensive manual recoding of all dependent classes. However, the basic abstractions assumed by the TES Framework are based on over 20 years of science software systems development. For purposes of the TES project, they are considered stable enough to allow the development to proceed without undue risk.

4. Software Design & Architecture

The Framework subsystem consists of several major components designed to reduce or eliminate the necessity for the scientific processing algorithms to have information on the underlying operating system and environment. These components fall into roughly three major categories: operating system-processing interfaces, algorithmic implementation and instantiation components, and utilities. Additionally, the Framework will encapsulate various toolkits, 3rd-party packages, and commercial off-the-shelf software, in what are called Foundation components. Figure 3 is an abstract view of the Framework and its major components, showing their scope and visibility relationships. In fact, many portions of the Framework could be used in a variety of applications. The domain-specific components tend to be focussed around file formats and external (project) requirements, such as data archiving and process logging.

Good, clear and developer-friendly documentation will be a key factor in Framework acceptance. The documentation will consist of interface definition, sample programs,

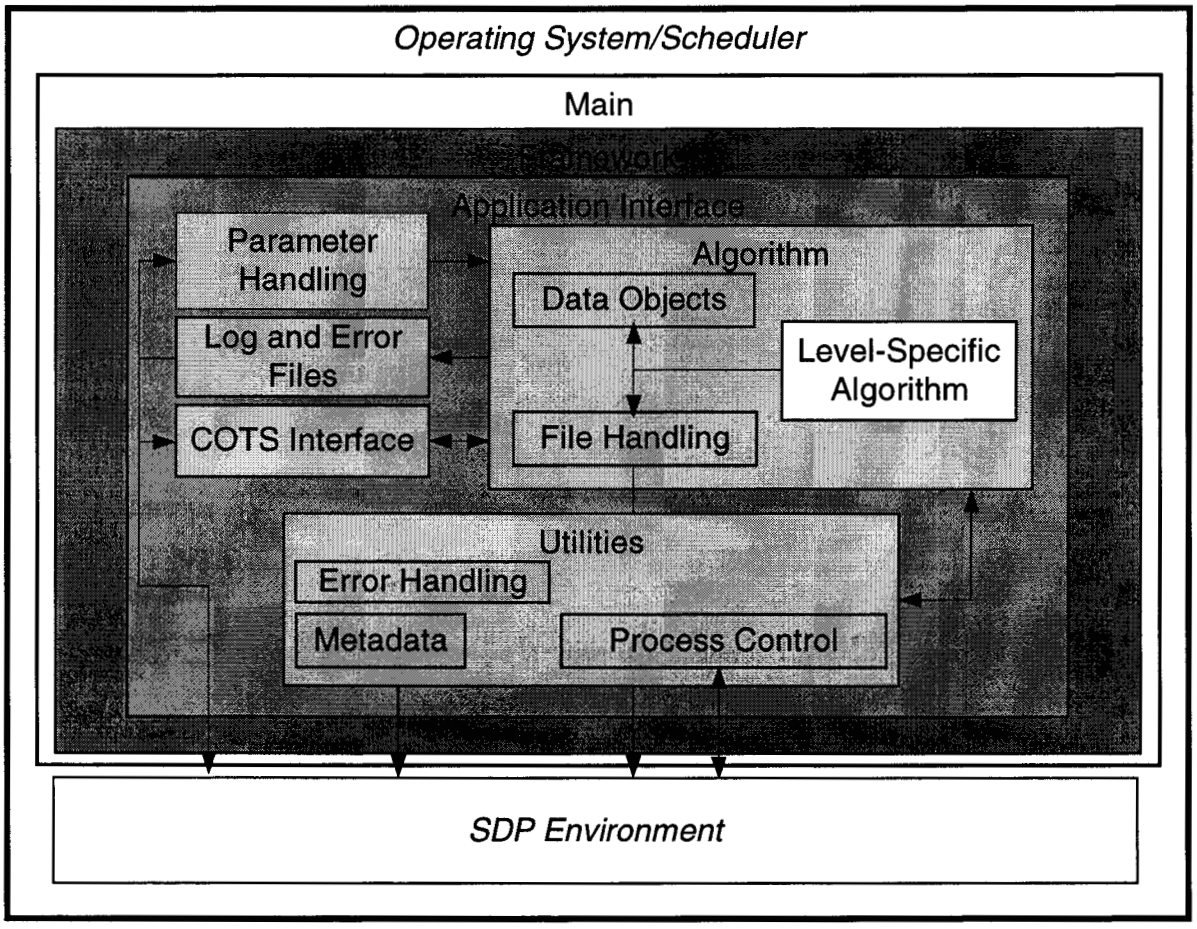

Figure 3. Framework Top-Level Diagram

instructions for building a science application from the Framework classes, skeleton program and information on the salient features of the Framework. The Framework architecture includes a top-level *Application Interface*, which is designed to encapsulate and hide all operating system details involved in initiating any specific processing algorithm from the underlying numerical code. This component includes objects specifically designed for handling parameters from a variety of sources and making them available to the algorithm itself. This top-level component provides the primary means for a PGE to instantiate and invoke Framework-based objects.

Within the Application Interface, the Framework contains an object called simply *Algorithm*, an abstract class which serves as a base class from which specific processing subsystems derive their own Algorithm classes. Primarily designed as a kind of template class for science processing teams to use for incorporating level-specific processing codes within the system, the Algorithm class and its derived process-specific objects also provide a high level of service to the algorithm, including instantiation of input and output files, error files, process logs, and so forth.

A large portion of the Framework is devoted to the notion of encapsulating file handling capabilities, in order to provide input/output functionality at a very high level abstraction, closely related to the nature of the domain-specific data. Due to the multi-platform nature of the SDPS and the requirements of the EOS program, data files are normally stored in a complex, although rich, format known as *Hierarchical Data Format* (HDF). Additionally, the EOS program has an extended form of HDF known, logically enough, as HDF-EOS. It is an express goal of the Framework to isolate entirely the underlying data file format from the algorithm developers, providing a clean, robust, broad interface to the data itself rather than to the underlying file structure. Additionally, support for standard binary and ASCII files must be provided, as well as extensibility to as-yet-undetermined additional file formats.

Clearly, then, the Framework must provide a full complement of *Logical Data Objects* appropriate to the science data processing being developed, including such items as interferograms, spectra, or collections of such items. It is intended that the data object hierarchy shall provide a means for the application developer to manipulate the data during processing, with an interface to the above-mentioned file handling components. Thus, the data objects must know how to read and write themselves via the file handling interface, while the developer remains unaware of the mechanics of these operations.

The Framework must provide mechanisms for handling some additional system-level operations, such as process control for multi-threaded algorithms, as well as future extensions for multi-processing.

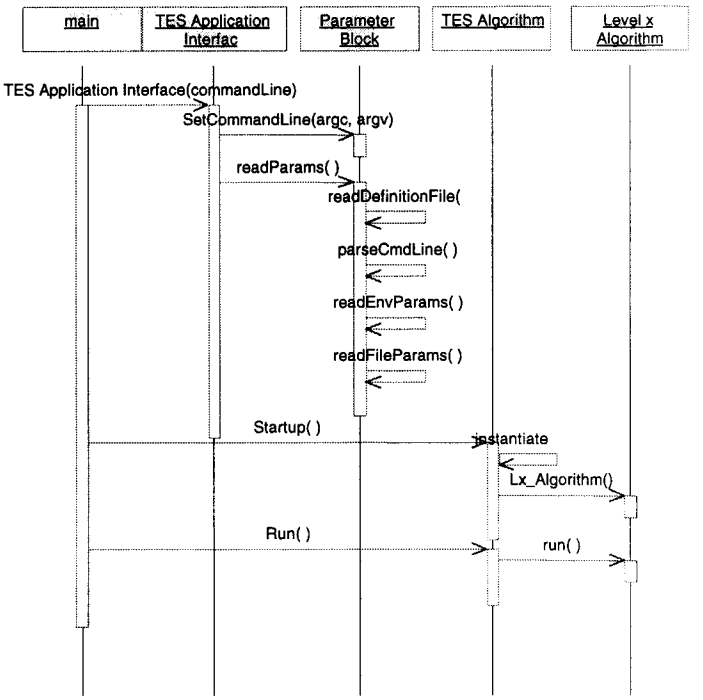

Figure 4. TAI Sequence Diagram

Finally, the Framework provides exception-handling capabilities, error logging facilities, status codes, Metadata handling, and other utilities. Many of these will, in fact, be utilizing the file handling objects for logging information about the processing status and/or errors. Again, it becomes clear that the file handling components is one of the most critical in the Framework.

In essence, the TES data processing Framework is a collection of Frameworks that interact with each other, the algorithm implementation, and the operating system and its environment.

TES Application Interface and Algorithm Classes

The primary interface between the science algorithm and the SDPS environment is through a set of objects called the *TES Application Interface (TAI)* and the *TES Algorithm*. The purpose of the TAI is to encapsulate and abstract all information from the external environment, and make it available to an algorithm in a clean, transportable manner. Ideally, this architecture should completely isolate the algorithm itself, and consequently the algorithm developers, from any system-specific details. Additionally, it provides a clean, portable and extensible interface between the algorithm and those items dependent upon the operating environment, such as files, logs, process control, etc.

The TAI executes two major tasks: it parses parameters and their definitions from multiple sources, and makes the complete set available to an algorithm; and it then creates an instance of a specific science data processing algorithm and executes it, providing all necessary information about the

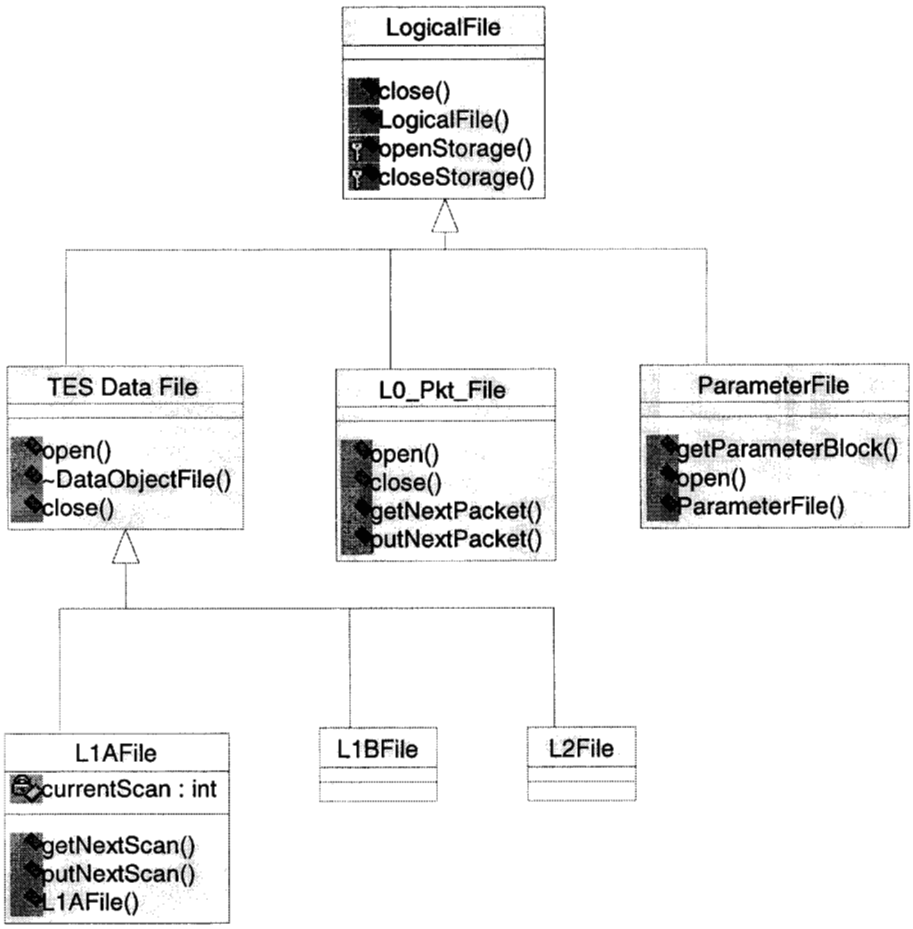

Figure 5. Logical File Hierarchy

system and the run-time parameters needed by the algorithm. A simplified sequence diagram is shown in Figure 4.

Instantiated within the TAI is a derived class of type *TES_Algorithm*. This derived class is highly specific and usually extremely computational. The Algorithm component is based on the Bridge design pattern. Each level of processing (as shown in Table 1) will derive its own algorithm object, and use this derived class as a template or wrapper for including or developing the scientific data processing code. The base class itself will provide access to those common properties and methods that are deemed essential to ensure consistent operation of algorithm objects throughout the data processing sequence. This base class supports log and error file handling, exception handling, process control, etc., via the aggregation of objects of types specific to those items. Access to system- or process-wide parameters is accomplished via the TAI's aggregated parameter instance (itself a singleton object).

File Handling

One of the most important areas that the Framework must deal with is that of file handling. The ultimate goal is to provide algorithm developers with a file mechanism that is completely transparent, incorporating complete independence of the underlying operating system. In fact, the requirements to support HDF-EOS via a toolkit for file access is driving a very complex file handling structure. Additionally, the nature of the data to be stored implies a large amount of additional complexity. As an example, it is likely that users and/or developers will wish to access time-dependent, multivariate complex data based on time, orbit number or geographic location or region. They may wish to read or write data items across any of these possible slices.

Currently, the design is based on a combination of several design patterns. The logical files, which provide the interface to all subsystems, have a straightforward dependency on a base class called *Logical File*. Logical files correspond to the needs of any given level-specific algorithm, such as Interferogram data files, raw data packet files, etc. The logical file inheritance tree with some sample file types is shown in Figure 5.

Physical files encapsulate environment and format-specific details of actual data storage, isolating these details from the application programmer. In some instances, such as HDF format files, data may span more than one actual disk file. It

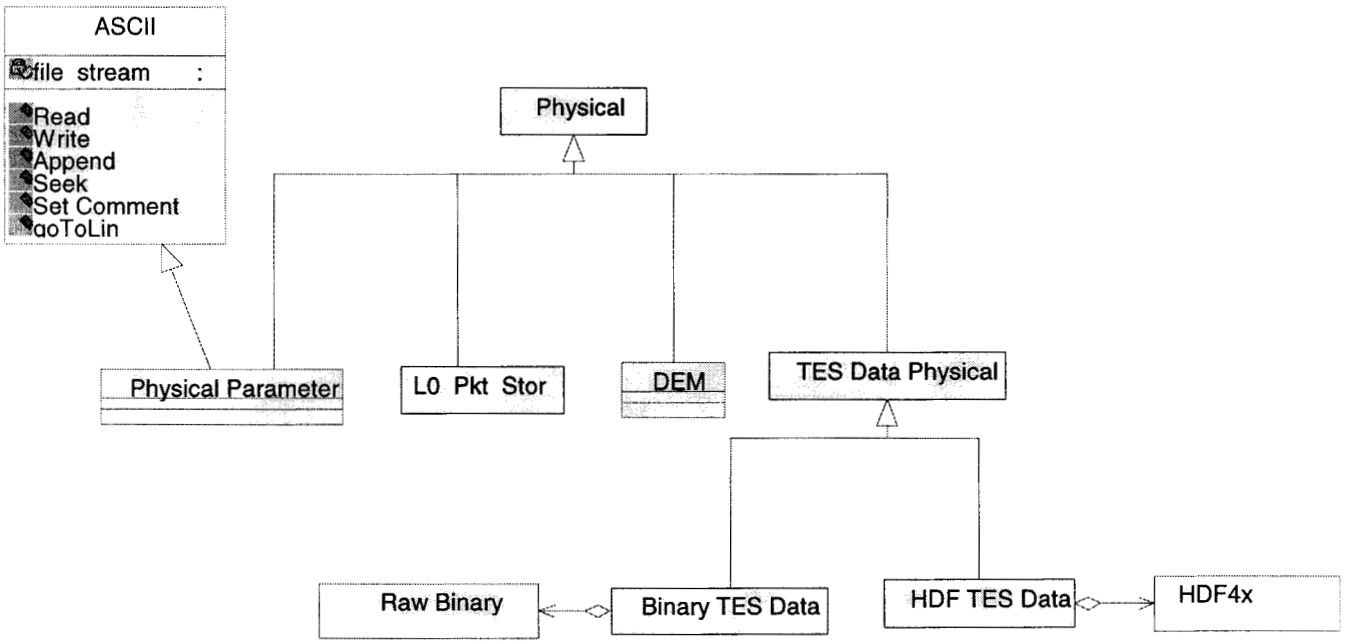

Figure 6. Physical File Hierarchy

is the responsibility of the Physical File hierarchy to handle such matters, perhaps using 3rd-party toolkits in some instances. The Physical File hierarchy is based on the Adapter pattern, and is shown in Figure 6.

The relationship between these two sub-components is maintained via a set of Bridge patterns, one bridge for each derived logical file type, as shown in Figure 7.

Logical Data Objects

Concurrent with the development of file handling and I/O mechanisms, we determined the need to develop a rich hierarchy of *Logical Data Objects*. Because these objects would need to interface with the file structure in order to read and write themselves, they naturally appear to require an internal structure that closely mirrors the internal (logical) structures of the HDF files that contain the data. Examples

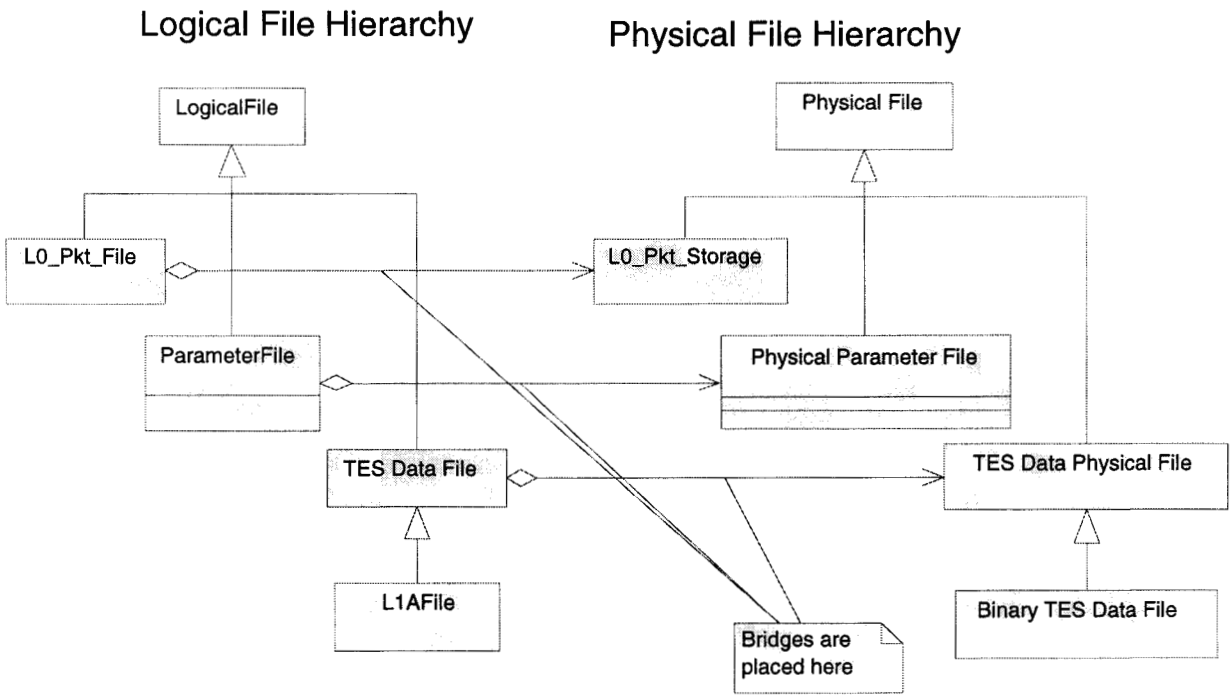

Figure 7. Use of Bridge Pattern to Link Logical and Physical Hierarchies

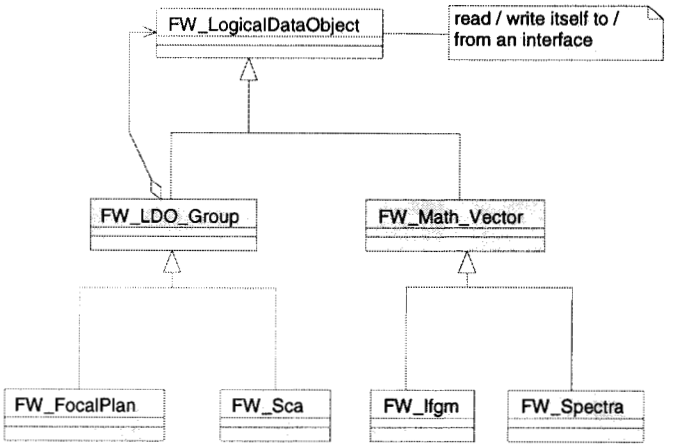

Figure 8. Possible Structure of Data

of high-level domain-related data types that must be supported are *scan*, *focal plane*, *interferogram*, and *spectrum*. These may be singular or combined in various ways, including multiple instances of any one type or combinations of types. An interferogram is a large array of either real or complex numbers, usually approximately 40,000 bytes in size. Included in an interferogram are various attributes, such as pixel number and temperature. A spectrum is similar to an interferogram, the two types being related via Fourier and Inverse Fourier transforms. A Focal Plane is a collection of 16 interferograms, plus additional attributes. Finally, a Scan is a set of 4 focal planes, plus spacecraft position and pointing information, etc. A draft class hierarchy for these objects is shown in figure 8.

It is highly likely that additional types will be required as science algorithm development and scientific data analysis proceeds during pre-launch and operational phases. Therefore, the architecture must provide an easily extensible model. Logical data objects are based on the Composite design pattern.

Naturally, there must be some sort of methodology for reading and writing data to and from the aforementioned file objects. In order to accomplish this, the Framework provides that every logical data object has an associated layout This layout specifies how a data object's read/write calls are to be interpreted by the file component, thus in effect providing a map from the logical structure to physical structure. The layout for a specific data object is provided by a Chain of Responsibility pattern-based object, which uses a Factory pattern-based object to actually construct the layout for a given file type and data object. These relationships are shown in Figure 9.

Parameter Handling

The Framework must be able to handle a virtually unlimited number of parameters that may be specified as inputs to any given algorithm (See Figure 10 for a simplified diagram of the parameter handling model). These parameters are defined in a separate parameter definition file for each level, as well as in coded default types for some parameters that may used in more than one algorithm. After parsing the definitions of all the specified parameters, the actual values may be declared in any number of places. The TAI instantiates a singleton object which contains a parameter block capable of parsing these values from the variety of

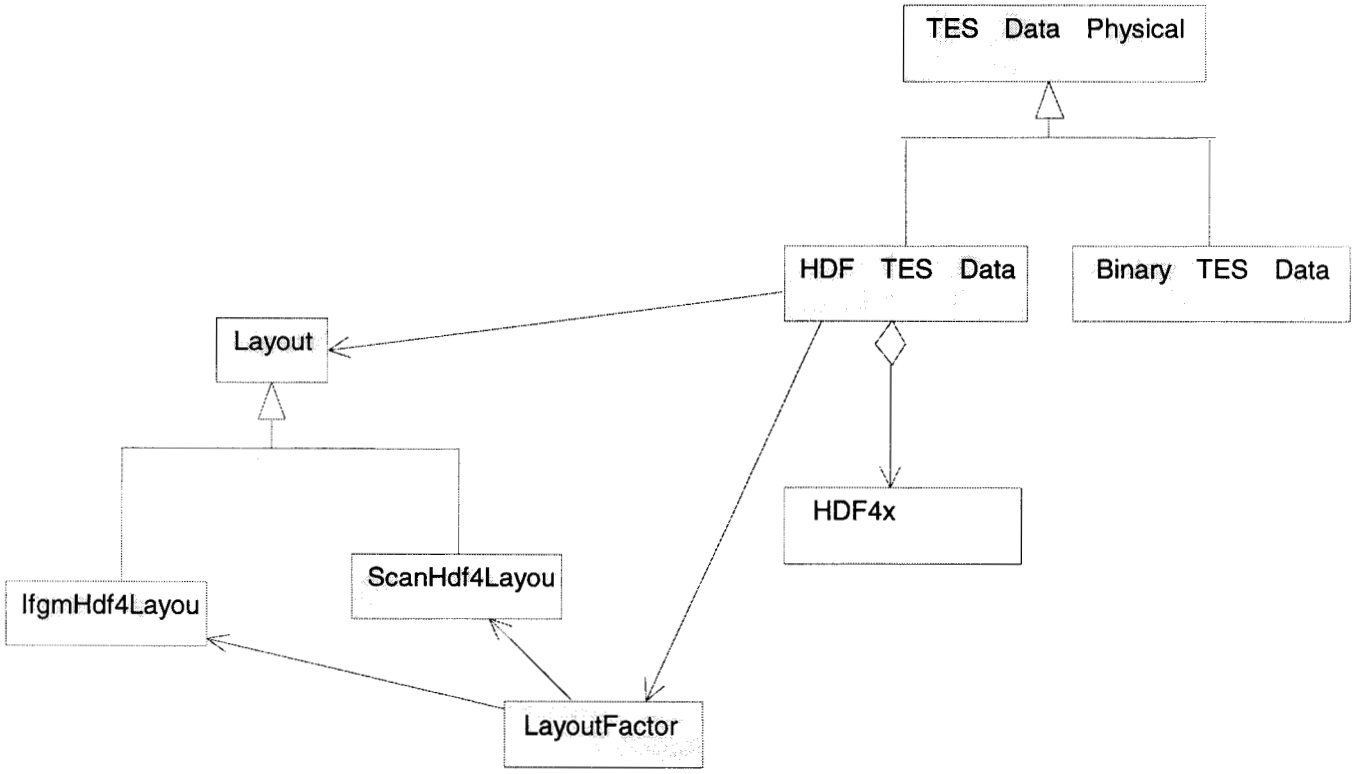

Figure 9. Relationship between Layout Classes and Physical File

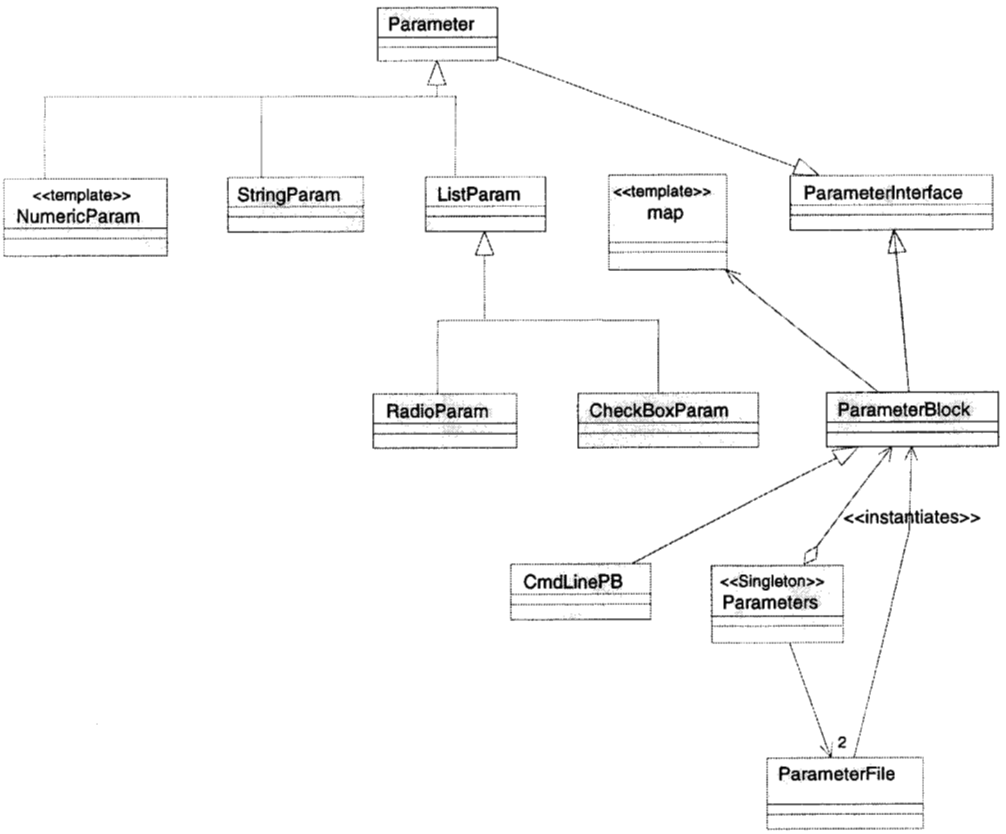

Figure 10. Parameter Handling Object

locales in which they may be set. These parameter values can be in a parameter file, as part of the operating system environment, specified on the command line or set as default values by the code itself. The parameters may be of virtually any type, including compound types. The parameter class which provides access to these data based on the Singleton pattern.

The requirement to support future, as-yet-undetermined data types seems to indicate that templates be used as a means of incorporating these types. One difficulty that arises as a result of this decision is the non-standard implementation of templates by various compilers. This issue is still under consideration by the development team.

Metadata Support

There are three types of metadata that must be supported by the Framework. Two of these types, inventory and archive metadata, are supported by the EOSDIS Core System (ECS) Product Generation System (PGS) Toolkit, supplied to all EOS projects by NASA. The Framework must provide a wrapper class for the Toolkit's metadata functions in order to simplify the task of application developers.

Inventory metadata include instrument identification, date and time of data acquisition, geographical location, product type and level, and data quality metrics. These data are stored in a relational database within the ECS, and are used to support user search and order functions. Archive metadata comprise a broader set of attributes that are stored within the data product files. These data are not stored in the ECS database, and thus are not available to search tools.

The third type of metadata supported by the Framework is instrument team-defined metadata. These data are not supported by the PGS Toolkit. Attributes defined by the instrument team are considerably more complex in structure than inventory and archive metadata. The latter are defined using the object description language (ODL), which is based on a "parameter = value" model. Instrument team metadata include such objects as error covariance matrices, residual vectors and references to supporting data or publications. These objects may themselves have associated metadata, introducing a recursive structure into the data organization.

Another important distinction between Toolkit-supported and instrument team-defined metadata is the level of configuration control applied. Toolkit-supported metadata are documented in ODL which is included in an interface control document between the TES project and the Langley DAAC. This is necessary since the ingest functions at the DAAC must be tailored to meet the TES metadata structure. Unilateral changes to the structure would break the automated ingest system, causing a significant disruption in the flow of data. Instrument team-defined metadata is not subject to these constraints, and thus may be changed more

freely. Framework components supporting inventory and archive metadata can be designed and implemented once, whereas components supporting instrument team-defined metadata are subject to change as science team requirements evolve.

Log and Error Files

During the normal course of scientific data analysis, an investigator may wish to evaluate the validity and/or appropriateness of the sequence of steps that led to a particular data item. For this reason, it is necessary to maintain a complete production history of all data processing steps in some form of log file or files. These files are archived along with the data, and some form of linkage between the data and the logs must be maintained. These logfiles are a special case of the file-handling model, with additional requirements imposed on them. They must be nearly global in scope to any algorithm, and must contain sufficient information about the process as to be able to recreate the circumstances of the process. This information includes but is not limited to: operating system information, processing times, parameter information, input and output data files, return and status codes, etc. For this reason, the Framework will include a special set of objects devoted to handling logfile and error file output information. Again, as with the logical data objects, there is slightly more coupling between these objects and the file manipulation objects than is normal for the bulk of the SDPS system, in order to hide the implementation of log file routines from the algorithm developers.

Status and Error Codes

Of the many items that may go into these log and error files, one standard type immediately emerges. Status codes are being implemented as an object hierarchy in their own right, in order to encapsulate error and return codes, severity levels, information output routines, and even the possibility of means of restarting failed processes with different parameters or control flow. Status code objects then also provide for a uniform means of dealing with exception handling, and a clean interface between the algorithms and the log or error files. Some possible difficulties, however, are the overhead involved in throwing and catching a somewhat more complex object, throwing such an object out of a routine, or returning a status code instead of a simple integer value. Since several data processing algorithms are exceptionally CPU-bound, any increase in processing time is to be stringently avoided. For this reason, we have no requirement on science algorithms that they use such an object within their own routines, but instead recommend a careful evaluation of the impact on their subsystem. Further, in order to simplify its use, we derive status code objects that encapsulate the normal process routine value of OK, and allow for simple tests for equality in such a case.

Process Control

The initial releases of the Framework are required only to support some form of multithreading support. Full multiprocessing (such as distributed processing, parallelization, etc.) is not initially required of the system. However, it is an express goal of the Framework design to be extensible enough to easily incorporate these requirements at a later date. For the first several iterations, however, all that is required is that the Framework provides an encapsulation of environment- and operating system-specific multithreading support. To that end, we anticipate a separate class hierarchy specifically designed to provide a simple interface to the underlying thread control methods. The terminal object(s) in this hierarchy may then be included in either the abstract base class *Algorithm* or a level-specific derived algorithm object thus providing thread control support to the derived algorithm objects.

Memory management

Due to the extremely complex, CPU-bound character of many of the processing elements, it is vital that memory be carefully managed. Some processing elements currently take tens of hours of CPU time on today's machines. In order to avoid the disastrous consequences of even the smallest memory leak on such highly computation-intensive subsystems, memory management is a crucial component. It is likely that the Framework will incorporate numerous memory management objects, including so-called "smart" pointers, local and remote memory pools, and so forth. The exact composition of the memory management objects is under review at this time.

However, as mentioned earlier, there is at least one concrete component that is likely to fall under the category of memory management: the 500GBs of absorption coefficients. This table will require a very sophisticated memory/file mapping scheme, possibly including multi-level random access, its own caching routines, or perhaps the equivalent of a data tiling or mipmap scheme, in order to provide extremely fast retrieval of the coefficients.

Foundation Objects

Finally, there are several major components of the data processing algorithms that utilize previously developed code, either specific to EOS data processing, or commercial off-the-shelf software. For example, there are toolkits for dealing with HDF-EOS files that we will incorporate into our file-handling objects using wrappers around those portions which the team specifies as required. Also, the high emphasis on mathematical precision in many of the processing steps implies that there will be one or more well-developed mathematical and statistical packages that will need to be incorporated. Many of these items fall into what we term the *Foundation*. This consists of a suite of wrapper classes for many of these libraries, where possible, or simply the interface definitions for the libraries where it is not. This allows us to hide implementation details and isolate the developers from changes in these libraries. Nothing in the

requirements prevents, where appropriate, direct access to these libraries by developers; however, such use is discouraged without good reason. Debugging and testing support is also planned to be included in this category (assertions, test scaffolding, 3rd-party test tools, etc.).

5. ISSUES

As mentioned earlier, there are risks, both technical and programmatic, which arose as a result of our decision to develop a framework. The broader deployment of a technique from the academic environment in which it was born is a complex process involving cultural, financial, as well as technical considerations. We discuss some of these issues in the specific context of the TES project below.

Framework Acceptance

Framework success will be determined at least in part on its level of acceptance among TES application developers. Not only must the Framework meet their requirements, its use must be seen as making their jobs easier. There are two main components to our strategy for Framework acceptance: team integration and documentation. As mentioned in section 3, we plan to involve members of the applications development teams directly in the Framework development. This should increase the level of "ownership" felt in the applications teams, as well as their understanding of what the Framework does and how it should be used.

The second component is good, clear and developer-friendly documentation. The documentation will provide an overview of the Framework design, a detailed description of the API, sample programs, and instructions for building a science application from the Framework classes.

Performance and Reliability

Since the Framework is to be used throughout the TES SDPS, there are naturally some rather stringent requirements on it in terms of performance and reliability. In general, the system requirements levy a set of requirements on the Framework which mandate error recovery and logging procedures. It is apparent from the intended use, however, that the Framework's requirements are considerably more strict, in that the Framework itself should rarely, if ever, terminate or abort, but should rather attempt recovery or throw an exception to the calling routine. That is, the Framework is not usually cognizant enough of its intended use to determine if it can fail gracefully. Components of Framework themselves must be highly reliable and robust. In nearly all cases, a Framework-based object or component should never fail catastrophically, but should raise an exception, attempt to recover, and (failing recovery) pass the exception to the appropriate subsystem for handling.

At this time, no strict performance requirements have been placed on the Framework. It is not anticipated that the Framework will be the bottleneck in most circumstances, given the high computational aspects of most processing, and the fact that the Framework generally deals with non-numerical components. The only items that may impact performance, according to initial evaluations, are the data layout objects and the logical data objects. Alternative designs are being evaluated for risk mitigation at this time.

Staffing

Our ability to attract and retain qualified staff is the most important non-technical risk factor associated with the Framework development. Staffing the Framework lead position proved to be particularly problematic, requiring over a year's time and one false start before a satisfactory candidate was put in place.

The TES Framework development must support a relatively large system over a long period of time. Thus, in addition to needing strong object-oriented analysis and design (OOA/D) and C++ skills, a successful lead engineer candidate also had to have strong system thinking skills as well.

The pool of potential candidates is not large relative to the demand. Further complicating our search is the presence of an increasingly IT-driven entertainment industry, and a growing number of high tech startups in Southern California. These competitors can offer earning potential that a federally funded research and development center like JPL cannot match.

Training

Domain experience is a critical element of success on science software projects. This kind of experience is not common, and individuals meeting our domain experience requirements typically have a background in C or Fortran programming. Training in OOA/D and C++ are therefore necessary to complete the skills base of the workforce.

We have found that finding or developing effective training programs is difficult and time-consuming. Additionally, the process of knowledge absorption and deployment into day-to-day work processes is a long-term rather than a short turn-around process. We have trained all staff in UML basics, but for those unfamiliar with modeling techniques, there are conceptual obstacles to achieving proficiency rapidly.

Similar problems obtain relative to our efforts to introduce OOA/D techniques. Making the transition from procedural to object-oriented thinking requires a paradigm shift that cannot be taught, but must instead be acquired over a period of time.

Virtually all team members have gone through an in-house three-day OOA/D training course specifically tailored to TES. This course was well received by participants, but was costly in terms of preparation time. It is not easily repeated for new team members who arrived after the course was taught. Many team members have attended a 5-day follow-

on OO course that was also found to be of value. The fact that some team members have found it necessary to repeat this and other courses underscores the fact that OO training is not a one-shot affair.

An additional obstacle to realizing the full benefit from training courses is the matter of timeliness. Budget and schedule instability, and a lack of control over when courses are taught, has made is virtually impossible to provide "just-in-time" training. Unfortunately, the longer the delay between training and utilization on the job the more likely the material is to be forgotten.

Resource Limitations

Ongoing pressure on the Federal government to reduce spending has affected the TES project since its inception in 1988, and is expected to continue for the foreseeable future. Particularly troublesome, given the long baseline for system development and operation, are the annual, and sometimes semi-annual changes to the current and future year budgets. These fluctuations have contributed substantially to a series of delays in staffing the Framework development., which have frustrated the efforts of the development team to sustain progress on the Framework task.

Framework Scope

Establishing the scope of Framework responsibilities has proven to be one of the most important technical challenges. Contributing factors are the lack of a pre-existing code base for the Framework, uncertainty in requirements, and lack of familiarity with the Framework concept on the part of users.

File management and data object support have proven to be particularly troublesome. Uncertainty in the requirements for multi-format support has at times driven the design team to solve problems that were later determined to be unnecessary. At other times requirements uncertainty has lead designers to make simplifying assumptions that were invalid.

Grappling with a large support system that is available only as a set of high-level abstractions has posed problems to users. There have been several cases where the Framework team and users proceeded down parallel paths with very different assumptions about what the Framework would provide. Budget-related staffing problems exacerbated this situation by making it difficult to support a sustained effort to clearly define the Framework responsibilities. The recent publication of a draft Framework API has helped resolve the majority of these types of issues.

Still unresolved is the issue of how much wrapping to provide third-party software. Viewpoints range from none to comprehensive. The underlying technical questions are related to the feasibility and desirability of complete isolation from these packages. How much is really needed? What is it worth? Will it be possible to provide a truly generic interface, say, to a linear algebra package that will allow the base package to be replaced at a later date without affecting application code? The data requirements and API structure of libraries are not always commensurate. Unless a comprehensive analysis of the APIs of all potential base libraries is carried out, there remains a chance that our encapsulation will fail.

6. SUMMARY

The TES project is currently at the end of the analysis and architectural design phase of the development of a science software applications framework. Detailed design and implementation of the first increment are now underway.

When the Framework decision was made over a year ago, many of the obstacles discussed in section 5 above were foreseen. While there have been some surprises, we have been fortunate that none have been significant enough to cause serious problems with the development process.

After a longer than expected search for the lead engineer, we have been able to produce a viable Framework architecture and high level design. Initial experience with the detailed design phase has so far been positive. Some concern about budget and staffing remains, and will remain until the first release of the Framework is successfully delivered in November of 2000. However, we currently have an adequate workforce and skills base to predict that this effort will be successful.

It will be a year before we will be able to report on whether the initial strategic decision to build a Framework from the ground up was the right one. However, presuming a successful implementation of release one, we feel confident that our architecture will fulfill our expectations for reuse and rapid application development in the future.

7. ACKNOWLEDGEMENTS

The authors would like to acknowledge the valuable comments provided in the preparation of this work by Ken Scott and Akbar Thobhani.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

8. REFERENCES

[1] http://eospso.gsfc.nasa.gov/eospso_homepage.html

[2] *HDF-EOS Library User's Guide, Volumes 1 & 2,* NASA Goddard Space Flight Center document # 170-TP-500-001, Greenbelt, MD: NASA GSFC, 1999. Also available at http://edhs1.gsfc.nasa.gov/waisdata/sdp/pdf/tp17050001.pdf and http://edhs1.gsfc.nasa.gov/waisdata/sdp/pdf/tp17050101.pdf

[3] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Reading, MA: Addison-Wesley,

1995.
[4] Gregory Rogers, *Framework-based Software Development in C++*, Englewood Cliffs, NJ: Prentice-Hall, 1997.
[5] *Proceedings of A NASA Focus on Software Reuse*, Washington, DC: National Aeronautics and Space Administration, 1996.
[6] See Draft Cooperative Agreement Notice at http://esdcd.gsfc.nasa.gov/ESS/CAN2000/CAN.html
[7] See the HPCC Home page at http://hpcc.arc.nasa.gov/
[8] Frederick Brooks, *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1995.
[9] Don Batory, *Intelligent Components and Software Generators*, paper available at http://www/cs/utexas.edu/users/schwartz/pub.html.

9. BIOGRAPHY

Steve Larson is a software project manager at the Jet Propulsion Laboratory. He currently manages the TES ground system development. He has been involved in the development of science data processing systems for several NASA Earth remote-sensing projects. Prior to joining JPL, he worked as a research assistant in the areas of plasma physics and low-energy nuclear physics. He has an Bachelor's degree in Art from Occidental College, and a Master of Science in Physics from California State University, Northridge.

Stephen H. Watson is a Senior Engineer at the Jet Propulsion Laboratory (JPL), and is currently the Cognizant Design Engineer for the Tropospheric Emission Spectrometer (TES) Science Data Processing System Framework. Prior to joining the TES team, he worked at Magellan Corporation developing Global Positioning System receiver and post-processing software, and was responsible for managing a multinational software team. He was previously employed at JPL as a software engineer specializing in scientific data visualization, supporting a variety of planetary and earth science missions. Mr. Watson received a Bachelor of Science in Mathematics and a Master of Science in Computer Science from Arizona State University.

Kalyani Rengarajan is a Principal Engineer at Raytheon ITSS, Pasadena. She is currently working on TES under a contract with JPL. Before joining Raytheon, she worked at JPL on advanced prototyping projects in distributive collaborative engineering, and Earth Science and deep space mission projects. She has a Masters in Computer Science and Masters in Mathematics from Indian Institute of Technology, Madras, India.