A Generic Software Architecture for Prognostics (GSAP)

Christopher Teubert[1], Matthew J. Daigle[2], Shankar Sankararaman[3], Kai Goebel[4], Jason Watkins[5]

[1,2,4] *NASA Ames Research Center, CA, 94035, USA*
christopher.a.teubert@nasa.gov
matthew.j.daigle@nasa.gov
kai.goebel@nasa.gov

[3] *SGT, Inc, NASA Ames Research Center, CA, 94035, USA*
shankar.sankararaman@nasa.gov

[5] *University of California, Irvine, CA, 92697, USA*
watkins1@uci.edu

ABSTRACT

Prognostics is a systems engineering discipline focused on predicting end-of-life of components and systems. As a relatively new and emerging technology, there are few fielded implementations of prognostics, due in part to practitioners perceiving a large hurdle in developing the models, algorithms, architecture, and integration pieces. Similarly, no open software frameworks for applying prognostics currently exist. This paper introduces the Generic Software Architecture for Prognostics (GSAP), an open-source, cross-platform, object-oriented software framework and support library for creating prognostics applications. GSAP was designed to make prognostics more accessible and enable faster adoption and implementation by industry, by reducing the effort and investment required to develop, test, and deploy prognostics. This paper describes the requirements, design, and testing of GSAP. Additionally, a detailed case study involving battery prognostics demonstrates its use.

1. INTRODUCTION

Prognostics is a systems engineering discipline focused on predicting end-of-life (EOL) of components and systems. EOL predictions can be used to inform actions to maintain the safety and efficiency of that system, either through maintenance and repair or online control reconfiguration. Although a significant amount of research in prognostics technologies has been performed in recent years, there are few fielded implementations of prognostics. This is due, in part, to practitioners perceiving a large hurdle in developing both the models and algorithms, as well as the architecture and integration pieces.

In fact, there are at this time no known existing open software frameworks for applying prognostics. That said, there are some open prognostics tools, past prognostics research, and work in similar fields that form the foundational algorithms and tools for such a framework. One example is the open-source MATLAB Prognostics Model Library (Daigle, 2016b) and Prognostics Algorithm Library (Daigle, 2016a), which provide some core model and algorithm implementations, but no architecture or integration pieces. There are also some general frameworks or operating systems that can be adapted for similar functions, such as General Electric's Predix platform (General Electric, 2017), an industrial Internet-of-things platform; the Robot Operating System (ROS) (*Robot Operating System*, 2017), an open collection of software frameworks for robot software development; and NASA's Core Flight System (cFS) (NASA, 2017), a reusable software framework for NASA flight projects and embedded software systems.

This paper introduces the Generic Software Architecture for Prognostics (GSAP), a general, object-oriented, cross-platform software framework and support library for prognostics technologies. GSAP implements many of the functions and algorithms used in prognostics as part of a prognostics support library. The GSAP framework implements and enforces the prognostic process. A standard interface is supplied for integrating new models and algorithms, and for integrating the system into data sources (sensors) and sinks (displays, decision support tools, etc.). Users are then able to create a prognostic application by integrating their algorithms, models, and interfaces to their systems into the GSAP framework, with possible integration onboard or offboard a vehicle or other asset.

This paper describes the design, testing, and use of GSAP. First, Section 2 provides a general description of the prognostics problem. Section 3 describes the current solutions available for performing prognostics. Section 4 outlines the requirements for the GSAP system, and Section 5 outlines the design chosen to meet these requirements. The testing, verification, and validation of GSAP is described in Section 6. Section 7 describes a case study where GSAP was used. The final section, Section 8 summarizes the design and benefits of GSAP and provides a summary of future work.

2. PROBLEM DESCRIPTION

In general, the problem of prognostics is to predict the future evolution of a given system, and, in particular, to predict the time of some (typically undesirable) event. In mathematical terms, we are interested in how the *state* of the system, $\mathbf{x}$, will evolve in time, whether some subset of the state space, $\mathcal{X}$, will be reached in some finite time, and, if so, *when* it will be achieved (Goebel et al., 2017). For example, $\mathcal{X}$ may represent failure states, and the earliest time at which a failure state is reached is the EOL. A *threshold function* defines the boundary between failure and nonfailure states.

In the general case, there are many different subsets of the state space of interest representing different conditions, such as failure/nonfailure, safe/unsafe, etc. For a state $\mathbf{x} \in \mathbb{R}^{n_x}$, we define a set of events E that apply to the state space where for each $e \in E$ we define a threshold function, T_e, as

$$o_e(k) = T_e(k, \mathbf{x}(k), \mathbf{u}(k)), \tag{1}$$

where $k \in \mathbb{N}$ is the discrete time variable, $\mathbf{u}(k) \in \mathbb{R}^{n_u}$ is the input vector, and $o_e(k) \in \mathbb{B}$ is a Boolean variable indicating whether the event e has occurred or not. Typically, o_e is defined only as a function of the state, but in general, it may also be time-dependent and input-dependent.

For prediction, we want to determine when the current state of the system will evolve into some other state where e occurs. The state evolves according to a state equation:

$$\mathbf{x}(k+1) = \mathbf{f}(\mathbf{x}(k), \mathbf{u}(k), \mathbf{v}(k)), \tag{2}$$

where $\mathbf{v}(k) \in \mathbb{R}^{n_v}$ is the process noise vector, and $\mathbf{f}$ is the state update function.

The inputs to the prediction problem are the following (Sankararaman, Daigle, & Goebel, 2014):[1]

1. a time horizon of prediction, $[k_o, k_h]$;
2. the initial state probability distribution, $p(\mathbf{x}_o(k_o))$;
3. the future input trajectory distribution, $p(\mathbf{U}_{k_o,k_h})$, where $\mathbf{U}_{k_o,k_h} = [\mathbf{u}(k_o), \mathbf{u}(k_o+1), \ldots, \mathbf{u}(k_h)]$; and
4. the future process noise trajectory distribution, $p(\mathbf{V}_{k_o,k_h})$, where $\mathbf{V}_{k_o,k_h} = [\mathbf{v}(k_o), \mathbf{v}(k_o+1), \ldots, \mathbf{v}(k_h)]$.

[1] Here, for a vector $\mathbf{a}$, we denote a trajectory of this vector over the time interval $[k, k+1, \ldots, k_n]$ as $\mathbf{A}_{k,k+n}$.

Note that, in general, these inputs are dependent on the time of prediction, k_o. Further, $p(.)$ denotes the probability density function, and these probabaility density functions need to represent the uncertainty in the underlying quantities. The GSAP framework hence has systematic capabilities for uncertainty representation, as discussed later in this paper.

The prognostics problem, as solved by GSAP, is to predict the future states of the system within a time interval and determine the occurrence of a set of events:

Problem 1. Given a time interval $[k_o, k_h]$, an initial state $p(\mathbf{x}(k_o))$, process noise $p(\mathbf{V}_{k_o,k_h})$, and future inputs $p(\mathbf{U}_{k_o,k_h})$, determine $p(\mathbf{X}_{k_o,k_h})$, and for each event $e \in E$, compute $p(\mathbf{O}_{k_o,k_h})$ and $p(k_e)$.

Here, k_e is the time when $o_e(k)$ first evaluates to $true$, i.e.,

$$k_e(k) = \min\{k' : k' \geq k \textbf{ and } o_e(k) = true\}. \tag{3}$$

Further, additional variables, $\mathbf{z}(k)$, that can be expressed as functions of the state may also be predicted:

$$\mathbf{z}(k) = \mathbf{g}(\mathbf{x}(k), \mathbf{u}(k)), \tag{4}$$

where $\mathbf{g}$ is an output function. Thus, $p(\mathbf{Z}_{k_o,k_h})$ may also be predicted. Note that the predictions are also probability density functions (Sankararaman, 2015), and hence, the aforementioned uncertainty representation tools will also be important, from this perspective.

There are different methodologies to solve the Prediction Problem (Problem 1), and GSAP does not enforce one over another. These are typically categorized into model-based, data-driven, and hybrid (combined model-based/data-driven) approaches.

2.1. Model-Based Prognostics

In the model-based prognostics paradigm, prognosis is performed using a combination of a state estimation algorithm (often a Bayesian filter) and a prediction algorithm, both of which rely on a model of the monitored system (Orchard & Vachtsevanos, 2009; Daigle & Goebel, 2013; Saha & Goebel, 2009). In this case, the model must include the state equation (2), and an output equation:

$$\mathbf{y}(k) = \mathbf{h}(k, \mathbf{x}(k), \mathbf{u}(k), \mathbf{n}(k)), \tag{5}$$

where $\mathbf{y}(k) \in \mathbb{R}^{n_y}$ is the output vector, $\mathbf{n}(k) \in \mathbb{R}^{n_n}$ is the measurement noise vector, and $\mathbf{h}$ is the output equation.

The state estimation algorithm will estimate the state $\mathbf{x}(k)$ based on the model and the measured outputs $\mathbf{y}(k)$. The prediction algorithm will predict the evolution of the state to compute its future values, the occurrence of events, and additional variables of interest, $\mathbf{z}(k)$.

2.2. Data-Driven Prognostics

In the data-driven prognostics paradigm (Schwabacher, 2005; Schwabacher & Goebel, 2007), the mapping from $\mathbf{u}(k)$ and $\mathbf{y}(k)$ to k_e is learned offline, given historical data. The learned model is then used online to compute k_e given measured system inputs and outputs.

2.3. Hybrid Approaches

Hybrid approaches to prognostics combine model-based and data-driven techniques (Chen & Pecht, 2012). For example, a particle filter could be used to estimate the system state, and then k_e predicted using a learned neural network model.

3. EXISTING ARCHITECTURES

In this section, we describe existing prognostics software and architectures (Section 3.1), and software frameworks that share similar characteristics to GSAP (Section 3.3).

3.1. Prognostics Tools

Owing to it being a relatively recent technology, there are no known existing open frameworks for applying prognostics. That said, there are some open prognostics tools, past prognostics research, and work in similar fields that is relevant. These related works are highlighted in this section.

In late 2016, the Prognostics Model Library (Daigle, 2016b) and Prognostics Algorithm Library (Daigle, 2016a) were released as open-source software. The Prognostics Model Library is a MATLAB-based modeling framework, with a particular focus on defining and building models of engineering systems for prognostics. It includes a library of prognostics models for select components developed within this framework, which is amenable for use within prognostics applications for these components in relevant systems. The Prognostics Algorithm Library is a MATLAB-based suite of algorithms commonly used within the model-based prognostics paradigm. As such, it includes algorithms for state estimation and prediction, including uncertainty propagation. The algorithms rely on component models developed within the Prognostics Model Library as inputs and perform estimation and prediction functions. The library allows the rapid development of prognostics solutions for given models of components and systems. Further, since the models and algorithms are implemented as objects, it facilitates comparative studies and evaluations of various algorithms to select the best algorithm for the application at hand. These libraries both support the design, creation, and testing of prognostics algorithms and models, however, they do not address the problems of prognostic system design, architecture, and interfaces. They have formed the basis for the some of the models and algorithms that were implemented in C++ for GSAP, along with the model-based prognoser.

3.2. Generic Architectures

Several standards organizations have addressed the topic of system health management in general and prognostics in particular. For example, the Operations and Maintenance Information Open Systems Alliance (MIMOSA) has brought forward a specification in which a standard architecture "*for moving information in a condition-based maintenance system*" (MIMOSA, 2006). This work was motivated by the Navy's need to control increasing costs resulting from manpower and part of developing and handling proprietary software and hardware for maintenance purposes. The idea was that standardization of information exchange specifications within the community of Condition Based Maintainance (CBM) users would ideally drive the CBM supplier base to produce interchangeable hardware and software components. Similarly, the Society of Automotive Engineers (SAE) has a dedicated Integrated Health Management Committee that is concerned with disseminating information about Integrated Vehicle Health Management (IVHM) and to provide guidelines for use and implementation of this technology. To that end, several publications address the topic of architecture in general even though that information does not provide the detailed hands-on information for ready use. The International Organization for Standardization (ISO) has a working group on Prognostics that seeks to illuminate the underpinnings of prognostics (ISO/TC 108/SC 5 Condition monitoring and diagnostics of machine systems, 2015). Similarly, the IEEE has a working group on Prognostics for Electronics that will issue prognostics guidelines for this particular sub-field. Currently, a draft (IEEE RS/SC Reliability, 2016) is available that seeks to "'*classify and define the concepts involved in prognostics and health management of electronic systems, and to provide a standard framework that assists practitioners in the development of business cases, and the selection of approaches, methodologies, algorithms, condition monitoring equipment, and strategies for implementing prognostics for electronic systems*"'. Earlier, the IEEE developed standard AI-ESTATE with the purpose "'*to standardize interfaces for functional elements of an intelligent diagnostic reasoner and representations of diagnostic knowledge and data for use by such diagnostic reasoners*"' (IEEE, 2015). Within the aerospace community, the safety-critical working group RTCA SC-167 published DO-178B (RTCA, 1992). This document is a guideline dealing with the safety of safety-critical software used in certain airborne systems. Although technically a guideline for software assurance using a set of tasks to meet objectives and levels of rigor, it has been a de facto standard for developing avionics software systems which had to be considered when developing health management solutions for aerospace.

In 2001, researchers at Boeing presented a high-level reference architecture for intelligent vehicle health management (IVHM) (Keller et al., 2001). This architecture included a description of the required functions for IVHM in a layered

form. The authors recognized the importance of support or logistics functions, such as those represented in the signal processing and presentation layers of their architecture. This architecture is thorough but still only conceptual, stopping short of the detail required for implementation. The literature also includes other examples (Deal, Ryan and Kessler, Seth, 2011) where the requirements for an architecture are delineated. Common to all these efforts is the recognized need for a standardized mechanism to deal with systems health management. What has been lacking is an open-sourced architecture that can be readily deployed for a variety of applications.

3.3. Software Frameworks

The Robot Operating System (ROS) is a set of open-source libraries and tools to build robot applications (*Robot Operating System*, 2017). It includes algorithms common to robotic applications, drivers, and development tools. ROS uses packages, nodes, and services to provide functionality to robotic systems. The ROS framework is completely open, allowing users to contribute packages, nodes, or services. This is similar to the GSAP model, in which the framework is designed to be open so users can add technologies.

Core Flight Software (cFS) is an open-source platform and software framework for flight missions (NASA, 2017). It was originally designed by NASA for use with space missions but has since been adapted for use on aircraft. It allows users to contribute components that integrate into cFS using a standard interface. Common tools for flight missions are implemented and incorporated into the software, and there have been some efforts to extend cFS for specific applications. One such example is the Autonomy Operating System (AOS), a NASA effort to extend cFS to support autonomy aircraft operations.

Predix is a general closed-source Industrial Internet of Things (IIoT) platform developed by General Electric (GE) (General Electric, 2017). It provides data management and data science tools to industry and includes a marketplace of contributed applications that provide additional capabilities. It is modular and has a standard interface for developers to add technologies.

ROS, cFS, and Predix are each platform to support a wide number of capabilities in their target environment. GSAP is a platform that instead supports a particular capability, i.e., prognostics, in a wide variety of target environments. By targeting a very specific application, GSAP can provide many advanced features for prognostics. Further, GSAP, like each of these platforms, is modular and supports the development of new tools or modules. A GSAP application could potentially integrate as an application into any of these platforms by adopting the necessary interfaces.

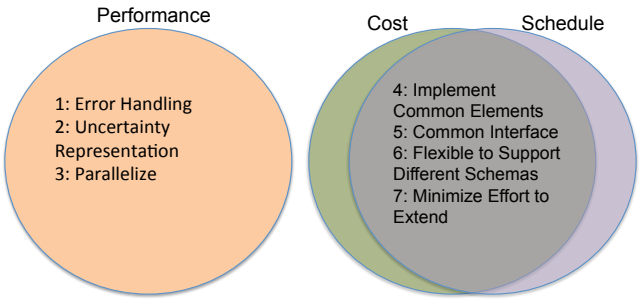

Figure 1. Goals and Top-Level Requirement Correlation

4. REQUIREMENTS

The development of requirements for prognostics systems continues to be an important and popular topic of research (Goebel et al., 2017; Saxena et al., 2010, 2012; Leao, Yoneyama, Rocha, & Fitzgibbon, 2008; Usynin, Hines, & Urmanov, 2007). Defining clear and complete requirements for prognostics applications is important. All top-level requirements for a system can be thought as deriving from performance, cost, or schedule goal (Saxena et al., 2012; SAE Aerospace, 2017).

Similarly, for GSAP, the purpose is to provide a tool to be used in prognostics applications, improving the performance, cost, and schedule of those applications. In this context, the goals of GSAP are:

Goal 1. Improve the performance of prognostics applications.

Goal 2. Reduce the cost of creating prognostics applications.

Goal 3. Improve the schedule for creating prognostics applications.

These goals translate into GSAP's top-level requirements. A list of the top-level requirement topics is included below. The flow down from goals can be found in Fig. 1.

Error Handling Provide basic error handling and reporting for improved program resilience and debugging

Uncertainty Representation Provide a manner of representing uncertainty

Parallelize Parallelize operations whenever possible for increased performance and reduced dependencies between the performance of separate program elements.

Implement Common Elements Implement the common algorithms and functions of prognostics applications

Common Interface Provide a common interface for integrating new technology

Flexiblity to Support Different Schemas Support different forms and methods of performing prognostics

Minimize Effort to Extend Reduce the effort required to extend GSAP with new prognostics technologies

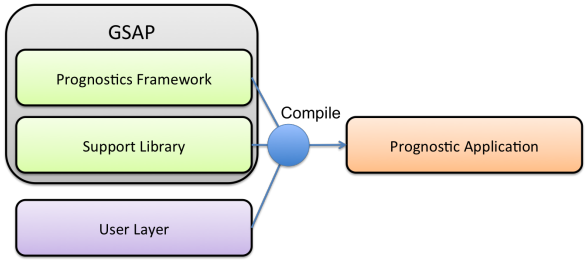

Figure 2. Using GSAP

A list of non-functional requirement topics can be found below:

Compliance with NASA and industry standards and practices (National Aeronautics and Space Administration, 2017, 2014; CMMI Product Team, 2010).

Documentation in an accurate, and efficient manner to support ease of use.

Open-Source GSAP to provide the greatest impact.

Extendable to support new prognostics targets, types of prognosis, and integration into new systems.

Scalable to support large applications where many systems are being monitored in parallel.

See Appendix B for complete GSAP requirements.

5. DESIGN

This section describes the overall design of GSAP. At the highest level, GSAP consists of two parts: The Prognostics Framework and the Prognostics Support Library. These are compiled with any user layer contributions to build a prognostics application, as shown in Figure 2.

In the remainder of the section, we describe the design in detail. Section 5.1 describes the design patterns utilized in GSAP. Section 5.2 and 5.3 describe the prognostics framework and support library, respectively. Section 5.4 describes the user layer. The ModelBasedPrognoser is described in Section 5.5.

5.1. Design Patterns

GSAP leverages many well-established design patterns. Understanding these patterns is essential for understanding the design of GSAP. This section provides a brief description of some of the design patterns found in GSAP.

5.1.1. Singleton

Singletons are classes of which only a single instance exists, which is shared by all program elements that need to use the class. GSAP uses singletons in situations where a static class might be appropriate, but there is significant code that can be shared between classes, making inheritance useful. By creating a singleton, we achieve the benefits of shared code via inheritance while allowing all elements using the class to share state by using a single instance.

Examples of singleton classes in GSAP include the various Factory classes, the ThreadSafeLog, and the CommManager class. Singletons in GSAP inherit from the singleton abstract base class (ABC).

5.1.2. Facade

A facade class simplifies access to a larger body of code. GSAP uses a facade to wrap the complexities of cross-platform networking code into the TCPSocket and UDPSocket classes. These classes expose all of the network communication functionality that GSAP requires in a simple cross-platform class for each protocol where the users need only concern themselves with the `Send` and `Receive` methods of the socket classes.

5.1.3. Abstract Base Class

Although not strictly a design pattern, abstract base classes (ABCs) are a design strategy used extensively by GSAP. Abstract base classes define an interface and shared core functionality of an object while leaving the concrete implementation of the functionality represented by the ABC to be implemented in virtual methods overridden in inheriting classes. This concept is used in conjunction with the factory pattern described in Section 5.1.4. Factories need only be aware of the ABC; it does not need to know anything about implementing classes.

There are several ABCs in GSAP. These classes include CommonCommunicator, CommonPrognoser, Model, Observer, and Predictor. This is the main mechanism by which a user can extend GSAP Each of these classes defines an interface that users can implement their inheriting classes to extend GSAP with custom technologies.

5.1.4. Factory

Factory classes provide a method for instantiating objects indirectly. GSAP uses factories extensively for classes in the framework which are likely to be implemented in practice by the end user extending an abstract base class provided by the framework or support library. By allowing the end user to extend a base class and then register the extended class with the factory, the framework can use user-supplied classes specified in configuration files without being directly aware of their concrete existence.

A large part of the flexibility of GSAP is derived from the extensive run-time configuration options provided by the framework. As such, factories are used to instantiate a large number of the framework and support library classes.

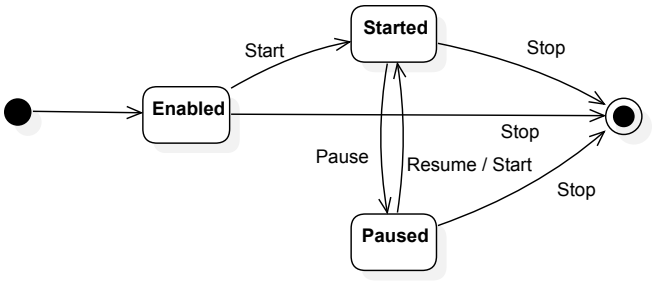

Figure 3. GSAP Prognoser Control States

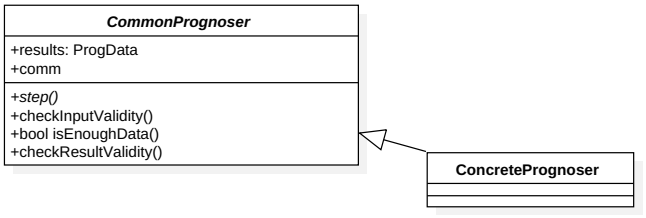

Figure 4. Prognoser Design

Table 1. Common Prognoser Configuration Parameters

Key	Description
type	The type of prognoser (ex: modelBasedPrognoser)
name	The name of the component being prognosed (ex: battery1)
id	A unique identifier for the piece of hardware being prognosed (ex: Serial Number)
histPath (Optional)	A path for the history files
inTags (Optional)	A list of tags expected from communicators
resetHist (Optional)	A flag to reset the recorded history for the component. Will archive the current history file and start a new one

5.2. Prognostics Framework

The framework contains the core functionality for running GSAP, including the logic to initiate and run prognostics. The primary class is the ProgManager. Initialization and control of GSAP applications is achieved through this class, as described in Section 5.4.2. The ProgManager reads the main configuration file, starts the CommManager class, and creates the prognosers, each on a dedicated thread. The ProgManager also handles the receipt of control commands, such as start, stop, pause, and resume. Figure 3 shows the different operational states of a GSAP application. Prognostic steps are only executed when the application is in the $Started$ state.

On creation, the CommManager initializes the communicators specified in the main configuration file, configured according to their particular configuration file (see Section 5.4.1). It then handles the following: 1) update and receipt of sensor data by the communicators, 2) requests for sensor data by the prognosers, 3) update of prognostics results by the prognosers, and 4) requests of prognostics results by the communicators. Upon receipt of the stop command from the ProgManager, the CommManager will stop and clean up all the communicators.

Perhaps most importantly, the framework includes all the interfaces for each of the modules in the user layer. These come in the form of Abstract Base Classes (ABC), as described in Section 5.1.3. Each ABC defines the interface with which the higher-level framework communicates with the concrete inheriting classes. For example, a concrete prognoser could be created to inherit from its ABC, CommonPrognoser. The concrete prognoser would then be required to implement the virtual methods defined in the ABC. They would also have the option to implement the optional virtual methods which would provide additional advanced features. Implemented optional methods override the implementation present in the ABC.

5.2.1. Prognosers

This is the core of the GSAP system. Inside the prognosers is the core logic for performing prognostics. A new prognoser is created to support a new method for performing prognostics. Many prognostics systems follow a standard model-based structure. Those systems do not require the creation of a new prognoser, only the creation of a new model that will be used by the ModelBasedPrognoser.

All prognosers must inherit from the base CommonPrognoser class (see Figure 4). Prognosers must have a constructor and a step function, but they can optionally also implement the other virtual member functions `checkInputValidity()`, `isEnoughData()`, and `checkResultValidity()`. In the constructor, the prognoser is configured from a map of the configuration parameters from that prognoser's configuration file. This configuration map is received as a constructor argument.

Each prognoser has a configuration file, which defines the configuration of that prognoser. Individual prognosers can have custom configuration parameters, which are documented with the prognoser. A list of the configuration parameters common to all prognosers can be seen in Table 1.

When the prognoser is started, it is initialized with the last state for that component (with the same id), if prognostics has been run before on that component. This is done using prognostic history files, saved in the path identified as histPath in the Prognoser Configuration File. Each specific component has a history file, identified by the component's unique id (the id field in the Prognoser Configuration File).

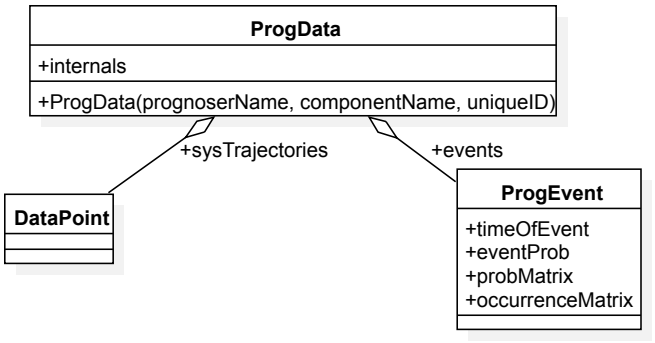

Figure 5. Prognostic Data Structure

The state is saved to this file periodically while the prognoser is running and on termination of GSAP. If the user would like to reset the history file, the resetHist flag can be used. This is done if maintenance or a change of configuration has occurred, causing the configuration file to no longer accurately reflect the state of the system.

The `step` function is called once every timestep. Inside the step function, the prognosor can access sensor data, perform calculations, and populate a ProgData (see Section 5.2.2) structure with the results. For an example of an empty prognoser, with explanations of each member function, see Section A.1.

The optional functions `checkInputValidity` and `checkResultValidity` are called once a timestep. These functions can be filled with basic sanity checks, to see if the input or results make sense. The most basic form of these is a check to make sure that the sensor values fall in a certain range. These checks can spot sensor problems that might lead to an incorrect prognosis. If they are not included with the prognoser, no check will be done.

The also optional `isEnoughData` function is used to determine if there is enough new, valid data for prognosis. This function returns a bool. If false is returned, prognostics will not be run that timestep.

GSAP is currently distributed with one prognoser, the ModelBasedPrognoser. This prognoser defines the model-based prognostics paradigm. Using this prognoser requires users to supply a model of the system. The exact workings of this prognoser are defined in Section 5.5.

5.2.2. Prognostic Data Structure (ProgData)

The Prognostics Data Structure, ProgData, is a class for storing and accessing the results of prognostics in a standard way. Each prognoser has a ProgData structure, which it fills with the results. Communicators can then access the progData structures for publishing results. The structure of ProgData can be seen in Figure 5.

At the highest level, each prognoser has a prognoser name, component name, and unique identifier. The prognoser name is the type of prognoser used, while the component name is a name for the particular component targeted. The unique identifier is an identification string or number unique to that physical component (e.g. serial number). The unique identifier should change if the component is replaced. For example, if a battery is being prognosed it might have the prognoser name "Battery", the component name "Batt5" (referring to the specific battery location), and a unique id "1394snvsdoe2" for that specific battery. These values can be set in the constructor or using the `set*()` methods (e.g. `setPrognoserName()`).

Table 2. Prognostic Events

Field	Description
timeOfEvent	The time the event will occur with uncertainty (`UData` type).
eventProb	The scalar probability of event occuring within the prediction horizon.
probMatrix	The probability that the event will occur at each time stamp. A one degree vector where `probMatrix[x]` is the probability that the event will occur at `time[x]`.
occurrenceMatrix	A 2-dimensional matrix storing if an event has occured for each sample. Is a time x unweighted samples matrix, so that `occurrenceMatrix[0][7]` represents if the event has occured for sample 7 at time 0.

The ProgData structure contains a structure of system trajectories. These correspond to the predicted outputs $\mathbf{z}(k)$ and its corresponding trajectory $\mathbf{Z}_{k_o,k_h}$. These system trajectories are typically used to store information about system variables that measure the degradation or fault severity. Common examples of these are State of Health (SOH) or State of Charge (SOC) for batteries. Potentially, these could also include operating efficiency or any other derived quantity that represents performance. System trajectories are accessed by name, e.g., `pData.systemTrajectory["SOH"]`. The value of the system trajectory is stored with uncertainty as a UData object (see Section 5.3.1).

The ProgData structure also contains a structure of prognostic events that can occur, i.e., E as defined in Section 2. Each prognostics event has a name (e.g. EOL) and time unit (e.g. cycles). Events are accessed by name, for example `pData.events["EO`
The information for each event is listed in Table 2. Each event must populate the timeOfEvent field. The other fields are optional, each field providing additional information about event probability with uncertainty. For example, the event occurrence trajectories $\mathbf{O}_{k_o,k_h}$ (see Section 2).

The internals field is used to store any information internal to the prognoser that is not to be published. Everything in the

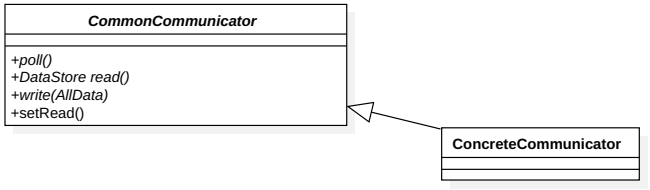

Figure 6. Communicator Design

internals field is still saved to the history file for reinitialization. Users have an option to save internal parameters that are not important for reinitialization as member variables, but note that these will not be saved between runs.

5.2.3. Communicators

Communicators are used to communicate data with the outside world. These function as interfaces with various data sources and sinks. Some examples could be a playback agent that reads from a file, a GUI for displaying prognostic results, an automated report generator, or a client that connects to a network messaging system (e.g., SCADA, CAN). These systems can receive data which will be used by prognosers or communicate the results with operators.

All communicators must inherit from the base CommonCommunicator class (See Figure 6). Communicators must have a constructor, a poll, a read, and a write function. In the constructor, the communicator is configured from a map of the configuration parameters from that communicator' configuration file. This configuration map is received as a constructor argument.

The `poll()` function is called periodically to check if there is more data to read. If there is data to read, the communicator will call the `setRead()` function to indicate that, and the read function will be called. In the `read()` function the communicator will fill a `DataStore` structure with any received data. The `DataStore` structure will then be returned. The `write()` function is used to communicate prognostic results to data sinks. For an example of an empty communicator, with explanations of each member function, see Section A.2.

Each communicator has a configuration file, which defines the configuration of that communicator. Individual communicators can have custom configuration parameters, which are documented with the communicators. Only one configuration parameter is common to all communicators, the `type`. Like for prognosers, this is used to specify the specific communicator to be created (e.g. `type:playback`).

GSAP is currently distributed with three simple communicators: Playback, Recorder, and Random. These are basic functions that are often performed in many prognostics applications. Additionally, many tools such as cross-platform TCP/UDP socket classes, which aid in the development of other common communicators, are supplied in the Prognostics Support Library. The included communicators are described below.

Playback This communicator is used to read sensor data from a delimited file. The file must include a header row with the names of each value.

Recorder This communicator is used to record sensor data and prognostics results to a delimited file. The file includes a header row with the names of each value.

Random This communicator is used for testing. It sets the value of every sensor to a random number in a range specified in the configuration file.

5.3. Prognostics Support Library

The support library is a collection of tools to support prognosers, communicators, and the framework. Each of these classes provides functionality to GSAP that can be used as part of a GSAP application, or externally with other prognostics products. These classes are all thoroughly tested and designed to be light, cross-platform, and easy to use. The support classes included and a description of their use are given in Table 3.

5.3.1. Uncertainty Representation

The GSAP support library includes tools for the representation of variable uncertainty, through the "UData" class. A single random variable can be expressed using either a parametric distribution type or a non-parametric distribution type. To describe a parametric distribution, it is necessary to know the distribution type (Gaussian, log-normal, etc.) and the distribution parameters (for example, mean and standard deviation in the case of a Gaussian distribution). A non-parametric distribution can be expressed regarding unweighted (i.e., equally-weighted) samples, weighted samples, percentiles and percentile values. A vector of random variables can be statistically dependent and follow varied distribution types; presently, the generic architecture supports only the inclusion of a vector of random variables when each variable is individually Gaussian, in which case, the statistical dependence can be completely expressed using the covariance/correlation matrix.

Below is an example using a parametric type:

```
1. UData u(MEANSD);
2. u.dist(GAUSSIAN);
3. u[MEAN] = 10;
4. u[SD] = 0.5;
5. printf("u(%f,%f) was last updated %f\n",
      u[MEAN], u[SD], u.updated());
```

In this example, a parametric UData object of uncertainty type mean with standard deviation. On line 2 the distribution type is set to gaussian. The mean is set to 10 and the

Table 3. General Support Classes

Function	Description
Logger	A singleton logger with multiple message verbosity levels designed for a multi-threaded environment. Logging messages is similar to use of printf
Configuration Map	A map data structure for loading, accessing, and parsing configuration from a key:value1,... style file
TCP Socket & Server	A class for connecting, sending, and receiving TCP stream data over a network
UDP Socket	A class for connecting, sending, and receiving UDP datagram data over a network
Singleton	A superclass for singleton classes. Enforces the interface for the singleton design pattern
Factory	A superclass for factory classes. Enforces the factory design pattern for creating new objects
UData	Uncertain Data Structure Classes - Classes used for storing, distributing, and manipulation data with uncertainty
Thread	A superclass for classes with a dedicated thread. Enforces finite state machine status's and standard control interface
Statistical Tools	A set of statistical tools
Matrix	A class for storage and manipulation of an arbitrary 2-D MxN matrix of doubles
Exceptions	Custom exceptions for GSAP applications
Benchmark	A prognoser for benchmarking GSAP applications

standard deviation is set to 0.5 (lines 3-4). The value of u and the time it was last updated are printed on line 5.

Below is an example using a non-parametric type:

```
1. u.uncertainty(SAMPLES);
2. u.npoints(100);
3. u.set(sampleVec);
4. u[22] = 3;
5. for (auto & sample : u) {
6.  // Some action
7. }
```

On line 1 and 2, the uncertainty type is set to samples (unweighted) with a size of 100 samples. The samples are set to the contents of a vector, and the 22nd sample is overwritten with the value 3. In lines 5-7 the samples are iterated through. Line 6 can be replaced with any action acting on the single sample.

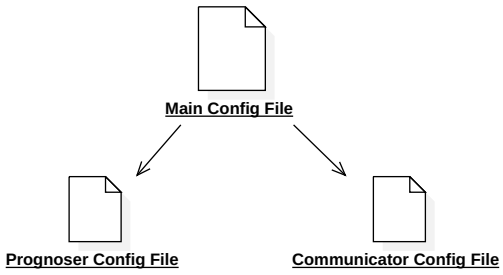

Figure 7. Configuration File Heirarchy

5.3.2. Algorithm and Model Libraries

The Prognostics Support Library also includes a selection of common algorithms used with Prognostics, including observers, predictors, and system models. These algorithms and models are described in Section 5.5.

5.4. User Layer

The user layer consists of the variable sections, or "hot spots" of GSAP (Srinivasan, 1999). Users create new modules in this layer to configure GSAP to their specific application.

Users must have a Prognostics Application Driver (PAD) (see Section 5.4.2), to start GSAP and configure it. Users can also configure their prognostics application through the creation of custom prognosers and communicators, described in the following sections. The prognosers and communicators use the facade design pattern, supplying a parent class with a predefined interface. Users create classes, which derive from these parent classes and implement the required functions.

5.4.1. Configuring

GSAP uses a two-layer hierarchical structure of configuration files to configure all the components to a specific application, as seen in Figure 7. These configuration files include a list of key:value pairs. Comments can be included by starting a line with a '#' character.

The top-level configuration file contains all the parameters for configuring the ProgManager. The most important of these parameters is the identification of the configuration files for the prognosers and communicators, the second layer of configuration files from Figure 7. For each configuration file specified the ProgManager would create a new prognoser or communicator and configure it to the parameters in its respective configuration file. The configuration parameters currently supported at the top level are listed in Table 4.

An example top-level configuration parameter can be seen below:

```
# Example Configuration File
Prognosers:Batt1.cfg,Batt2.cfg,Motor1.cfg
Communicators:Recorder.cfg, Playback.cfg
histPath:../test/validation/hist
```

Table 4. Top Level Configuration Parameters

Key	Description
Prognosers	A list of the prognoser configuration files to use. A prognoser will be made for each configuration file in the list
Communicators	A list of the communicator configuration files to use. A communicator will be made for each configuration file in the list
commmanger.step_size (Optional)	Wait time between iterations of the commmanager in milliseconds

```
commmanger.step_size:1000
```

Configuration for prognosers and communicators are described in Sections 5.2.1 and 5.2.3, respectively.

5.4.2. Prognostics Application Driver

The Prognostic Manager must be created and called by some higher level application. This is the only custom code strictly required to build a custom application if the user is using the provided prognosers and communicators. In this document, this is referred to as the Prognostics Application Driver (PAD). The primary role of the PAD is to create and start the Prognostics Manager, which will initiate prognostics. To do this requires the following two lines:

```
ProgManager PM = ProgManager(ConfigFile);
PM.run();
```

Here the first line creates the ProgManager and configures it to use the configuration file specified (here, shown as a string variable `ConfigFile`). The second line starts the prognostic manager.

Optionally, users can include their components to extend GSAP capabilities. These come in the form of custom Communicators, Prognosers, Models, Observers, or Predictors, as described in the following sections. When custom components are used, it must be registered with its respective factory. An example for a custom prognoser is included below. In this case, an ExamplePrognoser was registered with the name "example". Now, if users were to specify to use the prognoser "example" in the type field (i.e. `type:example`) of the prognoser configuration file, an ExamplePrognoser will be created. This is one example, but the same can be done with the CommunicatorFactor, ModelFactory, ObserverFactory, and PredictorFactory.

```
PrognoserFactory & progFactory =
    PrognoserFactory::instance();
progFactory.Register("example",
    PrognoserFactory::Create<ExamplePrognoser>);
```

Users can also add a search path for configuration files using the addSearchPath command, as shown below. This will allow configuration files to be specified by filename without their path.

```
ConfigMap::addSearchPath("../example/cfg/");
```

GSAP will automatically log the steps being taken and any errors to a log file. The verbosity of that logging can be done by setting the verbosity in the PAD (remember to include `"ThreadSafeLog.h"`), as shown in the below example.

```
Logger::SetVerbosity(LEVEL)
```

Here $LEVEL$ can be replaced with the level of log to print, ranging from LOG_TRACE (most verbose) to LOG_OFF (no log kept). It is suggested that you keep the log at LOG_INFO, unless you are actively debugging a problem. Note, that the log file size can grow quickly if kept at a high verbosity.

An example of an empty PAD can be found in Appendix A.6.

5.4.3. Extending

New prognosers and communicators can be added to extend the capabilities of GSAP. This is done using the interfaces described in Sections 5.2.1 and 5.2.3, respectively. A new Prognoser can be developed to perform prognostics using a method not already supported by the included ModelBasedPrognoser. This allows the framework to be agnostic to the specific method chosen for performing prognostics. Additionally, new communicators can be created to integrate with new data sources and sinks, allowing the framework to be application agnostic.

5.5. Model-based Prognoser

A common approach to prognostics is the model-based prognostics paradigm (Orchard & Vachtsevanos, 2009; Daigle & Goebel, 2013; Saha & Goebel, 2009), where prognosis is performed using a combination of a state estimation algorithm (often a Bayesian filter) and a prediction algorithm, both of which rely on a model of the monitored system. Because this is such a common approach, a `ModelBasedPrognoser` class, implementing the `Prognoser` interface, is provided by GSAP.

A model-based prognoser contains three main objects: a `PrognosticsMo` an `Observer`, and a `Predictor`. These all define interfaces, and the user provides specific instantiations of each of these, e.g., a battery model, an unscented Kalman filter (UKF), and a Monte Carlo-based predictor. Its configuration parameters are given in Table 5.

The `step` function of the prognoser performs the same actions, regardless of the specific components that comprise the prognoser. Thus, a user needs only to appropriately configure the prognoser and does not need to write any new code. The `step` function performs the following actions:

1. Retrieve the system inputs and outputs from the given

Table 5. ModelBasedPrognoser Configuration Parameters

Key	Description
model	Name of Model
observer	Name of Observer
predictor	Name of Predictor
Model.event	Name of system event to predict
Predictor.numSamples	Number of samples for prediction
Predictor.horizon	Time horizon of prediction
Model.predictedOutputs	Names of system predicted outputs
inputs	Names of system inputs from sensors
outputs	Names of system outputs from sensors

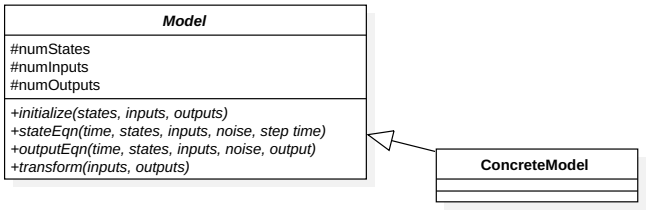

Figure 8. Model Design

data.

2. Run a step of the observer, given the inputs and outputs.
3. Retrieve the updated state estimate from the observer.
4. Run a step of the predictor, using the updated state estimate as the initial condition.

The details of the components are described in depth in the following sections.

5.5.1. Model

The `Model` class is essentially a wrapper around a discrete-time dynamic system model. It defines the system states, outputs, inputs, and specifies the interfaces for state update equation and the output equation. A system model can be created by deriving from this abstract class and implementing the state and output equations, i.e., Eq. 2 and Eq. 5, respectively.

For an example of an empty model, see Section A.3.

5.5.2. PrognosticsModel

THe `PrognosticsModel` class extends the `Model` class with the threshold equation (Eq. 1), the predicted output equation (Eq. 4), and an input equation:

$$\mathbf{u}(k) = \mathbf{i}(k, \boldsymbol{\theta}_u(k)), \tag{6}$$

where $\mathbf{i}$ is the input function, and $\boldsymbol{\theta}_u(k)$ is a set of "input parameters" that specify how to determine the inputs at a given time t. This is the method in which the future input trajectory and its distribution are specified, following the approach in (Daigle & Sankararaman, 2013), in which the input parameters are a form of surrogate variables. To specify a distribution of future input trajectories, we specify the distributions of the input parameters and sample from these in order to sample a future input trajectory distribution, as computed by the input equation.

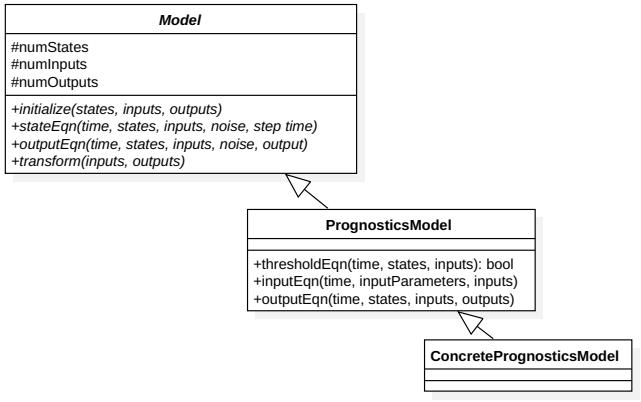

Figure 9. Prognostics Model Design

Included in GSAP is an electrochemistry-based prognostics model for a battery, based on the model developed in (Daigle & Kulkarni, 2013). The states, outputs, and dynamical equations are implemented as described in (Daigle & Kulkarni, 2013). The sole predicted output is the battery state of charge (SOC), and the predicted output equation computes SOC from the battery state. The event being predicted is the end of discharge (EOD), and the threshold equation compares the voltage to a lower limit to determine if EOD is reached. The future input trajectory is specified as a sequence of magnitude and duration values: the required battery power is linearly interpolated between consecutive magnitude values and each segment lasts for the corresponding duration value. So, for example, the set of input parameters $5, 60, 10, 30, 8$ specifies a ramping of the power from 5 to 10 W lasting 60 s, followed by a ramp down to 8 W lasting 30 s, and then being held at 8 W until EOD. A user specifies distributions for these values, and a Predictor would sample from those distributions to get a single input trajectory as computed by the input equation with those parameters.

5.5.3. Observer

The `Observer` class implements an interface for state estimation algorithms. It encapsulates a `Model` and the current state estimate, and implements a `step` function, which updates the current state estimate with new measured system inputs and outputs.

The `step` function, shown below, takes the current time, system inputs, and system outputs as arguments. Running the function updates the internal state estimate, which is a UData

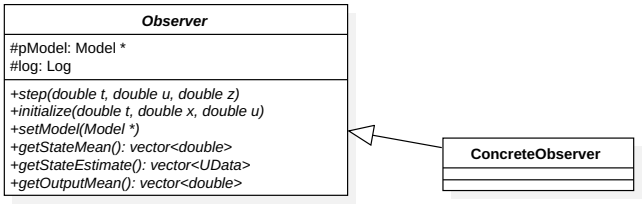

Figure 10. Observers Design

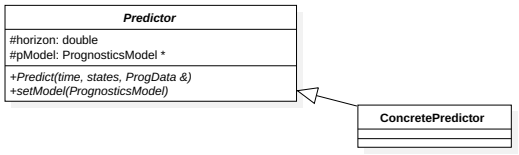

Figure 11. Predictor Design

object that can be retrieved with the `getStateEstimate` function.

```
void step(const double newT,
          const std::vector<double> & u,
          const std::vector<double> & z);
```

For an example of an empty observer, see Section A.4.

Currently included in GSAP are two different observers, the unscented Kalman filter (UKF) (Julier & Uhlmann, 2004) and the particle filter (PF) (Arulampalam, Maskell, Gordon, & Clapp, 2002; Doucet, Godsill, & Andrieu, 2000). Both are recursive Bayesian filtering algorithms for nonlinear systems. The UKF represents the state estimate using a set of deterministically selected samples using the unscented transform, whereas the particle filter represents the state estimate using a set of stochastically generated samples. In the future, other commonly used state estimation algorithms will also be added (Daigle, Saha, & Goebel, 2012; Daigle, 2016a).

5.5.4. Predictor

The `Predictor` class implements an interface for prediction algorithms. It encapsulates a `PrognosticsModel` and predicted data (e.g., EOL, predicted outputs trajectories), and implements a `predict` function, which takes a state estimate and updates the predicted data.

The `predict` function, shown below, takes the current time of prediction, the state estimate (from an Observer, for example), and a ProgData object as input arguments. The prognostics results are written to the ProgData object, which passed in by the ModelBasedPrognoser.

```
void predict(const double tP,
             const std::vector<UData> & state,
             ProgData & data);
```

Currently, GSAP provides a `MonteCarloPredictor`, which randomly samples the state estimate, the process noise, and the input parameters, and simulates a specified number of samples for a fixed time horizon (Daigle & Sankararaman, 2016). The uncertainty related to the process noise and input parameters are specified through the configuration file, whereas the uncertainty related to the state comes from the used Observer. This predictor samples from each of these distributions a specified number of times, and for each set of samples, simulates ahead using the specified `PrognosticsModel` to determine when and if any of the specified events occur and to compute the predicted outputs.

For an example of an empty predictor, see Section A.5.

6. TESTING

In this section, we describe the methodology and processes used to test GSAP and its functions.

6.1. Framework

The first consideration for testing is the selection of a test framework. Some languages make this selection straightforward, either because tools are included in the language's core library (e.g. Python), or because there is a strong consensus within the language's community (e.g. Java/JUnit). No such straightforward path is evident for projects written in C++. Ultimately we chose to write a simple testing framework that meets our needs in a single short (less than 1000 lines) header file.

There were several important considerations that led to the decision to develop our testing framework, such as the ability to control the features and structure of the test framework. Many unit testing products make extensive use of macros to "simplify" test suites. One of the principle goals of GSAP is to write the entire project in standards-compliant cross-platform C++. By implementing a test framework, we both ensure that the framework itself adheres to our code quality goals, but we can format the tests themselves to use ordinary C++ without obfuscating macros.

6.2. Unit Testing

Various unit tests were written alongside the classes that make up GSAP from the beginning. Once a useful testing framework was added to the project, existing tests were adapted to use the test framework. Each class is accompanied by a set of unit tests, where each unit test function thoroughly tests a single piece of functionality in the class.

Functionality is tested in three basic ways. First, simple initialization subroutines such as constructors are tested with several sample inputs, and the public properties of the class are compared to expected values to ensure that classes are initialized correctly. Because initialization code is usually very simple, no great effort is made in these tests to ensure exhaustive coverage of all possible inputs. Second, all functions are

tested using inputs for which the outputs are hand-calculated or otherwise known. This provides a minimum level of confidence that each function operates correctly when given correct inputs, or fails predictably when given inputs that are known to be invalid. Finally, for functionality that has been optimized for performance, a large number of random inputs are tested to ensure that the optimized code produces identical results to the slower, well-tested algorithm it replaces.

6.3. Code Reviews

Unit tests are excellent for verifying that code does what is expected. However unit tests do not catch algorithmic errors, where code performs as expected, but the programmer's expectations are incorrect. To catch this type of error, GSAP is subjected to periodic code reviews as part of each development cycle. During code reviews, developers outside of the GSAP project are given a section of the code and asked to comment on any part of the code that is either unclear or possibly incorrect. These comments are then discussed by the core GSAP developers along with the reviewers, and changes are developed to address the issues raised by reviewers. This process both corrects errors not found by unit tests and also identifies confusing sections of code that can be rewritten to improve clarity.

6.4. Verification Testing

Each iteration of the development of GSAP began with the development, review, and stakeholder approval of requirements, and concluded with verification tests. In these tests, GSAP is stressed in various ways to verify that each requirement is met. The results of these tests were documented and confirmed before continuing to the next design-build-test iteration.

6.5. Validation Demos

Each iteration would also conclude with a validation demo. Here the functionality of GSAP was demonstrated for stakeholder representatives who provided feedback. This is done to confirm that the product completed meets stakeholder expectations. Any comments or concerns of stakeholders were noted to be addressed in the next iteration.

6.6. Continuous Integration

The extensive work described above made it straightforward to add a continuous integration component to the GSAP development environment. This was accomplished using Atlassian's Bamboo tool to provide continuous test feedback to the project's developers. A simple modification to the test suite to produce output in a JUnit compatible format made it possible to run tests and see results within Bamboo. These test results were also easy to integrate into Atlassian's Jira issue tracker, which the development team was already using for issue tracking.

7. APPLICATION: BATTERY PROGNOSTICS

To demonstrate the application of GSAP in a real-world example, we consider end-of-discharge (EOD) prognostics for batteries. The prognoser in this scenario is an instance of the `ModelBasedPrognoser` class, configured to use (*i*) a `PrognosticsModel` implementing the electrochemistry-based battery model described in (Daigle & Kulkarni, 2013), (*ii*) the unscented Kalman filter for state estimation, and (*iii*) the Monte Carlo-based predictor for EOD prediction. The important configuration parameters for the prognoser in this scenario can be seen below.

```
# General configuration
type:modelBasedPrognoser
name:battery1
id:1234abcd
inTags: voltage:voltage, power:power,
      temperature:temperature

# Prognoser configuration
model:Battery
inputs:power
outputs:temperature,voltage

# Model Configuration
Model.event:EOD
Model.predictedOutputs:SOC
Model.processNoise: 1e-5, 1e-5, 1e-5, ...

# Observer configuration
observer:UKF
Observer.Q: 1e-10, 0, 0, 0, ...
Observer.R: 1e-2, 0, 0, 1e-2

# Predictor configuration
predictor:MC
Predictor.numSamples: 100
Predictor.horizon: 10000
Predictor.inputUncertainty: 8, 0.1
```

The model-based prognoser is specified, along with the named data inputs from the system. The prognoser is next specified and relates model inputs and outputs to the incoming data. The model configuration specified the event to be predicted (EOD) and predicted variables (SOC), along with process noise variances (8 total, one for each state). The observer is set as the unscented Kalman filter, and the process and sensor noise covariance matrices are specified (not all values are shown). The predictor is set as the Monte Carlo predictor, with the number of samples and prognostic horizon specified, along with the uncertainty (mean and standard deviation) associated with the future input (power) to the battery.

The prognostics application created for this application included a model-based prognoser, and three communicators: playback, recorder, and LVC. These communicators will be described with their relevant configuration in the following sections. Two configurations were used for this setup: a test

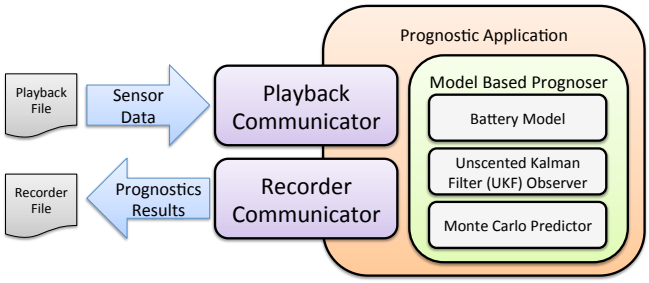

Figure 12. Prognostics Application - Test Configuration

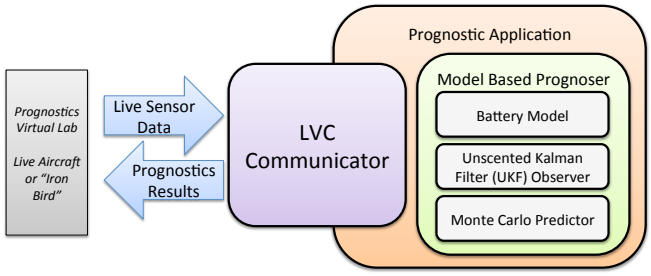

Figure 13. Prognostics Application - Deployed Configuration

configuration and a deployed configuration. The configuration of the prognoser was identical for each of these. These two configurations are described in detail in the following sections.

7.1. Test Configuration

A test configuration of this prognostics application was created to test the application separate from the physical system on which it will be deployed. It is important to have an environment for testing algorithms before they are deployed. The setup of this configuration can be seen in Figure 12. The configuration in the primary configuration file is included below.

```
# Test Main Configuration File
prognosers: Battery1.cfg
communicators: BattPlayback.cfg, Recorder.cfg
```

In this configuration, the `PlaybackCommunicator` was used to read sensor data from a file at a configurable rate, and the `RecorderCommunicator` recorded the results. The `PlaybackCommunicator` was configured (in BattPlayback.cfg) to playback from a specific file. Recorded data from real battery discharges were used for these tests. This configuration was used to test the basic setup of a basic prognostics application using GSAP.

7.2. Deployed Configuration

The next step was to apply this to a NASA application. For this, we integrated the Prognostics Virtual Lab (Kulkarni et al., 2017b). The Prognostics Virtual Lab uses the Live Virtual Connected (LVC) Gateway (Murphy, Jovic, & Otto, 2015; NASA, 2015) to pass data between several live and geographically separated systems. In this case, it was used to pass data from the Edge Iron Bird (Kulkarni et al., 2017a), which applies an electrical load to the battery corresponding to recorded flight data. The setup of the deployed configuration can be seen in Figure 13, and the contents of the main configuration file is included below.

```
# Deployed Main Configuration File
prognosers: Battery1.cfg
communicators: LVC.cfg
```

The sensor data from the Edge Iron Bird is communicated

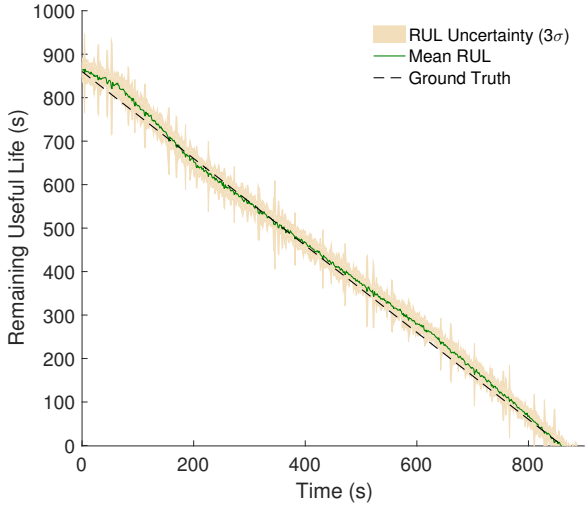

Figure 14. Remaining Useful Life Estimations

live using the Prognostics Virtual Laboratory. A LVC Communicator is used to subscribe to the live sensor information and communicate the prognostics results. The Edge Iron Bird will be replaced with a live Edge540 small unmanned aerial system (sUAS) in future tests.

7.3. Results

This section includes the results of running the test configuration with data from a real flight on an Edge 540T small unmanned aerial system (sUAS). The data was from a short flight where the aircraft conducted a climb, flew to a series of waypoints, and then landed. The end of life voltage was configured so EOL would be reached during the flight, providing a ground truth. The remaining useful life (RUL) estimation for this flight can be seen in Figure 14.

Here the mean estimated RUL is signified by a solid line, the three standard deviation uncertainty bound by the tan section, and the ground truth by the dashed line. Note that the ground truth stayed within the 3σ uncertainty bounds. The spikes in RUL uncertainty bounds would likely smooth with additional tuning. This example demonstrates GSAP being used for sUAS operations.

8. CONCLUSION

This paper introduces the Generic Software Architecture for Prognostics, a generic, object-oriented, open-source software framework and support library for deploying prognostics technologies. It is the only open framework that the authors are aware of that is specifically designed to target prognostics applications. This architecture was designed to make prognostics more accessible by reducing the effort and investment required to develop, test, and deploy prognostics.

GSAP was thoroughly tested and has been applied to multiple ongoing NASA projects. The requirements, design, testing and use of GSAP have been described in detail in this paper. A C++ implementation of GSAP is currently available open-source on GitHub (Teubert, Daigle, Sankararaman, Watkins, & Goebel, 2016).

Future work for the GSAP framework includes the further generalization and testing of GSAP under other paradigms such as an As-A-Service implementation on cloud resources, implementation on embedded processors, and implementation of GPU-accelerated algorithms. Additionally, GSAP will further be developed to include additional support tools to aid with the development of prognostics applications. Finally, the team intends to convert existing prognostics models into the GSAP framework so it can be used with future prognostics applications.

ACKNOWLEDGMENT

The team acknowledges the significant contributions of interns Marcus Postell (Bethune-Cookman University) and Micah Ricks (Alabama Agricultural and Mechanical University). They also acknowledge the following NASA projects for their support of this effort: System-wide Safety Assurance Technologies (SSAT) Project, Aviation Safety Program (AvSP), Aeronautics Research Mission Directorate (ARMD); Advanced Ground Systems Maintenance (AGSM) Project, Ground Systems Development and Operations (GSDO) Program, Human Exploration and Operations Mission Directorate (HEOMD); Autonomous Cryogenic Loading Operations (ACLO) Project, Advanced Exploration Systems (AES) Program, HEOMD.

REFERENCES

Arulampalam, M. S., Maskell, S., Gordon, N., & Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, *50*(2), 174–188.

Chen, C., & Pecht, M. (2012). Prognostics of lithium-ion batteries using model-based and data-driven methods. In *Ieee conference on prognostics and system health management.*

CMMI Product Team. (2010). *CMMI for development, version 1.3*. Retrieved from `http://www.sei.cmu.edu/reports/10tr033.pdf`

Daigle, M. (2016a, October). *Prognostics algorithm library.* Retrieved from `https://github.com/nasa/PrognosticsAlgorithmLibrary`

Daigle, M. (2016b, October). *Prognostics model library.* Retrieved from `https://github.com/nasa/PrognosticsModelLibrary`

Daigle, M., & Goebel, K. (2013, May). Model-based prognostics with concurrent damage progression processes. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *43*(4), 535-546.

Daigle, M., & Kulkarni, C. (2013, October). Electrochemistry-based battery modeling for prognostics. In *Annual conference of the prognostics and health management society 2013* (p. 249-261).

Daigle, M., Saha, B., & Goebel, K. (2012, March). A comparison of filter-based approaches for model-based prognostics. In *2012 ieee aerospace conference.*

Daigle, M., & Sankararaman, S. (2013, October). Advanced methods for determining prediction uncertainty in model-based prognostics with application to planetary rovers. In (p. 262-274).

Daigle, M., & Sankararaman, S. (2016, December). Predicting remaining driving time and distance of a planetary rover under uncertainty. *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering*, *2*(4).

Deal, Ryan and Kessler, Seth. (2011). *Architecture* (S. Johnson et al., Eds.). Wiley.

Doucet, A., Godsill, S., & Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, *10*, 197–208.

General Electric. (2017). *Predix platform.* Retrieved from `http://predix.io`

Goebel, K., Daigle, M. J., Saxena, A., Sankararaman, S., Roychoudhury, I., & Celaya, J. (2017). *Prognostics: The Science of Making Predictions*. CreateSpace Independent Publishing Platform.

IEEE. (2015, June). *Standard for artificial intelligence exchange and service tie to all test environments* (standard No. 1232-2010).

IEEE RS/SC Reliability. (2016, June). *Draft standard framework for prognostics and health management of electronic systems* (standard No. P 1856).

ISO/TC 108/SC 5 Condition monitoring and diagnostics of machine systems. (2015, September). *Condition monitoring and diagnostics of machines prognostics part 1* (standard No. 13381-1:2015).

Julier, S. J., & Uhlmann, J. K. (2004, March). Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, *92*(3), 401–422.

Keller, K., Wiegand, D., Swearingen, K., Reisig, C., Black, S., Gillis, A., & Vandernoot, M. (2001). An architecture to implement integrated vehicle health management systems. In *Autotestcon proceedings, 2001. ieee systems readiness technology conference* (pp. 2–15).

Kulkarni, C. S., Teubert, C., Gorospe, G., Quach, C. C., Hogge, E., & Darafsheh, K. (2017a). Application of prognostics methodology to virtual laboratory for future aviation and airspace research. In *Aiaa aviation conference.*

Kulkarni, C. S., Teubert, C., Gorospe, G., Quach, C. C., Hogge, E., & Darafsheh, K. (2017b). A virtual laboratory for aviation and airspace prognostics research. In *Aiaa modeling and simulation technologies conference* (p. 1767).

Leao, B. P., Yoneyama, T., Rocha, G. C., & Fitzgibbon, K. T. (2008). Prognostics performance metrics and their relation to requirements, design, verification and cost-benefit. In *International conference on prognostics and health management.*

MIMOSA. (2006). *Open system architecture for condition-based maintenance* (standard No. 1232-2010).

Murphy, J., Jovic, S., & Otto, N. (2015). Message latency characterization of a distributed live, virtual, constructive simulation environment. In *Aiaa infotech at aerospace conference.*

NASA. (2015). *Live virtual constructive distributed environment (lvc) lvc gateway, gateway toolbox.*

NASA. (2017). *Core flight software.* Retrieved from `http://cfs.gsfc.nasa.gov`

National Aeronautics and Space Administration. (2014). *Nasa software engineering requirements*. Retrieved from `http://specs4.ihserc.com/Document/Document/ViewDoc?docid=JUYNJFAAAAAAAAAA`

National Aeronautics and Space Administration. (2017). *Nasa software engineering handbook*. Retrieved from `https://swehb.nasa.gov/display/7150/Book+A.+Introduction`

Orchard, M., & Vachtsevanos, G. (2009, June). A particle filtering approach for on-line fault diagnosis and failure prognosis. *Transactions of the Institute of Measurement and Control*, *31*(3-4), 221-246.

Robot operating system. (2017). Retrieved from `http://www.ros.org`

RTCA. (1992). *Software considerations in airborne systems and equipment certification* (standard No. DO-178B).

SAE Aerospace. (2017, February). *Guidelines for writing ivhm requirements for aerospace systems-draft* (ARP No. 6883).

Saha, B., & Goebel, K. (2009, September). Modeling Li-ion battery capacity depletion in a particle filtering framework. In *Proceedings of the annual conference of the prognostics and health management society 2009.*

Sankararaman, S. (2015). Significance, interpretation, and quantification of uncertainty in prognostics and remaining useful life prediction. *Mechanical Systems and Signal Processing*, *52*, 228–247.

Sankararaman, S., Daigle, M., & Goebel, K. (2014, June). Uncertainty quantification in remaining useful life prediction using first-order reliability methods. *IEEE Transactions on Reliability*, *63*(2), 603-619.

Saxena, A., Roychoudhury, I., Celaya, J., Saha, S., Saha, B., & Goebel, K. (2010). Requirements specification for prognostics performance-an overview. In *Aiaa infotech@ aerospace 2010.*

Saxena, A., Roychoudhury, I., Celaya, J. R., Saha, B., Saha, S., & Goebel, K. (2012, June). Requirements flowdown for prognostics and health management. *The American institute of Aeronautics and Astronautics (AIAA) Infotech 2012 Conference*.

Schwabacher, M. (2005). A survey of data-driven prognostics. In *Proceedings of the aiaa infotech@aerospace conference.*

Schwabacher, M., & Goebel, K. (2007). A survey of artificial intelligence for prognostics. In *Proceedings of aaai fall symposium.*

Srinivasan, S. (1999). Design patterns in object-oriented frameworks. *Computer*, *32*(2), 24–32.

Teubert, C., Daigle, M., Sankararaman, S., Watkins, J., & Goebel, K. (2016, December). *Generic software architecture for prognostics (gsap).* Retrieved from `https://github.com/nasa/GSAP`

Usynin, A., Hines, J. W., & Urmanov, A. (2007). Formulation of prognostics requirements. In *Aerospace conference, 2007 ieee.*

BIOGRAPHIES

Christopher Teubert is a software engineer and deputy group lead of the Diagnostics and Prognostics group at NASA Ames Research Center. He is also the principal investigator for the Generic Software Architecture for Prognostics (GSAP). Christopher received his B.S. in Aerospace Engineering from Iowa State University in 2012 and is currently working on his M.S. in Computer Science and Engineering at Santa Clara University. Chris worked as a research engineer with Stinger Ghafarrian Technologies (SGT) at NASA Ames Research Center from 2012-2016. Since 2016, Chris has been a computer engineer and deputy group lead with the Diagnostics and Prognostics group at NASA since 2016, where he designs, develops, and tests software for performing prognostics for aircraft and spacecraft.

Matthew Daigle received the B.S. degree in computer science and computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, in 2004, and the M.S. and Ph.D. degrees in computer science from Vanderbilt University, Nashville, TN, in 2006 and 2008, respectively. From September 2004 to May 2008, he was a Graduate Research Assistant with the Institute for Software Integrated Systems and Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN. From June 2008 to December 2011, he was an Associate Scientist with the University of California, Santa Cruz, at NASA Ames Research Center. Since January 2012, he has been with NASA Ames Research Center as a Research Computer Scientist, and has been the lead of the Diagnostics & Prognostics Group since 2016. He has published over 100 peer-reviewed papers in the area of systems health management. His current research interests include physics-based modeling, model-based diagnosis and prognosis, simulation, and autonomy.

Shankar Sankararaman received his Bachelors degree in Civil Engineering from the Indian Institute of Technology, Madras in India in 2007 and later, obtained his Ph.D. in Civil Engineering from Vanderbilt University, Nashville, Tennessee, U.S.A. in 2012. His research focuses on the various aspects of uncertainty quantification, integration, and management in different types of aerospace, mechanical, and civil engineering systems. His research interests include probabilistic methods, risk and reliability analysis, Bayesian networks, system health monitoring, diagnosis and prognosis, decision-making under uncertainty, and multidisciplinary analysis. He is a member of the Non-Deterministic Approaches (NDA) technical committee at the American Institute of Aeronautics, the Probabilistic Methods Technical Committee (PMC) at the American Society of Civil Engineers (ASCE), and the Prognostics and Health Management (PHM) Society. Currently, Shankar is a researcher at NASA Ames Research Center, Moffett Field, CA, where he develops algorithms for system health monitoring, prognostics, decision-making, and uncertainty management.

Kai Goebel is the Area Lead for Discovery and Systems Health at NASA Ames where he was director of the Prognostics Center of Excellence during the time this research was conducted. After receiving the Ph.D. from the University of California at Berkeley in 1996, Dr. Goebel worked at General Electric's Corporate Research Center in Niskayuna, NY from 1997 to 2006 as a senior research scientist before joining NASA. He has carried out applied research in the areas of artificial intelligence, soft computing, and information fusion and his interest lies in advancing these techniques for real time monitoring, diagnostics, and prognostics. He holds 18 patents and has published more than 300 papers in the area of systems health management.

Jason Watkins is an undergraduate student studying Computer Science and Engineering at University of California, Irvine. Jason has completed several undergraduate internships working for NASA and Broadcom. These internships include 3 internships working as a member of the Diagnostics and Prognostics Group at NASA Ames Research Center. As part of those internships, Jason has worked on several NASA software projects, including the open source projects X-Plane Connect and Generic Software Architecture for Prognostics (GSAP).

A. EMPTY INTERFACES

A.1. Prognoser

```
/**  Empty Prognoser - Header (v0.1.0)
 *   The purpose of this class is to serve as a
 *   template for creating future prognosers
 *
 *   Requires: Prognostic Configuration File and
 *     Prognoster Configuration Files
 *
 *   Contact: Chris Teubert
 *      (christopher.a.teubert@nasa.gov)
 *   Created: November 11, 2015
 *
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 */

#ifndef PCOE_EMPTYPROGNOSER_H
#define PCOE_EMPTYPROGNOSER_H

#include "CommonPrognoser.h"

namespace PCOE {
 class EmptyPrognoser : public CommonPrognoser {
  /** @brief Example Prognoser Constructor
   *  @param config  Map of config parameters from
   *    the prognoser config file
   */
  EmptyPrognoser(GSAPConfigMap & config);

  /** @brief Prognostic Monitor Step:
   *    perform model updates. This is done every
   *    step where there is enough data. This is a
   *    required method in any component prognoser
   */
  void step();

  //*-------------------------------------*
  //| Optional Methods- Uncomment to use |
  //*-------------------------------------*

  /** @brief check the validity of any input/sensor
   *    data. This could be bound checks or a
   *    complicated analysis. By default this is
   *    not done- making this step optional in the
   *    component prognoser implementation
   */
   // void checkInputValidity() {}

   /** @brief check if there is enough new data to
    *    perform prognosis. Check if the data exists
    *    and is new enough to be used for prognosis.
    *    If false is returned prognostics steps will
    *    be skipped. By default this returns true-
    *    making this step optional in the component
    *    prognoser implementation.
    *  @return if there is enough data
    */
    // bool isEnoughData() {return true;}

    /** @brief check the validity of any prognostics
     *    results. This could be bound checks or a
     *    complicated analysis. By default this a
     *    simple bounds test on timeToEvent - making
     *    this step optional in the component
     *    prognoser implementation. Default
     *    implementation is in CommonPrognoser
     */
     // void checkResultValidity();
  };
}
#endif // PCOE_EMPTYPROGNOSER_H
```

```
/**  Empty Prognoser - Body (v0.1.0)
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 */

 #include "EmptyPrognoser.h"

namespace PCOE {
  EmptyPrognoser::EmptyPrognoser(GSAPConfigMap & c) :
    CommonPrognoser(c) {
    // DEFINE EVENTS FOR THIS SPECIFIC PROGNOSER
    // Ex: info.events.push_back(param("EOL", "s"));

    // Handle Configuration
    log.WriteLine(LOG_DEBUG, moduleName,
      "Configuring");
    // std::string a = c.at("ExampleParam");
  }

  void EmptyPrognoser::step() {
    log.WriteLine(LOG_TRACE, moduleName,
      "Running␣Monitor␣Step");

    // Update States
    // Ex: currentProgData.state["S1"].set(1.1);
    // Ex: currentProgData.state["S2"].set(0.9);

    // Update safety Metrics
    // Ex: currentProgData.
    //        safetyMetric[MEAN].set(1.2);

    log.WriteLine(LOG_TRACE, moduleName,
      "Running␣Prediction␣Step");
    // Update Time To Events
    // Ex: currentProgData.
    //      timeToEvent[MEAN].set(1.5);

    // Update Future Safety Metrics
  }

  // *-------------------------------------*
  // | Optional Methods- Uncomment to use |
  // *-------------------------------------*
  // void EmptyPrognoser::checkInputValidity() { }
  //
  // bool EmptyPrognoser::isEnoughData() {
  //    return true;
  // }
  //
  // void EmptyPrognoser::checkResultValidity() { }
```

A.2. Communicator

```
/** Empty Communicator - Header
 *  @class     EmptyCommunicator EmptyCommunicator.h
 *  @brief     Communicator Template
 *
 *  Contact: Chris Teubert
 *   (Christopher.a.teubert@nasa.gov)
 *  Created: March 25, 2016
 *
 *  Copyright (c) 2013-2017 United States
 *  Government as represented by the Administrator
 *  of the National Aeronautics and Space
 *  Administration All Rights Reserved
 */

#ifndef PCOE_EMPTYCOMMUNICATOR_H
#define PCOE_EMPTYCOMMUNICATOR_H

#include "CommonCommunicator.h"  ///< Parent Class

namespace PCOE {
  class EmptyCommunicator :
   public CommonCommunicator {
   public:
    /** @brief Constructor for EmptyCommunicator -
     *     Called by the CommunicatorFactory
     *  @param config  Reference to configuration
     *                 map for the communicator
     *  @see   CommunicatorFactory
     **/
    EmptyCommunicator(const ConfigMap & config);

    /** @brief Poll function- see if there is
     **    data to read from this communicator
     **/
    inline void poll() override;

    /** @brief Publisher callback function- to
     *   consume data from the prognostic framework
     *  @param data     Reference to DataStore
     *   containing all the input data
     *  @param progData Output from each prognoser
     **/
    void write(AllData data) override;

    /** @brief  Subscriber callback function- to
     *   introduce data into the prognostic framework
     *  @param  data  Reference to DataStore
     *   containing all the data
     *  @return Updated Datastore with new data
     **/
    DataStore read() override;

    /// @brief Optional destructor
    //~EmptyCommunicator();
  };
}
#endif // PCOE_EMPTYCOMMUNICATOR_H
```

```
/**  Empty Communicator - Body (v0.1.0)
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 */

#include <string>
#include "EmptyCommunicator.h"

namespace PCOE {
  const std::string MOD_NAME = "EMPTYCOMM";

  EmptyCommunicator::EmptyCommunicator
    (const ConfigMap & configMap) {
    log.WriteLine(LOG_DEBUG, MOD_NAME, "Configuring");
    ///-------------------------------------
    /// HERE IS WHERE YOU CONFIGURE THE Communicator.
    /// Read the configuration map
    ///  Example: std::string SomeParam =
    ///    configMap.at("SomeParam")[0];
    ///  Example2: std::vector<std::string>
    ///    paramList = configMap.at("otherParam");
    ///
    /// ADD COMMUNICATOR CONFIGURATION CODE BELOW:
    ///-------------------------------------
  }

  inline void EmptyCommunicator::poll() {
    // setRead(); //Call this if there is data to be
    // read. Not calling setRead() in this function
    // will mean that the read() function will not
    // be called
  }

  void EmptyCommunicator::write(AllData data) {
    DataStore & ds = data.doubleDatastore;
    DataStoreString & dsString = data.stringDataStore;
    ProgDataMap & pData = data.progData;
    ///-------------------------------------
    /// HERE IS WHERE YOU SEND DATA
    ///
    /// The DataStore ds contains the latest received
    /// sensor data. Access it by key, for example:
    ///   double value = ds["SomeKey"];
    ///   ms_rep t = ds["SomeKey"].getTime();
    /// The DataStoreString dsString contains the
    /// latest received sensor strings. Access it by
    /// key, for example:
    ///   std::string value = ds["SomeKey"];
    ///   ms_rep t = ds["SomeKey"].getTime();
    ///
    /// The ProgDataMap pData contains the results of
    /// all prognosers. Access it by prognoser name,
    /// for example:
    ///   ProgData & pdBatt = pData["Battery1"];
    /// ADD COMMUNICATOR PUBLISHER CODE BELOW:
    ///-------------------------------------
  }

  DataStore EmptyCommunicator::read() {
    DataStore ds;
    ///-------------------------------------
    /// HERE IS WHERE YOU RECEIVE DATA
    ///  Receive data and fill in the DataStore 'ds'
    ///  Example: ds["someParam"] = 1.0;
    ///
    /// ADD COMMUNICATOR SUBSCRIBER CODE BELOW:
    ///-------------------------------------

    ///-------------------------------------
    return ds;
  }
}
```

A.3. Model

```
/** EmptyModel - Header
 *   Contact: Matthew Daigle
 *     (matthew.j.daigle@nasa.gov)
 *   Created: January 10, 2017
 *
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 **/

#ifndef EmptyModel_H
#define EmptyModel_H

#include <cmath>
#include <vector>
#include "Model.h"
#include "ConfigMap.h"
#include "ModelFactory.h"

class EmptyModel final :
 public PCOE::Model {
 public:
  // Constructor
  EmptyModel();

  // Constructor based on configMap
  EmptyModel(const PCOE::ConfigMap & paramMap);

  /** @brief Execute state equation. This version of
   *  the function uses a given sampling time.
   *   @param t  Time
   *   @param x  Current state vector. This gets
   *       updated to the state at the new time.
   *   @param u  Input vector
   *   @param n  Process noise vector
   *   @param dt Sampling time
   **/
  void stateEqn(const double t,
    std::vector<double> & x,
    const std::vector<double> & u,
    const std::vector<double> & n, const double dt);

  /** @brief Execute output equation
   *   @param t Time
   *   @param x State vector
   *   @param u Input vector
   *   @param n Sensor noise vector
   *   @param z Output vector. This gets updated to
   *      the new output at the given time.
   **/
  void outputEqn(const double t,
    const std::vector<double> & x,
    const std::vector<double> & u,
    const std::vector<double> & n,
    std::vector<double> & z);

  /** @brief Initialize state vector given initial
   *    inputs and outputs.
   *   @param x Current state vector. Update this.
   *   @param u Input vector
   *   @param z Output vector
   **/
  void initialize(std::vector<double> & x,
    const std::vector<double> & u,
    const std::vector<double> & z);
};
#endif
```

```
/**  EmptyModel - Body
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 **/

#include "EmptyModel.h"
#include "ConfigMap.h"

using namespace PCOE;

EmptyModel::EmptyModel() {
    // Default constructor
}

// Constructor based on configMap
EmptyModel::EmptyModel(const ConfigMap & configMap)
  : EmptyModel::EmptyModel() {
  // Setup model based on configuration parameters
}

// EmptyModel State Equation
void EmptyModel::stateEqn(const double,
   std::vector<double> & x,
   const std::vector<double> & u,
   const std::vector<double> & n, const double dt) {
  // Extract states
  // double a = x[0];
  // double b = x[1]; ...

  // Extract inputs
  // double c = u[0]; ...

  // State equations
  // double adot = a + b; ...

  // Update state
  // x[0] = a + adot*dt; ...

  // Add process noise
  // x[0] += dt*n[0]; ...
}

// EmptyModel Output Equation
void EmptyModel::outputEqn(const double,
    const std::vector<double> & x,
    const std::vector<double> &,
    const std::vector<double> & n,
    std::vector<double> & z) {
  // Extract states
  // double a = x[0]; ...

  // Extract inputs
  // double c = u[0]; ....

  // Output equations
  // double d = a + b + c; ...

  // Set outputs
  // z[0] = d; ...

  // Add noise
  // z[0] += n[0]; ...
}

// Initialize state, given initial inputs and outputs
void EmptyModel::initialize(std::vector<double> & x,
  const std::vector<double> & u,
  const std::vector<double> & z) {
  // Determine x from u and z (model-dependent)
  //  or as fixed values
  // x[0] = u[0] + z[0]; ...
}
```

A.4. Observer

```
/**  EmptyObserver - Header (v0.1.0)
 *
 *   Contact: Matthew Daigle
 *      (matthew.j.daigle@nasa.gov)
 *   Created: January 10, 2017
 *
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 */

#ifndef PCOE_EmptyObserver_H
#define PCOE_EmptyObserver_H

#include <vector>
#include <cmath>
#include "Observer.h"

namespace PCOE {
  class EmptyObserver final : public Observer {
   public:
    /** @brief Set model pointer
     *   @param model given model pointer
     **/
    void setModel(Model *model);

    /** @brief Initialize UKF
     *  @param t0 Initial time
     *  @param x0 Initial state vector
     *  @param u0 Initial input vector
     **/
    void initialize(const double t0,
        const std::vector<double> & x0,
        const std::vector<double> & u0);

    /** @brief Estimation step. Updates xEstimated,
     *     zEsitmated, P, and sigmaX.
     *  @param newT Time value at new step
     *  @param u Input vector at current time
     *  @param z Output vector at current time
     **/
    void step(const double newT,
        const std::vector<double> & u,
        const std::vector<double> & z);

    // Accessors
    const std::vector<double> & getStateMean() const;
    const std::vector<double> & getOutputMean() const;
    std::vector<UData> getStateEstimate() const;
  };
}
#endif // PCOE_EmptyObserver_H
```

```
/**  EmptyObserver - Body (v0.1.0)
 *
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 */

#include "EmptyObserver.h"
```

A.5. Predictor

```
/**  EmptyPredictor - Header (v0.1.0)
 *
 *   Contact: Matthew Daigle
 *    (matthew.j.daigle@nasa.gov)
 *   Created: January 10, 2017
 *
 *   Copyright (c) 2013-2017 United States
 *   Government as represented by the Administrator
 *   of the National Aeronautics and Space
 *   Administration All Rights Reserved
 */
#ifndef PCOE_EmptyPredictor_H
#define PCOE_EmptyPredictor_H

#include <vector>
#include <string>
#include "Model.h"
#include "Predictor.h"
#include "GSAPConfigMap.h"

namespace PCOE {
  class EmptyPredictor final : public Predictor {

  public:
    /** @brief Constructor for a EmptyPredictor
     *     based on a configMap
     *  @param configMap Configuration map
     *     specifying predictor parameters
     **/
    explicit EmptyPredictor
      (GSAPConfigMap & configMap);

    /** @brief Set model pointer
    *   @param model given model pointer
    **/
    void setModel(PrognosticsModel * model);

    /** @brief Predict function for a Predictor
     *  @param tP    Time of prediction
     *  @param state state of system at time of
     *    prediction
     *  @param data  ProgData object, in which
     *    prediction results are stored
     **/
    void predict(const double tP,
      const std::vector<UData> & state,
      ProgData & data);
  };
}
#endif  // PCOE_EmptyPredictor_H
```

```
/**  EmptyPredictor - Body
*   Copyright (c) 2013-2017 United States
*   Government as represented by the Administrator
*   of the National Aeronautics and Space
*   Administration All Rights Reserved
*/

#include <random>
#include <string>
#include <vector>
#include "Exceptions.h"
#include "EmptyPredictor.h"
#include "Matrix.h"

namespace PCOE {
  // ConfigMap-based Constructor
  EmptyPredictor::EmptyPredictor
    (GSAPConfigMap & configMap) : Predictor() {
    // Setup based on configuration parameters ...

    log.WriteLine(LOG_INFO, MODULE_NAME,
      "EmptyPredictor␣created");
  }

  // Set model
  void EmptyPredictor::setModel
    (PrognosticsModel * model) {
    pModel = model;

    // Perform some checks on the model ...
  }

  // Predict function
  void EmptyPredictor::predict(const double tP,
    const std::vector<UData> & state,
    ProgData & data) {
    // Check that model has been set
    if (pModel == NULL) {
      log.WriteLine(LOG_ERROR, MODULE_NAME,
        "EmptyPredictor␣does␣not␣have␣a␣model!");
      throw ConfigurationError
        ("EmptyPredictor␣does␣not␣have␣a␣model!");
    }

    // Run prediction, and fill in the prog data ...
  }
}
```

A.6. Prognostics Application Driver

```
#include "ProgManager.h"
#include "ConfigMap.h"

// Factories- these register custom components
// (Prognosers, etc.). They are not necessary if
// you are only using the included components.
// If you are adding your own components, you
// must register them with the appropriate factory.
#include "PrognoserFactory.h"
#include "CommunicatorFactory.h"
#include "ModelFactory.h"
#include "PrognosticsModelFactory.h"
#include "ObserverFactory.h"
#include "PredictorFactory.h"

using namespace PCOE;

int main() {
  // Specify config file directories (optional)
  ConfigMap::addSearchPath("../example/cfg/");

  // Specify Prognosers - If using custom
  // prognosers, otherwise this is not necessary
  PrognoserFactory & prognoserFactory =
    PrognoserFactory::instance();
  prognoserFactory.Register("prognoser␣name",
    PrognoserFactory::Create<PrognoserName>);

  // Specify Communicators - If using custom
  // communicators, otherwise this is not
  // necessary
  CommunicatorFactory & commFactory =
    CommunicatorFactory::instance();
  commFactory.Register("communicator␣name",
    CommunicatorFactory::Create<CommName>);

  // Register model - If using custom models,
  // otherwise this is not necessary
  ModelFactory & pModelFactory =
    ModelFactory::instance();
  PrognosticsModelFactory & pProgModelFactory =
    PrognosticsModelFactory::instance();
  pModelFactory.Register("model␣name",
    ModelFactory::Create<ModelName>);
  pProgModelFactory.Register("model␣name",
    PrognosticsModelFactory::Create<ModelName>);

  // Register observers - If using custom
  // observers, otherwise this is not necessary
  ObserverFactory & pObserverFactory =
    ObserverFactory::instance();
  pObserverFactory.Register("observer␣name",
    ObserverFactory::Create<ObserverName>);

  // Register predictors
  PredictorFactory & pPredictorFactory =
    PredictorFactory::instance();
  pPredictorFactory.Register("predictor␣name",
    PredictorFactory::Create<PredictorName>);

  ProgManager PM =
    ProgManager("topLevelConfigFileName.cfg");
  PM.run(); // Starts the GSAP Application
  return 0;
}
```

B. GSAP REQUIREMENTS

Table 6. GSAP Requirements

	Name	Text
GSAP-1	Cross-Platform Compatability	GSAP shall fulfill all the below requirements on Mac, Windows, Linux OS versions documented in the User Guide
GSAP-2	Coding Standards	GSAP shall meet the project coding standards as defined in the Project Software Standards document
GSAP-3	Communicator Usability	The required code for a Communicator to interface to GSAP shall be less than 250 lines of code
GSAP-4	Prognoser Usability	The required code for a Prognoser to interface to GSAP shall be less than 250 lines of code
GSAP-5	Modes of Operation	GSAP shall support the modes of operation for each prognoser as defined in the control flow diagram
GSAP-6	Clean Termination	GSAP shall release all resources (e.g., Threads and allocated memory) upon termination
GSAP-7	Multi-Threaded	GSAP shall start each prognoser and each communicator on its own independent thread
GSAP-8	Prognoser Interface	GSAP shall implement the interface defined in CommonPrognoser.h for connecting prognosers
GSAP-9	Multiple Prognosers	GSAP shall simultaneously connect one or more prognosers
GSAP-10	Periodic History Recording	GSAP shall store state time of event, and systemTrajectories as defined in ProgData (excluding predictions) for each Prognoser at a configurable interval
GSAP-11	Shutdown History Recording	GSAP shall store state information as defined in the ProgData Class upon receipt of a stop command
GSAP-12	Import History	At startup, GSAP shall communicate previously stored state information as defined in the ProgData Class to the Prognoser with the same component-id, if they exist
GSAP-13	Reset History	GSAP shall provide the ability to reset the prognostic history at startup
GSAP-14	Schedule Future Input	GSAP shall make any received schedule of future input from any of the following available to the component to which it corresponds: 1) communicators (“live data”), 2)from the Prognoser configuration file
GSAP-15	Output Check	GSAP shall perform a bound check on the output of each prognoser to confirm that the output is possible
GSAP-16	Communicator Interface	GSAP shall implement the interface found in CommonCommunicator.h for connecting to different communicators (examples: file, UDP, etc.)
GSAP-17	Multiple Communicators	The GSAP framework shall simultaneously connect one or more communicators
GSAP-18	Receiving Data	The GSAP framework shall make data received from a communicator available to the prognosers at request
GSAP-19	Sending Results	The GSAP framework shall make PHM results from the prognosers available to the communicators at request
GSAP-20	Sharing Prognoser Data	The GSAP framework shall provide an interface for any Prognoser to read the prognostics results of any other Prognoser
GSAP-21	Data Timestamp	GSAP shall timestamp the data when received
GSAP-22	Result Timestamp	GSAP shall timestamp the PHM results when received
GSAP-23	Prognoser Specification	At startup, the GSAP framework shall create and initialize prognosers specified by the user(s)
GSAP-24	Communicator Specification	At startup, the GSAP framework shall create and initialize communicators specified by the user(s)
GSAP-25	Prognoser Configuration	GSAP shall support unique configuration parameters for each Prognoser
GSAP-26	Prognoser Configuration Communication	At startup, GSAP shall send the Prognoser-specific configuration parameters to the Prognoser to which it applies
GSAP-27	Data Storage	ProgData shall provide a method for storing the following: 1. Time of Event, 2. System Trajectories at multiple timesteps with corresponding timestamps, and 3. Internal Parameters
GSAP-28	Uncertain Data	ProgData shall provide at least one way of representing uncertainty for the following: 1. time of event and 2. System Trajectories
GSAP-29	Data Validity	ProgData shall provide a method for storing the validity of each data point for 1. Time of Event, 2. System Trajectories, and 3. Internal Parameters

	Name	Text
GSAP-30	Observers	The GSAP Support Library shall include the following observers for Prognosers: Unscented Kalman Filter, Particle Filter
GSAP-31	Predictors	The GSAP Support Library shall include the following predictors for Prognosers: Monte Carlo, Latin Hypercube
GSAP-32	Control	The GSAP Framework shall provide an interface for remote control and custom GUIs
GSAP-33	Non-Parametric Uncertainty Representation	The UData Type shall be capable of representing the following non-parametric uni-variate uncertainty types: Weighted Samples, Unweighted Samples, and Percentiles
GSAP-34	Parametric Uncertainty Representation	The UData Type shall be capable of representing the following parametric uni-variate uncertainty types: Gaussian, Log-Normal, Exponential
GSAP-35	Multi-variate Uncertainty Representation	The UData Type shall be capable of representing multi-variate Gaussian uncertainty type (with covar)
GSAP-36	Uncertainty Conversion	GSAP shall support the following conversions between UData Types: 1) non-parametric data type from any parametric data type, 2) weighted samples to unweighted samples, 3) unweighted samples to percentiles, 4) percentiles to unweighted samples, Where each conversion generates a new UData of the second type
GSAP-37	Playback	The Playback Communicator shall provide to the generic architecture software voltage and current measurements from a playback file at a configurable rate
GSAP-38	Playback Generality	The Playback Communicator shall associate the data in the playback file with a key specified in the playback file
GSAP-39	Recording	The Recorder Communicator shall record all data from the prognosers into a file at a configurable rate
GSAP-40	Recording Timestep	The Recorder Communicator shall with each data point include a timestamp at which the data was received
GSAP-41	Playback Records	The Recorder Communicator shall record the data in a format readable by the Playback Communicator

Table 7. GSAP Requirements Flowdown

1: Quality	2: Uncertainty Representation	3: Parallelized	4: Implement Common Elements	5: Common Interface	6: Flexible	7: Minimum Effort to Extend
GSAP-2 GSAP-6	GSAP-28 GSAP-33 GSAP-34 GSAP-35 GSAP-36	GSAP-7	GSAP-20 GSAP-21 GSAP-22 GSAP-27 GSAP-28 GSAP-30 GSAP-31 GSAP-37 GSAP-38 GSAP-39 GSAP-40 GSAP-41	GSAP-8 GSAP-9 GSAP-16 GSAP-17 GSAP-18 GSAP-19 GSAP-23 GSAP-24 GSAP-25 GSAP-26 GSAP-32	GSAP-1 GSAP-5 GSAP-10 GSAP-11 GSAP-12 GSAP-13 GSAP-14 GSAP-15	GSAP-3 GSAP-4

C. CLASS DIAGRAM

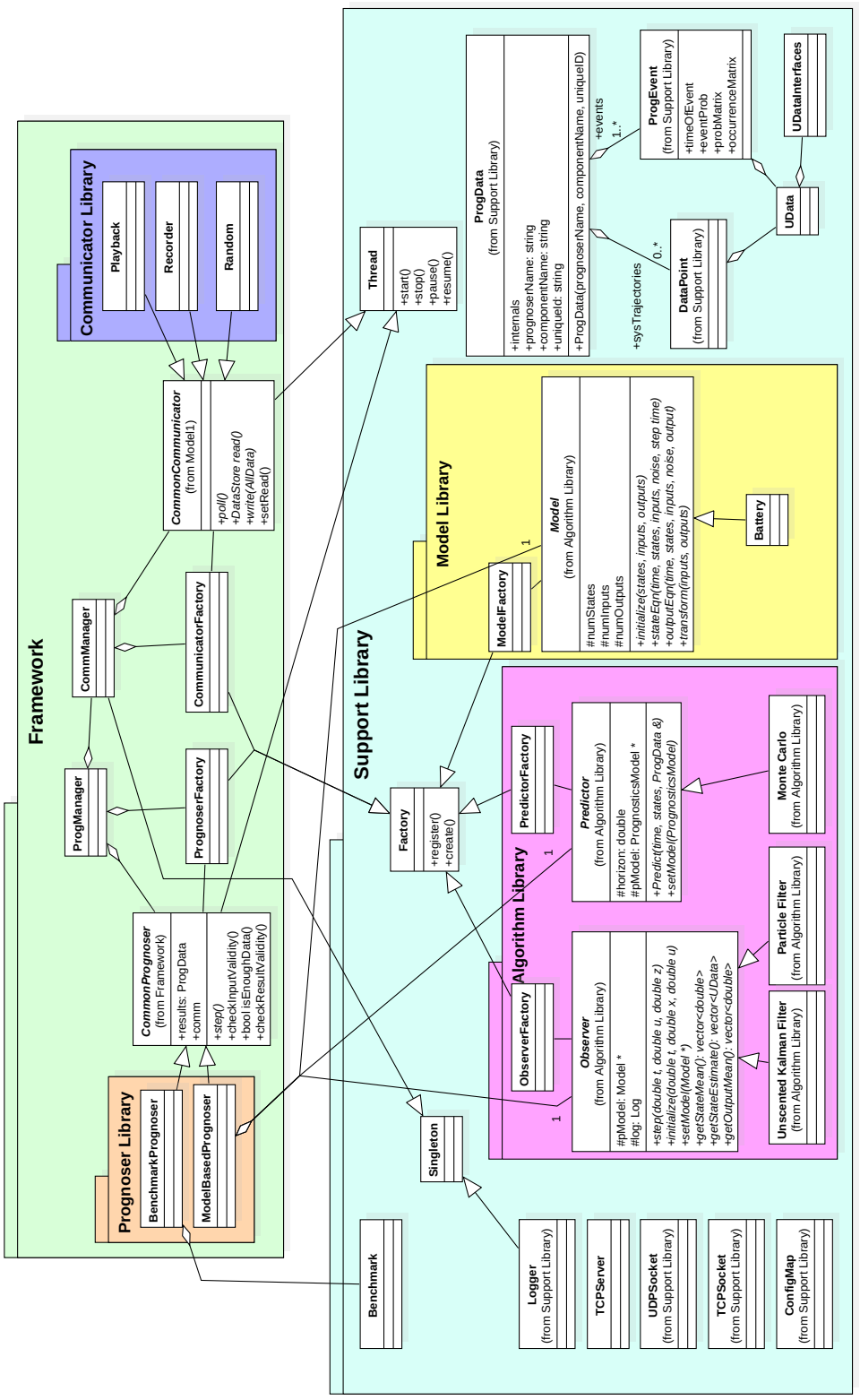

Figure 15. Class Diagram