

*Mission Data System*  
**Framework Description**

November 2, 2002  
Version 1.0

Approved by:

---

John Lai , MDS Project Manager

---

Date

Jet Propulsion Laboratory  
California Institute of Technology

## *Mission Data System*

# Framework Description

This document provides an overall description of the MDS Framework technology. Since the purpose is to provide a general reference for the frameworks, the descriptions are organized as compendium. This document does not provide guidance for how the MDS technology should be used.

This document includes 3 Sections:

- Section 1 provides a summary of the framework and a brief description of each framework package.
- Section 2 provides a description for each individual package including package functionality and relevant glossary items.
- Section 1 provides an index of glossary terms and abbreviations.

*NOTE: The frameworks are intended to describe functionality that is independent of a specific programming language. Nonetheless, this document includes many references to C++ constructs because the current MDS implementation is in C++. Subsequent implementations will be in other programming languages, like Java. References to C++ constructs have been included here because they address C++ specific issues, or because they describe capabilities that have not, as yet, been generalized.*

# 1 Overview of Framework Packages and Layers

The MDS-based systems are built from a reusable set of core capabilities. The complete set of these core capabilities is called a framework.

The framework is organized into a set of packages. Each package contains a set of related functions created to satisfy a capability area. The packages are described in detail in Section 2, "Frameworks."

The packages are organized into layers. These layers are used to manage the dependencies between code elements. Dependencies only flow down from higher layers to lower layers. For example, level-1 packages may *not* depend on packages in levels 2, 3, 4 or 5. Level-3 packages may depend on packages in level 1, 2 or 3, but may *not* depend on packages in levels 4, or 5. And so on.

The levels are organized roughly into types of functionality. This is, at best, a labeling of convenience and should not be seen as a rigorous organizing principle.

Level 1: Primitive Services	Generic programming capabilities that are commonly required for the development of embedded systems.
Level 2: Simple Services	Low-level services needed for debugging, runtime configuration and temporal programming
Level 3: Complex Services	Generic component and database capabilities
Level 4: State Services	Support services of the MDS state-based paradigm
Level 5: Application Services	Capabilities needed for runtime construction, system operations, system test and generic visualization tools

The layer diagram in Figure 1 depicts seven layers in the MDS core framework and the packages in each layer. Each package is briefly described in Table 1, "Brief description of MDS Framework Packages ."

Figure 1: The MDS Framework is layered

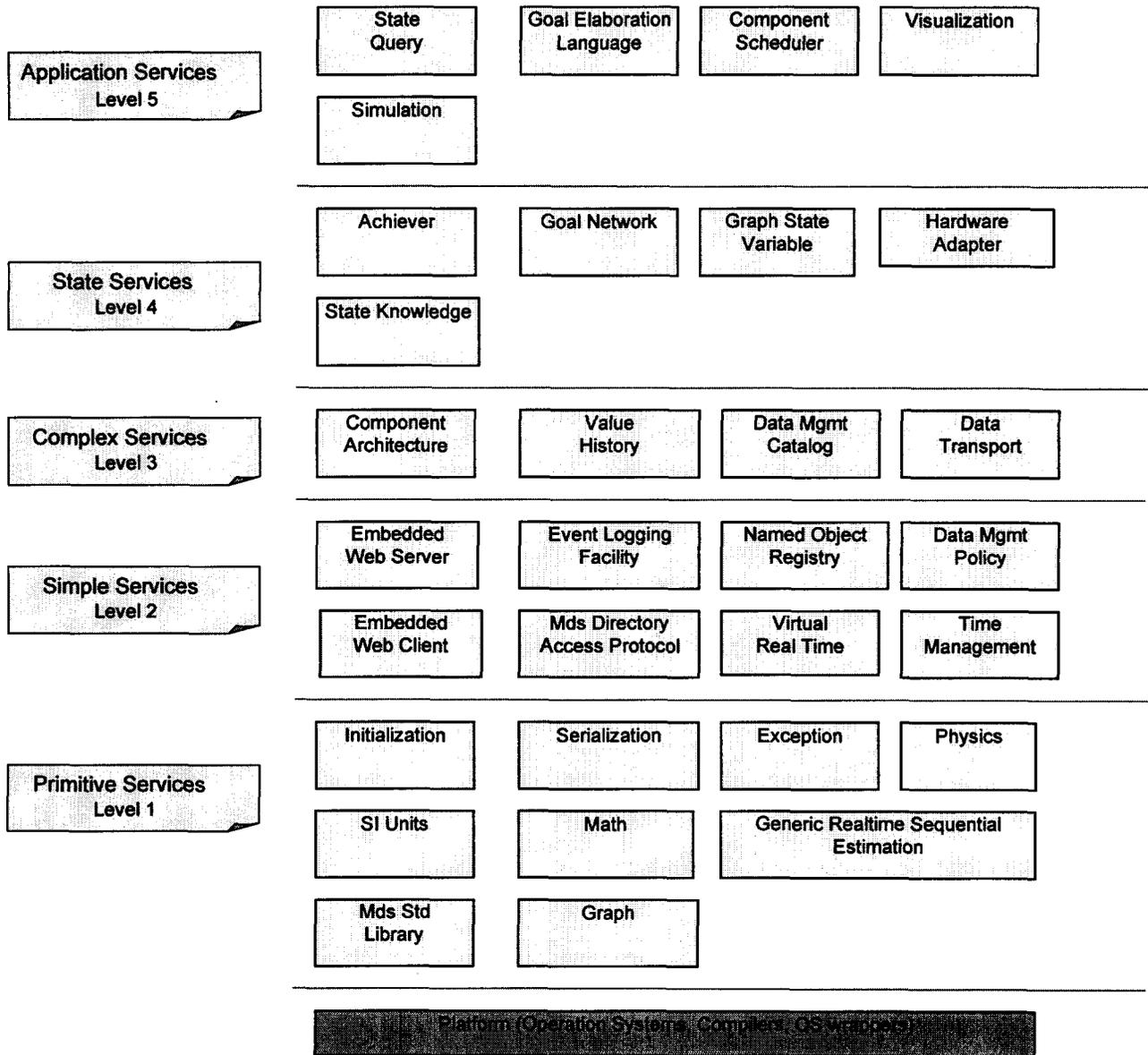


Table 1: Brief description of MDS Framework Packages

	<b>Package</b>	<b>Description</b>
1.	Achiever	Provides goal-driven executable components that strive to achieve executable goals.
2.	Component Architecture	Provides a runtime system for the management of explicit software architectures
3.	Data Management Catalog	Organizes data storage and provides query mechanisms.
4.	Data Management Policy	Provides a mechanism for creating objects that govern system activity by establishing rules for triggering actions.
5.	Data Transport	Provides abstract interfaces that must be supported by a MDS-compliant transport service
6.	Event Logging Facility	Provides a general logging mechanism for reporting noteworthy events during mission operation. Provides message filtering and transport interface.
7.	Embedded Web Client	Provides a thin web client following the HTTP 1.0 protocol. Enables interaction with an embedded web server (EWS) via the HTTP protocol, in the same way that browsers talk to web servers.
8.	Embedded Web Services	Provides a thin web server following the HTTP 1.0 protocol. Provides web access to the MDS component manager.
9.	Exception	Provides a standard mechanism for defining exception types and creating exception variables.
10.	Generic Realtime Sequential Estimation	Provides an application programming interface (API) for developing extended Kalman filters
11.	Goal Elaboration Language	Provides a language for specifying the elaboration of goals and other related high-level functions
12.	Goal Network	Provides the state control mechanism used to build and execute temporal constraint and goal networks.
13.	Graph	Provides the data structures and algorithms need to represent and work with discrete graphs.
14.	Graph State Variable	Provides a mechanism for defining relative states as pair-wise directed relationships between nodes in a graph
15.	Hardware Adapter	Provides base classes for three kinds of hardware adapter: basic actuator (command interface), basic sensor (measurement interface), and basic device (both interfaces).
16.	Initialization	Provides the mechanism for managing dependencies among singleton classes

	<b>Package</b>	<b>Description</b>
17.	Math	Provides common mathematical computations for MDS frameworks, adaptations, and deployments
18.	MDS Directory Access Protocol	Provides a lookup service for storing and retrieving deployment addresses
19.	MDS Standard Library	Provides utility functions that are used frequently throughout the MDS frameworks
20.	Named Object Registry	Provides capability for storing values with an associated name in a global registry
21.	Physics	Provides a set of common physics objects and computations
22.	Scheduler	Provides interfaces, components, and connectors specialized to support the design of multi-threaded software architectures
23.	Serialization	Provides functions needed to convert values to and from a series of bytes from transmission over a communications link
24.	SI Units	Provides data representations for scalars and associated mathematical operators. The purpose of the package is to detect errors in expressions involving physical dimensions and to eliminate errors involving units
25.	Simulation:	Provides tools that simplify the job of building simulation components.
26.	State Knowledge	Provides a mechanism for representing state variables and their values.
27.	State Query	Provides the functionality to query state and measurement histories
28.	Time Management	Provides mechanisms for the initialization and management of time services
29.	Value History	Provides data containers for components that generate state knowledge
30.	Virtual Real Time	Provides the virtual real time ticks to a network of deployments—the capability makes it possible to control the speed at which time passes in a testbed. Time can run slower or faster than wall clock time. Time can be started and stopped
31.	Visualization)	Provides the functions necessary to view the results of state queries

## 2 Frameworks

Each MDS Framework package provides a set of capabilities. This section includes an overview and a list of functions and descriptions of those functions for each framework package.

The following table provides a convenient list of the packages descriptions with page numbers.

<u>Package</u>	<u>Page</u>
<u>Framework Package 1: Achiever</u> .....	8
<u>Framework Package 2: Component Architecture</u> .....	9
<u>Framework Package 3: Data Management Catalog</u> .....	11
<u>Framework Package 4: Data Management Policy Package</u> .....	13
<u>Framework Package 5: Data Transport</u> .....	14
<u>Framework Package 6: Event Logging Facility</u> .....	15
<u>Framework Package 7: Embedded Web Client</u> .....	16
<u>Framework Package 8: Embedded Web Server</u> .....	17
<u>Framework Package 9: Exception</u> .....	18
<u>Framework Package 10: Generic Realtime Sequential Estimation</u> .....	19
<u>Framework Package 11: Goal Elaboration Language</u> .....	22
<u>Framework Package 12: Goal Network</u> .....	24
<u>Framework Package 13: Graph</u> .....	25
<u>Framework Package 14: Graph State Variable</u> .....	27
<u>Framework Package 15: Hardware Adapter</u> .....	29
<u>Framework Package 16: Initialization</u> .....	31
<u>Framework Package 17: Math</u> .....	33
<u>Framework Package 18: MDS Directory Access Protocol</u> .....	36
<u>Framework Package 19: MDS Standard Library</u> .....	37
<u>Framework Package 20: Named Object Registry (Nor)</u> .....	39
<u>Framework Package 21: Physics</u> .....	40
<u>Framework Package 22: Scheduler</u> .....	42
<u>Framework Package 23: Serialization</u> .....	44
<u>Framework Package 24: SI Units</u> .....	45
<u>Framework Package 25: Simulation</u> .....	47
<u>Framework Package 26: State Knowledge</u> .....	49
<u>Framework Package 27: State Query</u> .....	51
<u>Framework Package 28: Time Management</u> .....	52
<u>Framework Package 29: Value History</u> .....	53
<u>Framework Package 30: Virtual Real Time</u> .....	55
<u>Framework Package 31: Visualization</u> .....	57

## Framework Package 1: Achiever

### *Overview:*

The Achiever (or State Achiever) package provides goal-driven executable components that strive to achieve executable goals. State controllers and state estimators are types of goal achievers; the former strive to achieve a constraint on the value of a state variable and the latter strive to achieve a goal on the quality of knowledge in a state variable.

All achievers support the following interfaces: 'runtime execution', 'constraint execution', and 'constraint scheduling'. In addition, controllers support the 'command submit' interface, and estimators support the 'state update' interface.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Runtime execution	Interface for the component scheduler to dispatch an executable component.
Constraint execution	Provides two operations: one to check if a constraint is ready to start given current state, and one to start execution of the constraint.
Constraint scheduling	Provides operations used during scheduling of executable goals to: determine if a goal is achievable, determine if a goal-to-goal transition is achievable, determine the earliest achievable merge of one goal with another, and project the effect of a goal forward in time.
Command submit	Provides an interface whereby a controller submits a command to a hardware adapter.
State Update	Provides an interface whereby an estimator updates the value of a state variable.

## Framework Package 2: Component Architecture

### *Overview:*

The Component Architecture Framework, or Car, package provides a runtime system for the management of explicit software architectures according to the xADL2.0 Architecture Description Language (see: <http://www.isr.uci.edu/projects/xarchuci/>). Car provides facilities for defining software architecture elements (i.e., types of components, connectors, interfaces) and for managing the creation, destruction, modification and linkage of their instances. The practice of software engineering with explicit attention to software architecture design, representation and methodology is a necessary aspect of large-scale, extensible, reusable, compositional and reliable software systems.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Architecture definition	Functions that provide assistance in defining components, connectors, interfaces and related concepts. These type definitions are represented as objects that the Car runtime uses to verify and validate architecture prescription requests. All architecture type definitions are grouped in architecture libraries, units of binary deployment that can be composed with other architecture libraries to form a functioning software system.
Architecture prescription	Functions that provide assistance in defining a configuration of component and connector instances as well as linking them via their provided & required interfaces. Car provides multiple mechanisms for components, connectors and other software objects to interact with one another. These mechanisms include: <ul style="list-style-type: none"><li>• Several variations on the xADL notion of explicit links among interfaces</li><li>• An interception mechanism that allows the dynamic composition of aspects on interfaces</li><li>• An invocation mechanism that reifies synchronous interface function calls into event objects that can be invoked at a later time.</li></ul>
Architecture evolution	Functions that provide assistance in managing the lifetime of complex software systems via incremental composition and change of one or more architecture instances.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Component	A component is a unit of software deployment and reuse. It is defined in terms of provided and required interfaces. Each component has a corresponding model description in xADL.
Connector	A connector is a mechanism for mediating interaction among components via their interfaces. Each connector has a corresponding model description in xADL.
Deployment	Any instance of a running system. An MDS deployment is a running system built from the MDS frameworks.
Interface	An interface describes a set of software functions that a component can either provide to other components or require from other components.
Software architecture	Software architecture introduces a level of abstraction of software with notions of components, connectors and interfaces. This abstraction facilitates the description, modeling, design, reuse, testing and validation of complex software.

## Framework Package 3: Data Management Catalog

### *Overview:*

Data Management Catalog (or Catalog) package provides the persistent storage service, the Value History service, and the database management services for State Variables and other data mediating components. The Catalog also serves as the interface to the Data Transport subsystem.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Create collection	Create a new collection with the given name in the given collection.
Create container	Create a new container collection with the given name in the given collection if one doesn't already exist.
Find collection	Find only one matching collection by name.
Get sender	Accessor for the current product sender object.
Report contents	Reporting method for listing the contents of the named collection.
Require collection	Same as create collection except that it will succeed even if the specified collection already exists.
Retrieve product	This function is a request to find a previously submitted product. The parameters to the function can include quite complex conditions on the metadata of the products to be retrieved. For example, a retrieve call might specify that only products created from 11:45 to 12:15 am, with size not more than 10kb, and appearing in certain collections are to be retrieved.
Send sendable	Issue a special query to find all the products that are marked for transport but haven't been sent yet and put them in the send queue.
Set sender	Configure the catalog to use the given product sender to support transport. The catalog accepts products for transport, but it itself does not interface to communications equipment, so it requires an object called a sender to which it can give products destined for transport.

<u>Function Name</u>	<u>Description</u>
Submit product	Adds a product to a catalog collection. In executing this function, the catalog makes a copy of the product, stores it to persistent storage, and stores meta data about the product in its internal data structures for later retrieval of the product. If a product has certain special meta data attributes, then this product is understood to be queued for Transport.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Data product (or product)	The unit of data visible in the Catalog or in its interfaces. It's convenient to think of a product as a file with meta data attributes, and in workstation configurations of MDS, a product is stored as a file.
Accessor	Function for retrieving an encapsulated variable.
Product sender	An object that provides an interface to communications software and which can send products out on the communications link view this interface.

## Framework Package 4: Data Management Policy Package

### *Overview:*

The Data Management Policy package, or Policy, provides a mechanism for creating objects that govern system activity by establishing rules for triggering actions. For example, policies are can be used by value histories to implement rules like: 'Every time 5 new state values have been created, put the values in a product and submit the product for transport.'

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Policy actuator	A user-defined type that implements rule. Each policy actuator class has a condition and action. When a policy actuator object is evaluated, the object tests the condition—if it is true, the action is executed.
Condition	The part of a policy actuator object that is used to determine if the policy's action should be executed. Users of this package can define their own types of conditions, and still use the policy actuator class to implement the rule.
Action	The part of a policy actuator object that is executed when the conditions of the rule hold. Users of this package define their own action types. Thus, the policy package can be used to implement a large variety of rules.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Policy actuator	User-defined type that implements a rule for triggering system activity.

## Framework Package 5: Data Transport

### *Overview:*

The Data Transport package provides the abstract interfaces required of a MDS-compliant transport service. These interfaces define the data transport service's ability to enqueue and dispatch data products to and from a remote system.

In addition, the package includes a simple reference implementation that may be used as a test harness. This reference implementation uses a simple TCP socket connection between two instances of the transport session in different deployments.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Send product	Request that the given data product be sent across an associated link using the transport service.
Receive product	This is a callback method. When a product has been received and completely reconstructed, this method will be called to handle the product and decide what to do with it.

## Framework Package 6: Event Logging Facility

### *Overview:*

The Event Logging Facility package, or ELF, provides a mechanism for recording messages generated when the system is operating. The ELF record, or log, may be used later by human operators to help diagnose problems or evaluate the performance.

ELF includes three sub packages:

- An interface package for reporting events
- A package that specializes the reporting package for initialization, finalization, or other special cases.
- A package for storing and transporting recorded messages. In particular, it provides control for storing and filtering messages. It also provides a mechanism for specifying which messages are generated.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Generate message	This is the generic function that can be called to generate an event message. The argument contains the message. The method will add a time tag to the event message and will decide whether or not to save it based on internal filtering options.
Set filters	Set filtering options for a particular kind of event message. Messages can be filtered to suppress on a duration basis (permit messages to be generated no more often than the given duration), interval basis (suppress all but one of every N instances), or a severity basis (suppress messages whose severity is less than X). A given message type can also be disabled entirely.
Initialize	Initialization options primarily include specification of a class (method) for handling messages. Elf can be configured to send messages to a file, to a console, or to a handler event that will convert the event messages into data products for transport to another MDS deployment.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
ELF	Error Logging facility

## **Framework Package 7: Embedded Web Client**

### ***Overview:***

The Embedded Web Client Package (or EWC) provides a web client following the HTTP 1.0 protocol. EWC enables interaction with an embedded web server (Ews) in the same way that browsers talk to web servers. This service sends HTTP requests via a TCP/IP port to a web server. The service then parses the HTML response for the user of the client.

### ***Description of basic functions:***

<u>Function Name</u>	<u>Description</u>
Http send	Sends an HTTP request to a particular address.
Http send to server	Sends an HTTP request to a server name looked up via Mds Directory Acces Protocol (MDAP).

### ***Glossary:***

<u>Term</u>	<u>Definition</u>
EWC	Embedded Web Client.
HTTP	Hyper Text Transfer Protocol.
HTML	Hyper Text Markup Language.
MDAP	MDS Directory Access Protocol (See MDAP document).

## Framework Package 8: Embedded Web Server

### *Overview:*

The Embedded Web Server package, or EWS, provides a web server following the HTTP 1.0 protocol. This service receives HTTP requests via a TCP/IP port and dispatches them to other software components via a registration table. These software components register themselves at runtime with a text string corresponding to the resource string sent in the HTTP request. The software components then format an HTML response to send back to the requestor.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Register CGI Function	Function called to register a software component for dispatching HTTP requests. The registration is with a string corresponding to the resource string passed in the server HTTP request.
Do dispatch	Dispatches a given HTTP request.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
EWS	Embedded Web Server.
CGI	Common Gateway Interface. A standard used by web servers to communicate with other hosted processes.
HTTP	Hyper Text Transfer Protocol.
HTML	Hyper Text Markup Language.

## Framework Package 9: Exception

### *Overview:*

The Exception package provides a standard mechanism for defining exception types and creating exception variables. The exception package simply defines a base class for all MDS exception (error) messages. Exception can be used directly, or extended in other MDS classes to define all exceptions that can be generated at runtime from MDS code.

Having a common base class allows the exceptions generated at runtime to be more easily identified as having come from MDS code (as opposed to some third-party software). The MDS exception base class is derived from the standard library exception class and provides no additional functionality.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Basic exception	Base class for all MDS exceptions. Having a single Base Class of all thrown objects in MDS simplifies exception handling.
Message exception	A class that defines exception objects containing messages that provide detailed information about the exception. Message exceptions make creation of exception variables convenient and facilitate ability to add pieces of information from other variables of various types.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Class	In C++ or Java, a user-defined type that incorporates functions and data variables. A class may be extended to define refinements of the original class, in which case the extensions are said to be 'derived' from the original 'base' class.
Base class	A class from which other classes are derived. Base class are usually intended to establish the basic functionality or role of an entire hierarchy of classes.

## Framework Package 10: Generic Realtime Sequential Estimation

### *Overview:*

The Generic Realtime Sequential Estimation package, or GREASE, provides an application programming interface (API) that facilitates developing extended Kalman filters (an algorithm reported extensively in the open literature since the early 1960's) for real-time and post-processing applications. The GREASE API provides a generic mechanism for estimating parameters, based on streams of measurement data, measurement sets measurement and dynamics models.

In practice, GREASE is co-compiled with other, non-GREASE, user-supplied software modules that mathematically compute the measurements and system dynamics of particular interest to the user. The resulting product of the co-compilation is a new extended Kalman filter routine, program, or application.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Add matrices	See glossary for "matrix". This function adds two matrices together and returns the sum.
Multiply matrices	See glossary for "matrix". This function multiplies two matrices together, and returns the product.
Convert quaternion to Euler angles	See glossary for "quaternion" and "Euler angles". Converts between these two attitude representations.
Convert Euler angles to quaternion	See glossary for "quaternion" and "Euler angles". Converts between these two attitude representations.
Propagate quaternion in time	See glossary for "quaternion". Given the attitude quaternion of a body at a start time, computes the attitude quaternion of the same body at a different time, by multiplying the time rate of change of the body's attitude quaternion, by the change in time, via standard integration steps.
Propagate covariance matrix	See glossary for "covariance matrix". Given the covariance matrix at a start time, computes the covariance matrix at a different time, using appropriate user-provided matrices containing the user's system dynamics models, and uncertainty/assumed error in the dynamics models.
Propagate state vector	See glossary for "state vector". Given the state vector at a start time, computes the state vector at a different time, using appropriate user-provided matrices containing the user's system dynamics, in the form of the first derivative in time of the state vector.

<u>Function Name</u>	<u>Description</u>
Kalman-update covariance matrix	See glossary for “covariance matrix”. Given the covariance matrix at some time that a measurement is made, representing the uncertainty in the state vector at that time but without the information in the measurement, this function computes the covariance matrix (by standard Kalman update referenced in the literature) representing the improved uncertainty after applying the information in the measurement.
Kalman-update state vector	See glossary for “state vector”. Given the best estimated state vector at some time that a measurement is made, this function applies the standard Kalman update (referenced in the literature) to make a linear improvement “step” to the estimated state vector, based on the information in the measurement. Value of the improvement “step” depends on the user-supplied formulation for the mathematical model of the measurement.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Covariance matrix	Associated uncertainty of each parameter in the state vector, and correlated uncertainty, in the form of an $n$ -by- $n$ matrix, where $n$ is the dimensionality of the state vector. An always-present by-product of any extended Kalman filter.
Dynamics model	The mathematical expression of the time evolution of a system. For example, the mathematical expression for the sum of the shear and normal forces acting on the wheels of a vehicle, along with the weight of the vehicle, and the wind force pushing on a vehicle, which gives the rate of change of the velocity of a vehicle. A user-defined input to any extended Kalman filter; therefore, not supplied by GREASE.
Euler angles	Another standard representation of the attitude (“pointing direction”) of a body in space, with property of having well-known, intuitive physical meaning, in terms of roll, pitch, yaw about a set of axes. Euler angles have problematic mathematical singularities (regions where numerical values go to infinity, even though expected physical value has not gone to infinity) when propagated in time.
GREASE	Generic Realtime Sequential Estimation.

<u>Term</u>	<u>Definition</u>
Matrix	Standard representation of an ordered set of numerical values, in an arrangement of n rows by m columns (n and m integer, greater than zero). Standard rules of linear algebra apply to adding, subtracting, multiplying and dividing matrices.
Measurement	Numerical value of the output of a sensor, said output being mathematically represented as a function of the state vector. An user-defined input to any extended Kalman filter; therefore, not supplied by GREASE.
Quaternion	Standard representation of the attitude (“pointing direction”) of a body in space, with mathematical property of not having mathematical singularities when propagated in time.
State vector	Set of n parameters being estimated with the extended Kalman filter – defined by user. Main product of any extended Kalman filter.

## Framework Package 11: Goal Elaboration Language

### *Overview:*

The Goal Elaboration Language package, or GEL, provides a language for specifying the elaboration of goals and other related high-level functions. We also refer to the processor for this language as “GEL”.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Define goal	Define a goal in GEL. A goal definition specifies such things as the underlying state variable, state constraint, and the time points of the goal, as well as any tactics for maintaining the goal. These tactics may include other sub-goals and temporal constraints. For example, other values may be passed as parameters to be used inside of tactics.
Make goal	Create a new goal from a GEL goal definition. Any number of goals can be created from a single goal definition.
Elaborate goal	Once a goal is created, it may be elaborated as a separate step. Elaboration entails trying the tactics for maintaining the goal, until one is found. Elaboration relies on package GNET to do much of the work, Gel acting as an interface.
Define subnet	A subnet is a network of goals and temporal constraints, intended for insertion into the overall goal network. It is similar to a tactic, the difference being that it does not reside within some goal definition.
Insert subnet	Insert a defined subnet into the network.
Make Xgoal	Make an executable goal. Executable goals (“Xgoals”) are imported from lower level code. When an Xgoal is made, it is immediately inserted into the network, in contrast to a goal, which does not affect the network until it is elaborated.
Make time point	Create a new time point, for example one to be used in conjunction with one or more Xgoals.
Make temporal constraint	Create a new temporal constraint, for example one to be used in conjunction with Xgoals to constrain their execution start or finish.

<u>Function Name</u>	<u>Description</u>
Define value	In addition to the items mentioned above, many other kinds of values can be defined in GEL, which provides the computational capability of a general purpose programming language.
Compute value	GEL provides an expression language that can be used to compute arbitrary arithmetic and symbolic values. The syntax for this aspect is similar to conventional languages such as Lisp or Scheme.
Define function	GEL provides for defining arbitrary computational functions, either in GEL itself or by binding to a function defined in the implementation language, currently C++. These functions are bound to names so as to be useable within GEL.
Load file	GEL can specify that files containing additional GEL expressions are to be loaded and interpreted.
Trace	Turn on or off a trace of GEL execution.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Binding	The association between a name and a value or function.
GEL	Goal Elaboration Language.
Goal	A constraint on a state variable that is to be maintained between two time points.
Time point	A symbolic point in time. The actual time represented by a time point is determined during execution, and is constrained by temporal constraints with other time points.
Temporal constraint	A constraint between two time points that specifies a minimum and maximum delay time between those time points.
Xgoal	Executable goal.

## Framework Package 12: Goal Network

### *Overview:*

The Goal Network package, or GNET, provides the framework for state control. GNET is used to build and execute temporal constraint and goal networks. GNET also provides planning and scheduling capabilities for these networks.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Propagate network	Calculate the temporal consistency of the temporal constraint network. Calculate the temporal relationship between time points in a network.
Schedule network	Temporally constrain conflicting goals so they do not overlap in time.
Elaborate goal	Recursively create supporting goals as specified by a dependent goal.
Execute goal	Deliver a goal specification to the appropriate controller.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Goal	A desired state over a particular temporal interval.
GNET	Goal Network package.
Temporal constraint	The specification of a flexible temporal relationship between time points in a network.
Temporal constraint network	A directed graph whose edges are temporal constraints and nodes are time points.
Goal network	A directed graph whose edges are goals and nodes are time points.

## Framework Package 13: Graph

### *Overview:*

The Graph package provides data structures and algorithms needed to represent and work with discrete graphs. A discrete graph is a mathematical structure defined in terms of a finite set of vertices and a finite set of edges. A graph edge defines a binary relationship between two vertices. Such graphs are pervasive throughout computer science—working with graphs is a fundamental aspect of this field.<sup>1</sup>

The MDS Graph framework is a straightforward implementation of the concepts and techniques described in the popular computer science textbook: “Introduction to Algorithms” by Cormen, Leiserson and Rivest. There are two differences from the treatment of graphs in this text: 1) caching and 2) membership graphs. The textbook focuses on algorithms but does not address the computational issues involved in repeated uses of such algorithms. A simple optimization consists in storing the results of the first execution of an algorithm and reusing these results for subsequent requests for the same algorithm. A membership graph is a specialized application of a discrete graph that tracks the membership of each vertex to different groups of vertices.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Graph editing	Construct an empty graph. There are two kinds of graphs: topological graphs & data graphs. Topological graphs represent only the topology of vertex/edge connectivity. Data graphs include a topological graph and an association of a user-defined data for each vertex and each edge. Basic editing functions include adding & deleting vertices & edges.  To optimize query performance, graphs maintain a query cache. Cache editing consists of closing the graph for editing (i.e., it can be queried but not edited) and of opening the graph for editing (i.e., it can be edited but not queried).
Graph query	Access properties of the graph (number of edges, vertices). Browse the graph topology (vertices, edges, paths through the graph).
Graph algorithms	Topological sort, Depth-first search (nearly a verbatim adaptation from Cormen et al).

---

<sup>1</sup> Graph theory is part of the standard computer science curriculum in either advanced undergraduate coursework or first year graduate studies in computer science.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Graph	A data structure that has a collection of vertices and edges. In a topological graph, vertices and edges are identified with a unique number. A data graph is like a topological graph but it associates a user-defined data item to each vertex and to each edge.
Vertex	An item that is part of the graph.
Edge	An item that represents a relationship between two vertices in a graph.

## Framework Package 14: Graph State Variable

### *Overview:*

The Graph State Variable package, or GSV, provides a mechanism for defining relative states as pair-wise directed relationships between nodes in a graph. GSVs are a general graph based state representation that (1) can derive a state's value by combining relationships, (2) produces different results for different derivation paths, and (3) handles changes to topology and relationships between nodes.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Close	Completes the modification of a graph state variable's topology.
GetRelationship	Retrieves a value for the relationship between two nodes in a graph state variable.
Open	Prepares a graph state variable's topology for modification.
RecoverRelationship	Restores a value from persistent store for the direct relationship between two nodes in a graph state variable.
UpdateChildren	Adds constituent graph state variables to a composite graph state variable as specified in a read/write parameter file.
UpdateRelationship	Stores a value for the direct relationship between two nodes in a basis graph state variable.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Frame	A frame is point of reference.
GSV	Graph State Variable.
Node	A node in a graph state variable is an abstraction of a frame.
Direct relationship	A direct relationship within a graph state variable is abstracted as an edge between two nodes, where the edge references the value of the relationship between the two frames represented by the two nodes.
Derived relationship	A derived relationship within a graph state variable is a relationship computed by concatenating the relationships along a path that has 3 or more nodes.
Basis graph state variable	A basis graph state variable is comprised of direct relationships that are estimated locally.

<u>Term</u>	<u>Definition</u>
Proxy state variable	A proxy state variable is comprised of direct relationships that are copied from a remote location.
Composite graph state variable	A composite graph state variable is comprised of graph state variables that are connected by sharing one or more nodes.
Constituent graph state variable	A constituent graph state variable is a graph state variable that is an element of a composite graph state variable.

## Framework Package 15: Hardware Adapter

### *Overview:*

The Hardware Adapter package provides base classes for three kinds of hardware adapter: basic actuator (command interface), basic sensor (measurement interface), and basic device (both interfaces). These classes can each be further specialized to suit given hardware devices.

A “hardware adapter” is an executable software component that provides uniform interfaces to a hardware device or its simulation. Interfaces are defined for submission of commands to actuators, querying of measurements from sensors, and policy-driven management of command and measurement histories. A hardware adapter may extend the functionality of a raw hardware device to provide a more convenient basis for monitoring and control, but it must not hide information needed for fault protection.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Submit command	This method is invoked by a controller component to submit a command. The method should do little more than store the argument command, so that the command can subsequently be enacted by the run method. The command may supersede, amend, or complement previously received commands.
Command hardware	In the commandable base classes listed above, this method is invoked by the submit command method. This can be used to execute all of the Hardware Adapter’s functions at the expense of the controller invoking the submit command, so that the Hardware Adapter doesn’t have to be scheduled for a separate time-slice, but that’s not really recommended.
Get measurement	This method is invoked by an estimator component. Querying measurements out of a Hardware Adapter’s value history is enabled by framework code. No adaptation is needed, beyond writing measurements into the value history by invoking.
Run	The component scheduler periodically invokes the method for cyclically executed hardware adapters. The method should enact previously received commands (if any) and/or construct measurements for querying by an estimator (if any) or ground system.

***Glossary:***

Term

Hardware adapter

Definition

Executable software component that provides uniform interfaces to a hardware device or its simulation.

## Framework Package 16: Initialization

### *Overview:*

The Initialization, or INIT, package provides the mechanism for managing dependencies among singleton classes. A singleton class is a special class type that can have one and only one class instance at runtime.

The singleton is one of the many design patterns documented in “Design Patterns” by Erich, Gamma, Johnson and Vlissides. The Init framework is a substantial improvement over the basic singleton pattern for rigorously handling initialization, finalization issues of singletons in a way that provides both metrics on pattern usage, measures of testing complexity and a capability for exhaustive testing. The Init framework is highly portable across C++ compilers and platforms because it delays the initialization of all application-level singletons until the C++ runtime system has been fully initialized (including exception handling facilities) and forces the finalization of all application-level singletons to occur before the C++ runtime system loses its exception handling facilities.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Singleton pattern	There are several variations of the basic pattern available for regular C++ classes as well as for parametric classes also known as C++ template classes.
Instance	The framework includes an instance method for each user-defined application of one of the singleton patterns. This instance method performs runtime checks relative to the proper initialization and ordering of the singleton.
Topological sort	The Init framework sorts all singletons according to the topological sorting of their partial dependencies. This is a straight application of the Depth-First Search algorithm from “Introduction to Algorithms” by Cormen, Leiserson, Rivert. This algorithm is further instrumented to allow the framework to generate all possible topological sortings of the dependencies. This generative capability is essential for rigorous verification and validation of software.
Pseudo-static initializer	A pseudo-static initializer is an object that uses the “Resource Allocation Is Initialization” principle to ensure that all singletons will be initialized when the object is constructed and that all singletons that have been initialized will be finalized before the object is destroyed.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Singleton	A C++ class type that has a static method (usually called “instance”) that retrieves the unique instance of this class type. See “Design Patterns” by Erich, Gamma, Johnson and Vlissides for a complete description.
INIT	Initialization package
Topological sort	An algorithm from graph theory intended to find the total order of a set of items that have only partial ordering constraints among them.
Initialization & finalization	All software that runs on hardware is subject to an initialization phase before the software actually runs and a finalization phase after the software has completed its execution. The C++ standard provides weak guarantees about this phase of a program execution. These guarantees are insufficient to meet the needs of robust software. INIT includes a solution that provides strong guarantees about this process that either address critical software requirements or greatly facilitate their resolution.

## Framework Package 17: Math

### *Overview:*

The Math package provides common mathematical computations for MDS frameworks, adaptations, and deployments. It includes object definitions and methods for standard library limits class, time intervals, 3-vectors, quaternions, Euler angles, normal distributions, polynomial functions, Taylor series, linear algebra algorithms, six degree-of-freedom transformations, and math exceptions. The MONTE project provided many of the functions required to implement 3-vectors, quaternions, Euler angles, and linear algebra algorithms.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Contains	Returns true if the interval contains a member value.
Is superset of	Returns true if this interval contains another interval.
Is contained by	Returns true if this interval is a subinterval of another interval.
Is equivalent to	Returns true if this interval is equivalent to another interval.
Has null intersection	Returns true if this interval has no overlap with another interval.
Unit	Computes a unit vector (and time derivatives).
Transpose	Transposes a matrix.
Subtract	Computes the component-by-component difference of a matrix or a vector.
Project	Computes the projection of a vector onto another vector.
Orthogonal	Computes the component of a vector that is orthogonal to another vector.
Multiply	Multiply all elements of a vector (or a matrix) by a scalar.
Max magnitude element	Computes the maximum magnitude (absolute value) element of a vector.
Matrix vector multiply	Multiply a matrix times a vector.
Matrix transposed vector multiply	Multiply a matrix transposed times a vector.
Matrix matrix multiply	Multiply a matrix times a matrix.
Matrix transposed matrix multiply	Multiply a matrix transposed times a matrix.
Invert 3	Computes the inverse of a 3x3 matrix.

<u>Function Name</u>	<u>Description</u>
Magnitude	Computes the vector magnitude (and time derivatives).
Identity	Sets a square matrix equal to the identity matrix.
Do-t	Computes the vector dot product (and time derivatives).
Determinant 3	Computes the determinant of a 3x3 matrix.
Cross	Computes a vector cross product (and time derivatives).
Combine	Computes a linear combination of two vectors.
Angle	Computes the angle (in radians) between two vectors.
Add	Computes the component-by-component sum of 2 vectors or 2 matrices.
Taylor Series	Constructs a Taylor series function from a list of coefficients.
Compute value of Taylor series	Evaluates a Taylor series function at a point in time.
Compute n <sup>th</sup> derivative of Taylor series	Evaluates a derivative of a Taylor series function at a point in time.
Polynomial function	Constructs a polynomial function from a list of coefficients.
Compute value of polynomial function	Evaluates a polynomial function at a point in time.
Get n <sup>th</sup> derivative of polynomial function	Evaluates a derivative of a polynomial function at a point in time.
Normal distribution	Constructor for creating a normal distribution of a given mean value and standard deviation.
Get probability of normal distribution	Returns probability that a value is within a given range of a normal distribution.
Get probability of standard normal distribution	Return probability that a value is within the range [low high] in a standard normal distribution. This class supports fast probability calculations in a standard normal distribution via table look-up rather than via numeric integration.
Get nz	Return probability that a value lies in the range from -infinity to v in a standard normal distribution.
Derivative	Constructs a quaternion (and its derivative) corresponding to a coordinate transformation obtained by rotating about a coordinate axis at a specified angular rate; constructs a quaternion (and its first and second derivatives) corresponding to a coordinate transformation

<u>Function Name</u>	<u>Description</u>
	obtained by rotating about a coordinate axis at a specified angular rate and acceleration.
Quaternion	Constructs a quaternion from real and imaginary parts; from a series of three axes and Euler angles; from a 3-1-3 series of Euler angles; from a rotation about a coordinate axis; that is one of the purely imaginary basis quaternions, i, j, or k; that is the quaternion multiplicative identity; that is a purely real quaternion; that is a purely imaginary quaternion.
Euler	Constructs a 3-1-3 Euler angle rotation from 3 angles.
Six DOF	Constructs a six degree-of-freedom transformation from rotation and acceleration objects; from a quaternion, angular velocity, angular acceleration, cartesian position, cartesian velocity, and a cartesian acceleration; from a quaternion, angular rate, angular acceleration, spherical coordinates, spherical rate, and spherical acceleration.
Invert	Constructs a six degree-of-freedom transformation that is the inverse transformation.
Concat	Constructs a six degree-of-freedom transformation that is equivalent to successive applications of two different transformations.
Propagate	Constructs a six degree-of freedom transformation from another by propagating the position and rotation in time assuming a zero angular acceleration.

***Glossary:***

<u>Term</u>	<u>Definition</u>
DOF	Degrees of freedom

## Framework Package 18: MDS Directory Access Protocol

### ***Overview:***

The MDS Directory Access Protocol, or MDAP, package provides a lookup service for storing and retrieving deployment addresses. MDAP servers run as a separate process in a multi-deployment scenario. At startup, each deployment registers its address with the MDAP server.

### ***Description of basic functions:***

<u>Function Name</u>	<u>Description</u>
Add entry	Adds a name/address pair to the MDAP database.
Retrieve entry	Retrieves an address from the MDAP database given and name.

### ***Glossary:***

<u>Term</u>	<u>Definition</u>
MDAP	MDS Directory Access Protocol (See MDAP document)

## Framework Package 19: MDS Standard Library

### *Overview:*

The MDS Standard Library, or `MdsStd`, package provides utility functions that are used frequently throughout the MDS frameworks. `MdsStd` includes low-level utility classes and interfaces that are not already provided by our operating-system wrapper, the language's standard library, or the programming language itself. For example, smart pointers, file manipulators, and case-insensitive string comparison. Much of the functionality of this package is very similar to that found in several well-known C++ extension packages, such as Loki or Boost.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Caseless string	This function makes it easy to compare two character sequences without regard to the case of the letters in the sequences.
Reference-counting pointer	This function makes it easy for several different pieces of software to have access to a value without having to have their own copy of the value, and to automate releasing the memory associated with the value when no more references to it exist.
Auto-pointer	This function automates the releasing of dynamically allocated memory occupied by a variable when that variable is no longer needed.
Context string	This function makes it easy to read and manipulate character strings, such as a path string, that imply a hierarchy. Sub functions include breaking such a string into its component pieces.
Stream tokenizer	This function makes it easy to accept a stream of characters and have it automatically broken into single words.
Print formatter	This function defines a simple interface for formatting text reports.
Usage timer	This functionality makes it possible to time the execution of a function or code block.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
<code>MdsStd</code>	MDS Standard Library package

<u>Term</u>	<u>Definition</u>
Path String	A string that defines a folder in a computer's disk. For example, the string <code>'C:\windows\system\Program Files'</code> is a path string, and it implies a hierarchy: Folder 'C' contains folder 'windows', which in turn contains folder 'system', and so on.
Pointer	In C++, a variable that contains the memory location, or address, of another variable.

## Framework Package 20: Named Object Registry (Nor)

### *Overview:*

The Name Object Registry, or NOR, package provides an easily accessible capability for storing values with an associated name in a global registry. Once values are stores in a NOR registry, they can be used by NOR savvy functions.

This package also provides the ability to create 'contexts' or local regions. Each context contains a local registry for the value-names pairs can be that allows the registration of values under a name, but these associations are defined only within the context.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Register object	Creates an association in the registry between the given object and a name. The name can be supplied by the caller, or can be generated by this package. In either case, the name must be unique. The object can be subsequently looked up in the registry by name. The registration can be specified to be in the global registry, or in a registry that is contained within a context.
Deregister object	Removes the association between the given object and its name in the registry. The object must have previously been registered using the 'register object' function.
Find object	Given a name, finds the object that is registered under that name.
Create context	Creates (defines) a new context, optionally as a sub-context to an already-existing context.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Context	A scope or limit of definition for object registries. An example is a folder in a computer disk: the folder provides a context in which the names of the files in the folder are defined. Filenames must be unique within a single folder, but different folders may contain files of the same name.
NOR	Named Object Registry package
Object	In object-oriented programming, a value of a specific, usually user-defined, type.
Object Registry	A set of associations of a name to an object.

## Framework Package 21: Physics

### *Overview:*

The Physics package provides a set of common physics objects and computations. It contains object definitions for cartesian, cylindrical, and geodetic coordinate systems and methods for conversions between them. In addition, it contains representations and methods for positions and rotations, and their first and second derivatives; and methods for rotating positions and their derivatives. Finally, the Physics package contains object definitions and methods for ellipsoids and physics exceptions.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Cylindrical system to Cartesian	Converts a cylindrical position, velocity, and acceleration into Cartesian coordinates.
Cylindrical system from Cartesian	Converts a Cartesian position, velocity, and acceleration into cylindrical coordinates.
Geodetic system to Cartesian	Converts a geodetic position, velocity, and acceleration into Cartesian coordinates.
Geodetic system from Cartesian	Converts a Cartesian position, velocity, and acceleration into geodetic coordinates.
Spherical system to Cartesian	Converts a spherical position, velocity, and acceleration into Cartesian coordinates.
Spherical system from Cartesian	Converts a Cartesian position, velocity, and acceleration into spherical coordinates.
Rotation	Constructs a rotation from a coordinate axis, angle, rate, and acceleration; an axis (eigenvector of the rotation), angle of the coordinate rotation about that axis, rate, and acceleration; a quaternion and its derivatives; a quaternion, angular velocity and angular acceleration; a 3x3 rotation matrix and its first and second derivatives; a sequence of Euler angles and their derivatives.
Is within	Determines if one rotation is close to another.
Dangle and axis	Computes the angle and axis (Eigen vector) of a rotation and the axes and values of its derivatives.
Derivatives for Euler	Computes the angles and their first and second derivatives for an Euler factorization corresponding to a user specified sequence of coordinate axes.
Quaternions	Computes the quaternion associated with a rotation as well as the quaternion's first and second derivatives

<u>Function Name</u>	<u>Description</u>
Propagate	Computes the rotation propagated from a given rotation assuming constant angular velocity.
Matrices	Computes the rotation matrix and its first two derivatives corresponding to a given rotation.
Rotation concatenation	Computes the rotation corresponding to two successive rotations; and computes a rotated position, velocity, and acceleration.
Invert	Computes the multiplicative inverse of a rotation
Apply inverse	Computes the inverted rotation of a position, velocity, and acceleration.
Near point	Determines the point on the surface of an ellipsoid that is closest to a specified point.
Derivatives to normal	Determines the unit normal and its first and second time derivatives at a point with known velocity and acceleration on the surface of an ellipsoid.
Base of normal	Given a non-zero input vector and its first two time derivatives, determines the point (and its first two time derivatives) on the surface of the ellipsoid whose outward pointing normal is parallel to the input vector.

## Framework Package 22: Scheduler

### *Overview:*

The Scheduler framework package provides interfaces, components, and connectors specialized to support the design of multi-threaded software architectures. The fundamental aspect of this package is to define software interfaces to access operating system services needed for multi-threaded applications as well as encapsulating key operating system entities such as queues, semaphores and threads into components that interact via these interfaces with the rest of the application.

Since the design of multi-threaded software is fraught with many challenges and tradeoffs, this package emphasizes the separation of a thread into a policy and functions. A thread policy dictates the order of execution of functions relative to asynchronous events such as timers and message arrival.

This package supports the design of thread policies as behavioral state machines for which third-party design tools such as Matlab's Stateflow can be used. The Scheduler package defines a number of interfaces, components, and connectors that together provide a number of architecture elements for designing multi-threaded applications.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Event communication	Architecture elements to communicate events among components and for converting function calls into events and vice-versa.
Alarms	Architecture elements to define, manage, and use alarms that can occur at a relative time or at an absolute time.
Queues	Architecture elements to define, manage, and use queues.
Pump	Architecture elements to define, manage, and use operating system threads that govern the dispatching of events from queues according to a thread policy.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Queue	A list to store a number of items for processing. A queue is an asynchronous coupling mechanism that allows one thread to insert items into the queue independently and at different times than another thread that takes items from this queue.
Semaphore	An operating system entity to synchronize two different threads that must perform a rendezvous to coordinate progress in their respective computations.

<u>Term</u>	<u>Definition</u>
State machine	A model of computation suited for discrete software systems where there are different states in which computations take place. Explicit transitions among states coordinate the different computations among all states according to explicit input events and conditions.
Thread	An operating system entity that can perform a computation independently of other threads' computations in the system.
Thread policy	A state machine that governs the behavior of an operating system thread with respect to its use of scheduler-related entities such as queues and alarms.

## Framework Package 23: Serialization

### *Overview:*

The Serialization package provides functions needed to convert values to and from a series of bytes from transmission over a communications link. This package also provides a support for deserialization; i.e. the conversion of serialized data back to its initial value. These conversions are done in a platform-independent way; i.e. the value is preserved, even when platform-specific internal representations differ.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Data input stream	Functions that convert serialized simple values, such as numbers and character strings, from a series of byte values into the values themselves.
Data output stream	Functions that convert simple values into series of bytes in a platform-independent way.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Serialization	The conversion of a value from its internal representation in C++ or Java into a series of byte values, which can then be read by another piece of MDS software, regardless of which type of compute it is running on.
Deserialization	The opposite operation of serialization; the conversion of a series of byte values to their original C++ or Java values. For example, an 8-byte series may represent the real number 8.0. And the same 8-byte series would be translated, or deserialized, into the value 8.0 regardless of the kind of computer running the software that is reading the byte series.

## Framework Package 24: SI Units

### *Overview:*

The SI Units package provides data representations for scalars and associated mathematical operators.

The purpose of the package is to detect errors in expressions involving physical dimensions (mass, length, temperature, etc) and to eliminate errors involving units (e.g. meter versus kilometer versus mile). For example, an expression that adds a length to a temperature is dimensionally incorrect, and an expression that adds 2 meters to 3 feet requires an easy-to-forget unit conversion. Also, accidental transposition of two positional arguments in a function's argument list, such as voltage and current, can be detected as a dimensional error. Such errors are not uncommon in mainstream programming languages such as C++ and Java because they provide no built-in support for scalars.

“SI” refers to the international system of units<sup>2</sup> that defines dimensions and units of measurement for “scalars”, as well as physical constants. A *scalar* is a quantity such as mass, length, time, or temperature, completely specified by a number on an appropriate scale. For example, if John's height is 1.75 meters, the dimension is “length” and the number is “1.75” and the scale is “meters”. The same scalar can be equivalently represented as 5.74 feet, approximately.

Since units of measurement are part of physics, this package exists as a sub-package of the Physics package.

### Implementation Note

An implementation of this package faces several tradeoffs involving memory space versus processing time, compile-time versus run-time checking, and full coverage of SI versus size of object code. Given that the main objective of this package is to reduce sources of human error in dealing with scalars, the guiding principle should be to seek a design that is easy to use, with modest runtime overhead, so that the package gets used rather than avoided.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Arithmetic operators	Functions for arithmetic operations: addition, subtraction, multiplication, division, square, square root, etc.
Comparison operators	Functions to test for: equal, not equal, less than, less than or equal, greater than, greater than or equal.

---

<sup>2</sup> “SI” is an abbreviation for “Le Système International d’Unités”, the international system of units.

<u>Function Name</u>	<u>Description</u>
Dimensions	The package must support the seven base dimensions of SI—length, mass, time, current, temperature, amount of substance, and luminous intensity—and derived dimensions such as velocity, power, volume, frequency, force, radiation, etc.
Physical constants	The package should define scalar constants such as meter, kilogram, second, ampere, Kelvin, mole, candela, newton, hertz, pascal, watt, volt, lightspeed, etc.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Scalar	A quantity such as mass, length, time, or temperature, completely specified by a number on an appropriate scale.
SI Units	<i>System Internationale</i> , the international system of units.

## Framework Package 25: Simulation

### *Overview:*

The Simulation package provides tools that simplify the job of building simulation components.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Get data	This function allows remote deployments to get the sensor data from a simulated sensor device.
Command	This function allows remote deployments to issue commands to the simulated actuator device.
Run	This function from sim device allows the thread scheduler to schedule the sim device component for a run
Set request	This function allows the sim device via sim access bridge to communicate with a third-party software. It sends a request to the third-party software.
Get response	This function gets a response back from third-party software.
Translate request	This function translates the earlier request made by a set request function call into the form required by the third-party software tool to communicate
Translate response	This function translates response returned by the third-party software into an internal response data structure needed by the get response command.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
Sim	Truncated name for simulation.
Sim access bridge	A Sim access bridge contains a discrete value history for transactions, which record requests from and responses to a remote hardware adapter. In addition to providing this value history, a sim-access-bridge component provides the communications channel with a remote deployment, interpreting as needed to command and query a sim-device component.

<u>Term</u>	<u>Definition</u>
Sim devices:	Sim devices are state generators, with additional capability to provide an interface for commanding and data retrieval. Sim devices may either contain internal device modeling or communicate with third-party dynamic simulation tool in order to simulate discrete device states.
State generator:	Analogous to remote estimators, they generate state functions with which to update state variables. A state generator may contain either internal dynamics modeling or communicate with third-party dynamic simulation tool in order to simulate continuous physical states.
Sim Model Bridge:	A sim model bridge is a component through which simulation communications with external simulation models or third party tools. This capability provides a simulation model interface to implement a remote-procedure-call style interface between simulation components and external models.

## Framework Package 26: State Knowledge

### *Overview:*

The State Knowledge package provides a mechanism for representing state variables and their values. The concept of “state knowledge” is fundamental to the MDS architecture and refers to “what we know and how well we know it.”

State knowledge encompasses information such as device operating modes, device health, resource levels, attitude and trajectory, temperatures, pressures, etc, as well as environmental states such as the motions of celestial bodies and solar flux. Such information is maintained in state variables that serve, in a sense, as “Grand Central Station” because of the many activities that use state information, including estimation, control, planning, and telemetry.

This package provides general-purpose classes and methods for representing, accessing, and managing state knowledge. The three basic elements of state knowledge are state variables, state functions, and state values. State variables provide uniform methods for querying and updating state knowledge as well as managing the data stored within. State functions, which describe how a state’s estimated value varies with time, are used to update state variable timelines. State values are returned by state variables in response to a query for a particular instant of time. The classes for state functions and state values are abstract base classes that users extend to represent data in a suitable form; thus this package does not try to prescribe user data formats.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Update state	Updates the state of a state variable for the time interval specified in a supplied state function.
Get state	Returns the value of a state variable for a specified instant in time. A legitimate value is “unknown”, meaning that the state variable currently contains no information for that instant in time.
Notify listener	This callback notifies listeners that the state variable has been updated.
Policy control	Allows user to set/modify data management policies that govern how much data is to be retained, when to compress, when to mark for transport, when to checkpoint, how much to restore upon reboot, etc.
Constraint scheduling	Provides operations used during scheduling of executable goals to: determine if a goal is achievable, determine if a goal-to-goal transition is achievable, determine the earliest achievable merge of one goal with another, and project the effect of a goal forward in time.

<u>Function Name</u>	<u>Description</u>
Constraint execution	Provides two operations: one to check if a constraint is ready to start given current state, and one to start execution of the constraint.

***Glossary:***

<u>Term</u>	<u>Definition</u>
State value	The value of a state variable at an instant in time. If there is uncertainty in the value then it must be represented in some form as part of the value. Function 'get state' returns a state value.
State function	A function of time, bounded by a start time and end time that describes how a state varies over time, if at all. Function 'update state' takes a state function as an argument.
State variables	Variables used to provide uniform methods for querying and updating state knowledge as well as managing the data stored within.

## Framework Package 27: State Query

### *Overview:*

The State Query package provides the functionality to query state and measurement histories. This functionality includes the ability to submit queries from either text files or from a GUI application, and to receive query results in batch or real-time modes. The package also contains helper classes to make it easy to extend the query functionality for new kinds of state-data types, and an application to help a user generate an adaptation of the query library for a new kind of state data.

The package contains C++ and Java elements, and includes the functionality to communicate between C++ and Java deployments.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Create query	Implemented by a Java GUI application window, this function allows a human user to interactively build a query, by selecting a time range, type of query, and a set of states to be queried. This application can also store or read queries in text form to/from a file.
Submit query	This functionality allows the submission of a query for processing. Submission can be done by a human user via the query builder application, or by sending an Http command to a batch-mode Java application, which in turn locates the C++ deployments involved in the query and sends each of them the query.
Execute query	In C++ deployments, this function accepts a query from a Java application via the Embedded Web Server (in the form of an Http message). This function reads and analyzes the query, searches for the state objects to be queried, and retrieves the data. It then builds a response message containing the query results and sends it back to the requestor.
Generate queries	This functionality automatically searches in a C++ deployment for queryable state and measurement histories. It generates and executes a query for every one found. This is especially useful in testing.
Search states	This function is implemented as a Web command in a C++ deployment. It searches the deployment to find all queryable states, and returns the list of these to the requestor. This function is used by the Java query builder application to discover the list of available states from which the user can select.

## Framework Package 28: Time Management

### *Overview:*

The Time Management package provides mechanisms for the initialization and management of time services.

The Time Management package includes methods for defining time frames of reference, and transforming epoch values (values representing a fixed point in time) from one time frame to another. It also includes functions for comparing, adding, and subtracting epoch and duration values as well as functions for converting time values to and from human-readable format.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Initialize time	This function establishes the current time (by reading a parameter from an MDS configuration file), establishes the database of time frames available (e.g. TAI - international atomic time) and conversions between them, and establishes the default time frame.
Current time	This function allows a caller to get the current time, in the default time frame in use.
Read time	This function converts a human-readable time value into an internal epoch value. The time frame may be specified in the human-readable value, or if not, the default is used.
Write time	This function outputs an internal epoch value to a human-readable character string. The default format of the output is YYYY-HH-MM:SS.sss<time frame>, e.g. 2002-08-19:23.034TAI.
Set time frame	This function converts an epoch value from being expressed in one time frame to being expressed in another. If the conversion is not possible, the function generates an exception.
Add epoch and duration	Adds duration (which is an amount of time) to an epoch (a fixed point in time), and produces a new epoch. Duration values also are expressed in a timeframe, and so this operation may require a conversion of frames before doing the addition. An invalid frame conversion results in a thrown exception.
Compare epochs	This function allows the evaluation of logical expressions such as 'e1 < e2' or 'e1 >= e2', where e1 and e2 are epoch values.

## Framework Package 29: Value History

### *Overview:*

The Value History package provides container functionality for components that generate state knowledge.

State knowledge, which includes the state functions produced by state variables, measurements produced by hardware adapters or controllers, or other time-stamped data produced in simulation deployments, is stored in time-sorted containers. The Value History package provides insertion and retrieval functionality on these containers, and also provides an active interface to the Data Catalog and Data Transport capabilities.

Value History containers contain objects of user-defined types. The only thing all of these types have in common is that they are associated with a time, either a single point or a range.

There are two types of Value History containers: discrete history containers - for data items associated with a point in time, and intervallic history containers, for items associated with a range of time.

All Value History containers provide the functionality to install data policies on the container (see the Policy package description for a general description of policies). For example, a container might have a policy that says that every 100 inserts will trigger the creation of a Data Product containing the last 100 new items, and the transport of that product to the ground. Such a policy is called a transport policy.

There are other kinds of policies. For example:

- Compression policies, that control how large a value history container is allowed to grow in volatile memory;
- Retrieval policies, that control what a value history container attempts to retrieve from persistent storage at initialization time
- Persistence policies, that control whether and how often the value history saves its data into persistent storage.

The Value History package is designed to be extendable as far as what kinds of policies are available for use with a container.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Add item	This functionality adds a new item to a VH container. In the case of Intervallic items, a check is made to see if the new item is a 'continuation' of the last one inserted: if so, the two items are combined into one to save space.
Retrieve item	Retrieves an item covering a specific point in time from the container.

<u>Function Name</u>	<u>Description</u>
Retrieve last before	Retrieves the last item available with a timestamp not after the given timestamp. Available on Discrete Histories only.
Get items in range	Retrieves a contiguous subset of the VH container's content. If data is not in the container to cover the requested range, then an attempt is made to retrieve data from persistent storage (via the Catalog).
Install policy	Activates the given data policy on the container.
Command policy	Forces the named policy to be applied immediately.
Remove policy	Deactivates the named policy on the VH container.

***Glossary:***

<u>Term</u>	<u>Definition</u>
Item	Data record in a value history

## Framework Package 30: Virtual Real Time

### *Overview:*

The Virtual Realtime, or VRT, package provides the virtual real time ticks to a network of deployments—the capability makes it possible to control the speed at which time passes in a simulation testbed. Time can run slower or faster than wall clock time, and it can be started and stopped

On the embedded target deployment, virtual time ticks are measured by the embedded processor's CPU cycles. On a workstation-based simulation deployment, time advances by simulating the physics of objects. On a workstation-based operational deployment virtual time ticks are generated after arbitrary time advancement.

This package also provides time synchronization between various deployments.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Run	This function runs the network of deployments providing the virtual time ticks to the deployments. Internally the deployments are being synchronized at each time tick.
Abort Run	This function allows user to abort a run that was previously started using a run command.
Stop Run	This function stops the synchronization between deployments and lets the deployments run in a synchronization-free run mode. This function is called at the end of a test run in a workstation environment.
Run Status	This function provides run states, such as IDLE, RUNNING, STOPPED.
Start Virtual Clock	This function starts the Virtual clock and sets up the mode in which to operate – Master, Slave, Stand-alone. It is called at the initialization time after establishing the mode of operation set by the RWP parameters.
Shutdown Virtual Clock	This function is called at finalization to shutdown the Virtual Clock.
Get Virtual Real Time	This function provides current Virtual Real Time in Hours:Minutes:Seconds:Milli-Seconds format.

### *Glossary:*

<u>Term</u>	<u>Definition</u>
VRT	Virtual Real Time
Embedded processor	Processors designed to perform useful work while “embedded” in the physical environment.

<u>Term</u>	<u>Definition</u>
RWP	Read/Write parameters
CPU	Central Processing unit

## Framework Package 31: Visualization

### *Overview:*

The Visualization package provides the functions necessary to view the results of state queries. The capabilities of this package, include the ability to make graphical plots of state data, and to allow a user to configure a display based on values of the data. For example, a user may want the field in which a temperature value is displayed to turn red if the temperature goes above 180 degrees. This package also provides utility programs for converting binary query results into CSV (comma separated value) files, or for examining only a subset of the columns of data available in a binary result file.

### *Description of basic functions:*

<u>Function Name</u>	<u>Description</u>
Plot results	This functionality appears as a 'Plot' button in several of the screens of the visualization applications. It causes a plot of the state values in the selected data as functions of time. Each different curve in the plot is given a different color, and there is a legend describing the color assignment to curve as well as unit information. The X-axis and Y-axis are labeled with unit information (e.g., a single plot might have a temperature in degrees Fahrenheit and a distance in meters. In this case, the Y-axis would have both units in its label).
Snapshot results	This function displays data not as a plot, but simply in a tabular-like format in which each data value is displayed with its own timestamp. These types of displays are normally used with real-time query data - then the user sees individual fields updated as fresh data is received. These kind of displays can be color-coded to bring special attention to certain values for each field.
Export results	This function allows a user (or batch program) to write out binary query results into a CSV file, which can then be read by many applications, e.g. Microsoft Excel.
Read binary results	This function is used to allow applications to read in a previously stored file containing binary query results. After reading the data in, the user can plot or display or export the data.
Subset results	This function, available either as a stand-alone application or as a window under the main Visualization application (called 'Query Manager'), allows the user to pick a subset of the columns of data available in a result for viewing or export. Sometimes queries have many columns of data, and a plot of them all is hardly decipherable, so it's very helpful to be able to select just a few columns for viewing.

### 3 Index of glossary terms and abbreviations

accessor .....	12	MDAP .....	16, 36
base class .....	18	MdsStd .....	37
basis graph state variable .....	27	measurement .....	21
binding .....	23	node .....	27
CGI .....	17	NOR .....	39
class .....	18	object .....	39
component .....	10	object registry .....	39
composite graph state variable .....	28	path string .....	38
connector .....	10	pointer .....	38
constituent graph state variable .....	28	policy actuator .....	13
context .....	39	product .....	12
covariance matrix .....	20	product sender .....	12
CPU .....	56	proxy state variable .....	28
data product .....	12	quaternion .....	21
deployment .....	10	queue .....	42
derived relationship .....	27	RWP .....	56
deserialization .....	44	scalar .....	46
direct relationship .....	27	semaphore .....	42
DOF .....	35	serialization .....	44
dynamics model .....	20	SI Units .....	46
edge .....	26	sim .....	47
ELF .....	15	sim access bridge .....	47
embedded processor .....	55	sim devices .....	48
Euler angles .....	20	sim model bridge .....	48
EWC .....	16	singleton .....	32
EWS .....	17	software architecture .....	10
frame .....	27	state function .....	50
GEL .....	23	state generator .....	48
GNET .....	24	state machine .....	43
goal .....	23, 24	state value .....	50
goal network .....	24	state variables .....	50
graph .....	26	state vector .....	21
GREASE .....	20	temporal constraint .....	23, 24
GSV .....	27	temporal constraint network .....	24
hardware adapter .....	30	thread .....	43
HTML .....	16, 17	thread policy .....	43
HTTP .....	16, 17	time point .....	23
INIT .....	32	topological sort .....	32
initialization & finalization .....	32	vertex .....	26
interface .....	10	VRT .....	55
item .....	54	Xgoal .....	23
matrix .....	21		