Journal of Aerospace Information Systems, Volume 11, Number 2 (2014), Pages 82-97.
© AIAA 2014

A Software Framework for Prototyping Embedded IVHM Applications

T. Sreenuch[1], A. Tsourdos[2] and I. K. Jennions[1]

[1]Integrated Vehicle Health Management Centre, Cranfield University, Conway House, University Way, Cranfield, Bedford MK43 0FQ, UK

[2]Department of Engineering Physics, Cranfield University, Cranfield, Bedford MK43 0AL, UK

Email: t.sreenuch, a.tsourdos, i.jennions@cranfield.ac.uk

Integrated Vehicle Health Management (IVHM) is a major component in a new future fleet management paradigm where a conscious effort is made to shift aircraft maintenance from an unscheduled, reactive approach at the time of failure to a more proactive and predictive approach. Its goal is to maximize fleet operational availability while minimizing logistics footprint through monitoring deterioration of equipment conditions. A comprehensive IVHM system will be executed in an environment that includes different sensor technologies, multiple information systems and different data models. IVHM implementers have therefore to deal with an integration problem that involves different specialized algorithms and embedded hardware platforms. IVHM applications will have common execution logic and many will share the same data processing algorithms, hence development productivity and quality of IVHM applications can be increased through reusable software building blocks and algorithm libraries, or in particular by using a software development framework.

This paper presents an approach to distributed IVHM systems that offers reusable software architecture for a class of IVHM applications. The focus of this paper deals with an open software framework for development of IVHM applications stemming from the Open System Architecture for Condition Based Maintenance (OSA-CBM) specification, which is an architecture promoting interoperability, and a component framework that enables reuse, data process partitioning, configuration and rapid prototyping. The

framework is developed using Java and Internet Communications Engine (ICE) distributed middleware and its application is demonstrated through a gearbox health monitoring system, where the IVHM software is deployed on the distributed embedded devices. This approach provides software enabled capability to distribute/re-configure the IVHM data process (through the OSA-CBM common interface and data model) across the hardware platforms to meet the given system configuration. The performance evaluation for the test example shows negligible overhead in CPU, bandwidth and latency when using the framework.

I. Introduction

Integrated Vehicle Health Management (IVHM) involves data processing which comprehensively consists of capturing data related to aircraft components, monitoring parameters, assessing current or future health conditions and providing recommended maintenance actions. In today's globally competitive environment, operators are seeking to maximize their fleet usage and hence to demand a comprehensive fleet management system. More reliance is now being placed on IVHM systems to extend functional life of the fleets and to allow repairs to maximize affordability [1]. From the systems development view point, the major obstacles in implementing IVHM systems are: the development of software systems for platform distributed embedded devices and the integration of the existing/new IVHM components. A software development framework that makes use of open standards and allows software reuse would provide a great deal of benefit toward ease of proof-of-concept prototyping, development and integration between IVHM components.

Several related examples of distributed IVHM system are reported in the literature. However, these examples only address the measurement and control problems, notably those based on the IEEE 1451 smart transducer interface standards [2]. In the IEEE 1451 based implementation, the Network Capable Application Processer (NCAP) is used to enable the distribution of the measurement and control of the systems. Its applications are found in the areas of water management [3]–[4], industrial automation [5] and environmental air pollution monitoring [6]. Despite IEEE 1451 standard maturity, [3]–[6] were not implemented using any application

frameworks. They were hard-coded, and hence a significant effort would be required if there are to be changes (or updates) in the system. In fact, there are currently no projects reported in the literature actively working on the software framework that will allow rapid deployment (or prototyping) of IEEE 1451 applications.

In IVHM applications, there are many more data processing steps apart from measurement and signal conditioning. Interoperability between multiple vendors' components is central in a distributed IVHM system. Moreover, the data generated by the IVHM components should be able to map into the maintenance database for further maintenance operations or data mining. Open System Architecture for Condition Based Maintenance (OSA-CBM) is an emerging open standard which in particular addresses interoperability and enterprise systems integration [7]. [8]–[10] describe concepts in developing an IVHM retated application. In these papers, IVHM software is systematically modularized based on the standardized abstract data processing functionalities. However, the papers somewhat fall short as the interoperability and implementation issues are not addressed. In [7], IVHM applications employ OSA-CBM as a standard for data exchange. The standard is implemented using C++ and MATLAB/Simulink. In the paper, in addition to interoperability, rapid deployment and re-configuration are identified to be the future capabilities that will benefit the IVHM development and integration efforts. Until now, there is no open software development framework for generic embedded IVHM applications reported in the literature. Hence, a need exists to create a framework which will enable IVHM implementers to accelerate or ease the development (or prototyping) process and configuration of IVHM applications. In addition, such a framework should also make use of open standards (e.g. OSA-CBM) where appropriate. The framework and its usage example will be the contribution of this paper.

The outline of this paper is as follows: II describes the OSA-CBM specification and what is required for development of the framework. III outlines the proposed software development framework and describes the middleware technology used for implementing the framework. An example of how the proposed framework is applied to a specific IVHM application, i.e. gearbox, is described in IV and V. VI evaluates the performance overhead in terms of both CPU and bandwidth. VII discusses the benefits of the software framework, and then concluding remarks are made.

II. Background and Motivation

A. OSA-CBM Specification

The OSA-CBM specification [11] is an open standard architecture for moving information in a condition based maintenance system. Its goal is to address requirements for interoperability between multiple vendors' IVHM components. OSA-CBM is divided into the interface specification and information specification (or data model). These specifications are defined using the Unified Modeling Language (UML) and are intended to be platform independent. They can be mapped into various programming languages and middleware technologies.

Data Model

The OSA-CBM data model is based on the concept of metadata, i.e. OSA-CBM data are always identifiable and traceable. The aim is to have data that supports the database centric maintenance information management. In fact, OSA-CBM data can be mapped into any MIMOSA-compliant relational database maintenance systems with ease. There are 4 primary OSA-CBM data classes: DataEvent, Configuration, Explanation and Extensible. Configuration gives information about a IVHM application's input sources, a description of algorithms used for processing input data, a list of outputs and various output specifics such as engineering unit, thresholds for alerts, etc. Explanation is a reference to the data used by an IVHM application to produce an output. The Extensible class is still immature and not well defined in the specification.

DataEvent is the dynamic data related to IVHM events generated by an IVHM application such as measurements, manipulated or processed data, etc (see Fig. 1). Its metadata (e.g. id, site, confid, time, sequenceNum and alertStatus attributes) are used to identify and inform status of the data event. The DataEvent inheritance hierarchy is associated with particular abstract IVHM data processing functionalities defined in the ISO 13374 Condition Monitoring and Diagnostics of Machines – Data Processing, Communication and Presentation [12]. Those classes have inherited classes below them describing particular types of data (see Fig. 2 for examples of DADataEvent). The functional definitions can be viewed as abstractions of data into knowledge with higher and higher abstraction level of information. The DA data type is an acquired sensor

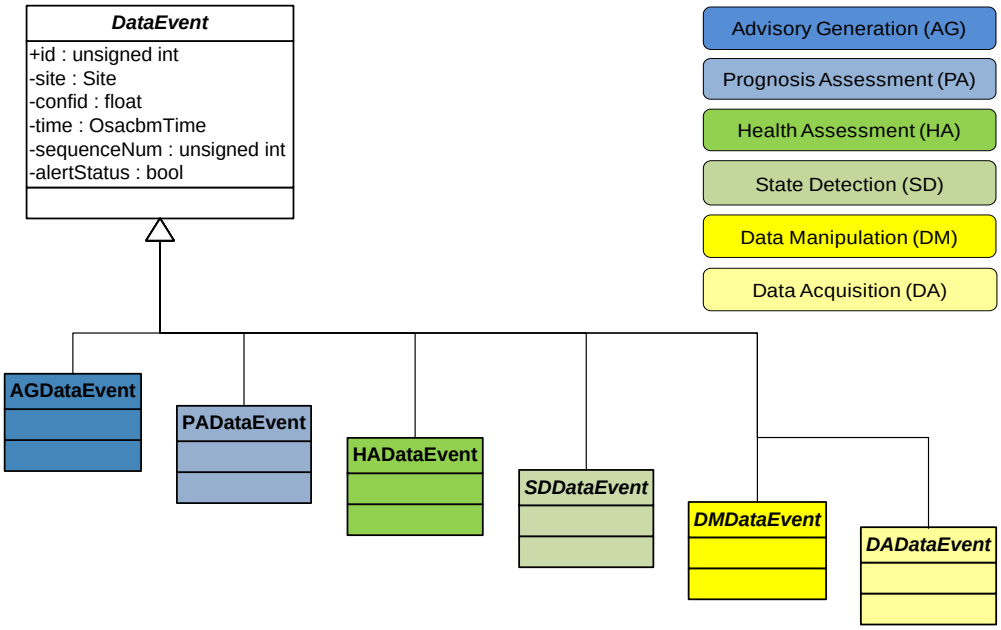


Fig. 1. Examples of OSA-CBM Data Model

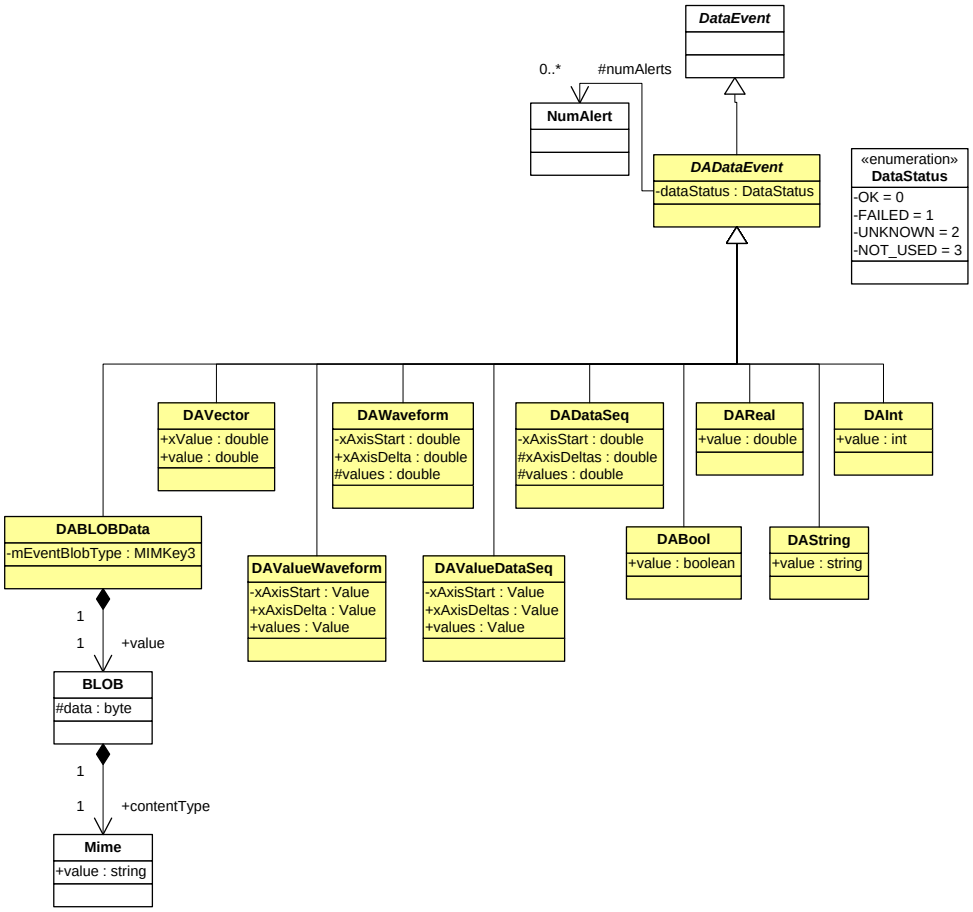


Fig. 2. Data Acquisition – DADataEvent

data which is formatted into a consistent form. The DA data is then transformed into one or more meaningful features, which are in the DM-type data format. The features are compared against expected values, and the resulting enumerated condition indicators are stored in the SDDataEvent class. The data in the HA, PA and AG formats are the data related to current health of the machine, predicted future failures and recommended action steps, respectively.

Interface Specification

The interface specification describes how information will be moved. There are 4 primary types of interface to accommodate different purposes and technological capabilities: Synchronous, Asynchronous, Data Service and DataEvent Server. The Synchronous interface returns data with the call. It models the Web XML over HTTP fetch technology. The Asynchronous interface allows any number of IVHM applications to establish and maintain a two-way connection for the duration they need. The data is returned either on request, on alert or by push all. The latter of these pushes data to the connected IVHM applications every time it collects data without the need for a request beforehand. The Service interface is for a one-way data input device. Two possible uses would be data storage utility and maintenance advisory receiver service. The DataEvent Server interface is similar to the return on alert or the push all modes of communication in the Asynchronous interface (see Fig. 3). A given implementation will likely not implement every interface types. In this paper, the DataEvent Server interface is selected for simplicity in implementation and also to naturally capture the event-driven characteristic in the IVHM applications.

B. Configurable IVHM Software

In a distributed IVHM system, available computational power and bandwidth are the major constraints. A system integrator must be able to partition or distribute an IVHM data process according to how resources are allocated. In addition, an approach that facilitates faster development of IVHM applications and updates would be very desirable.

A development framework capable of rapid deployment has to be based on a generic/common software structure for constructing an IVHM application (or data process) which is in itself an

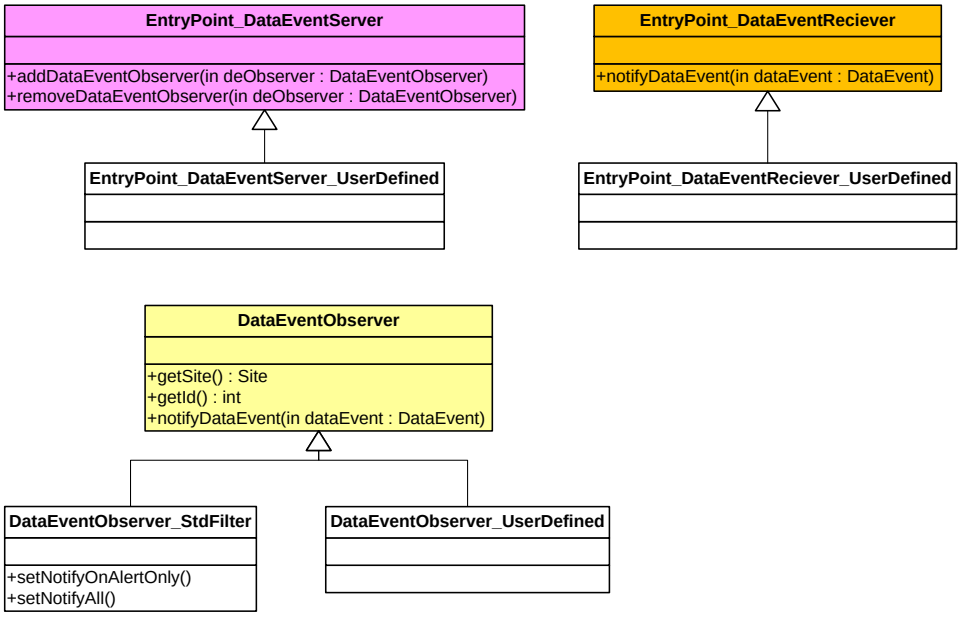


Fig. 3. OSA-CBM DataEvent Server Interface

integration of multiple discrete generic/specialized algorithms. A software wrapper can be created to make an IVHM algorithm configurable. The main executable will not change, and the application dataflow interchange need not change. Only a data process descriptor for the application needs to change. This way, an IVHM application can be constructed via processing a configuration file. A configuration file will give information about an application's inputs, algorithms used for processing input data and a list of outputs. To partition or distribute a IVHM data process, it only needs to create different configuration files for each distributed IVHM applications. The framework can allow very rapid deployment (or configuration change) of a distributed IVHM system. [7]

C. Programming Language – Java

Embedded IVHM applications are complex with different parts of the IVHM data process being distributed across different processing units to perform the data processing in a cooperative way. However, the embedded space is rapidly changing in two ways: 1) faster processors that consume less power and 2) low cost memory. This means a greater choice of development

platforms and code from a broader software community can be used. Hence, with the available CPU and memory resources, an operating system (OS) and Java technology are chosen to make the implementation process of the application framework much more productive. The OS and drivers are used to handle low-level device and network communication tasks. Virtual Machine (VM) is a key concept in Java technology. Source code is written and then compiled to a Java bytecode. The compiled bytecode is verified and interpreted under the Java Runtime Environment (JRE). The same bytecode is able to execute on various JRE platforms and still producing the same result. Therefore, a IVHM application written in Java is compiled once and can run across different Java-enabled embedded platforms (CPU: x86, PowerPC, ARM and OS: Linux, Windows Embedded). The task of porting IVHM code to different embedded devices is simplified. Moreover, until this end, our aims are to create a research software framework for prototyping IVHM applications and effectively to ease the task of building IVHM technology proof-of-concept demonstrations. This process is very experimental and highly iterative. It is more productive to code in Java than to code in C++. The task of implementing the IVHM algorithms is also reduced by using many well-developed open numerical or scientific libraries (without a need for code re-compilation). Productivity (i.e. time and ease of development) offered by Java outweighs performance offered by C++ in our case, where activities are mainly on research and proof-of-concept prototyping.

III. Application Framework

A. Distributed Middleware – Enabling Technology

In distributed IVHM systems, low-level interoperability requires IVHM messages and access methods (i.e. API) to be in a binary protocol that can be recognized by different distributed components. Middleware consists of pieces of software that handle data communication between distributed components. It sits between a low-level transport protocol (e.g. TCP, UDP) and an application. It abstracts out low-level communication processes and allows applications to be developed at a higher level. Data and API that are in a middleware-specific format will be recognizable by the distributed components using that particular type of middleware.

The degree of interoperability depends on which middleware technologies are employed. It is

desirable to have a middleware that can provide a multi-language, cross-platform environment. In this paper, Internet Communications Engine (ICE) is used as an underlying middleware for implementing the IVHM framework. ICE is a descent version of Common Object Request Broker Architecture (CORBA). It is open source and CORBA-like in terms of multi-language and cross-platform. However, ICE is much smaller and less complex than CORBA.

In order to use or validate the information content of OSA-CBM messages, a mapping of UML specification to specific programming language classes is required, see Fig. 4. The UML model is first converted into ICE interface definition language (IDL) files, called slice. These files contain OSA-CBM UML equivalent information, see Fig. 5 and 6. Note that slice files are language-independent. The ICE compilation tool is then used to generate language-of-choice classes. In our case, Java is the programming language of choice. The Java generated interface and data classes are the equivalent of the OSA-CBM specification. At the high level, these classes form parts of the OSA-CBM compliant framework and are also used in the development of specialized Functions (see III.D). They are the data structure used for storing IVHM information. Meanwhile, at the low level, they are used by the ICE components to encode/decode binary OSA-CBM messages in the data communication process.

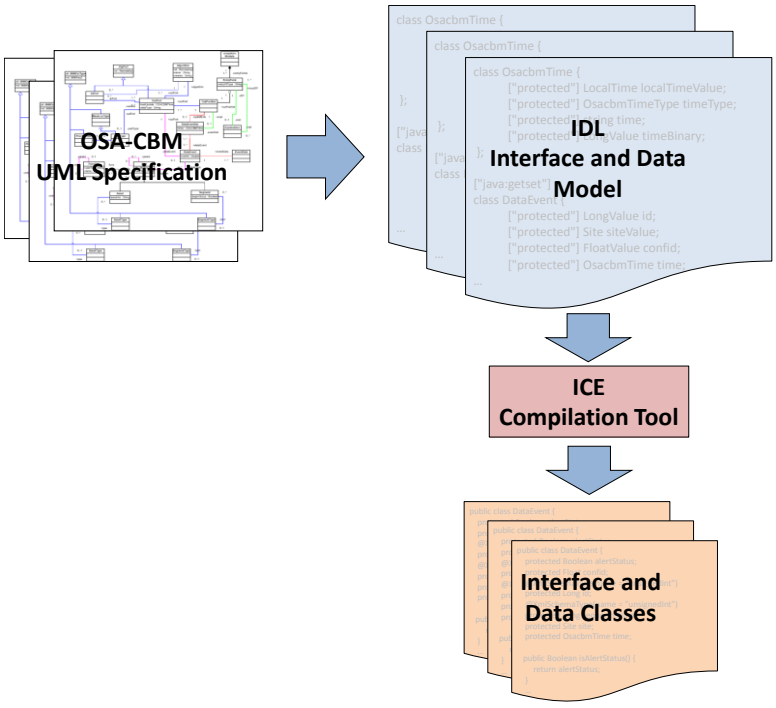


Fig. 3. OSA-CBM UML – Programming Language Mapping.

```
interface EntryPointDataEventServer {
   void addDataEventObserver(DataEventObserver deObserver);
   void removeDataEventObserver(DataEventObserver deObserver);
};

interface DataEventObserver  {
   :Site getSite();
   long getId();
   void notifyDataEvent(DataEvent de);
};

interface EntryPointDataEventReciever {
   void notifyDataEvent(DataEvent de);
};
```

Fig. 4. IDL Example of OSA-CBM DataEvent Server Interface.

```
...
class DataEvent {
    long id;
    Site siteValue;
    FloatOpt confid;
    OsacbmTime time;
    LongOpt sequenceNum;
    BoolOpt alertStatus;
};

class DADataEvent extends DataEvent {
    DataStatusOpt dataStatusValue;
    NumAlertSeq numAlerts;
};

class DAWaveform extends DADataEvent {
   DoubleOpt xAxisStart;
   double xAxisDelta;
   DoubleArray values;
};
...
```

Fig. 5. IDL Example of OSA-CBM Data Model.

Note that different IVHM applications developed using different programming languages are able to communicate with each other if the language-specific classes are generated from the same OSA-CBM IDL files. Since the OSA-CBM interface and data model are used in the IVHM

application framework, hence it is OSA-CBM compliant, from now the terms 'IVHM application' and 'OSA-CBM module' will be used interchangeably.

B. Building Blocks

To allow reuse, configuration and extension, a component framework approach is adopted in this paper. The framework aims to simplify development of IVHM applications. Several key features are identified and built into the component model that stresses modularity and extensibility. Fig. 6 illustrates a generic OSA-CBM module software structure. Its equivalent UML diagram is shown in Fig. 7. The main components are described as follows:

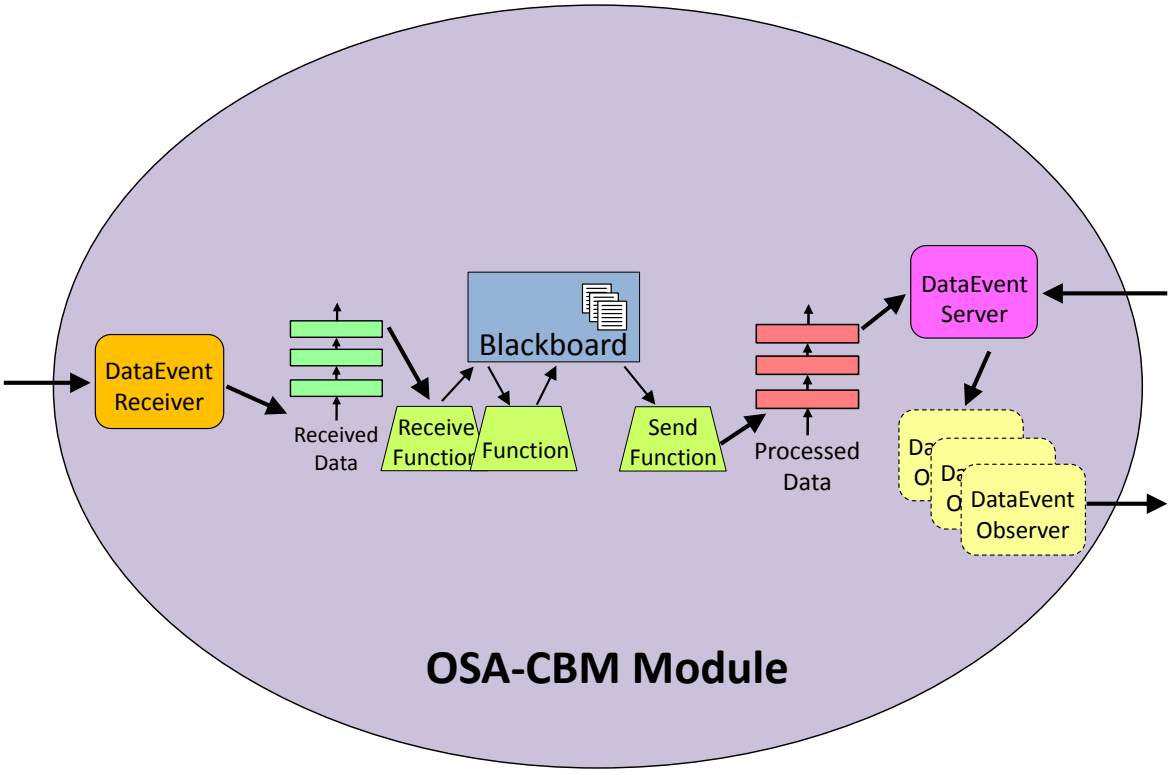


Fig. 6. Generic Component Model.

Entry Points

The OSA-CBM interface specification's Subscription defines two interfaces for an OSA-CBM module (see Fig. 3): DataEventServer and DataEventReceiver. DataEventServer is the interface provided by an OSA-CBM module to other OSA-CBM modules that have an interest in receiving data from it. When a module wants data from a server module, the requesting/client module uses the server module's DataEventServer interface to make a request. During the requesting process, the client module will provide a DataEventObserver to the server module. The DataEventObserver contains a reference to the DataEventReceiver of the client and has the notification method to be called by the server module when a new data is ready.

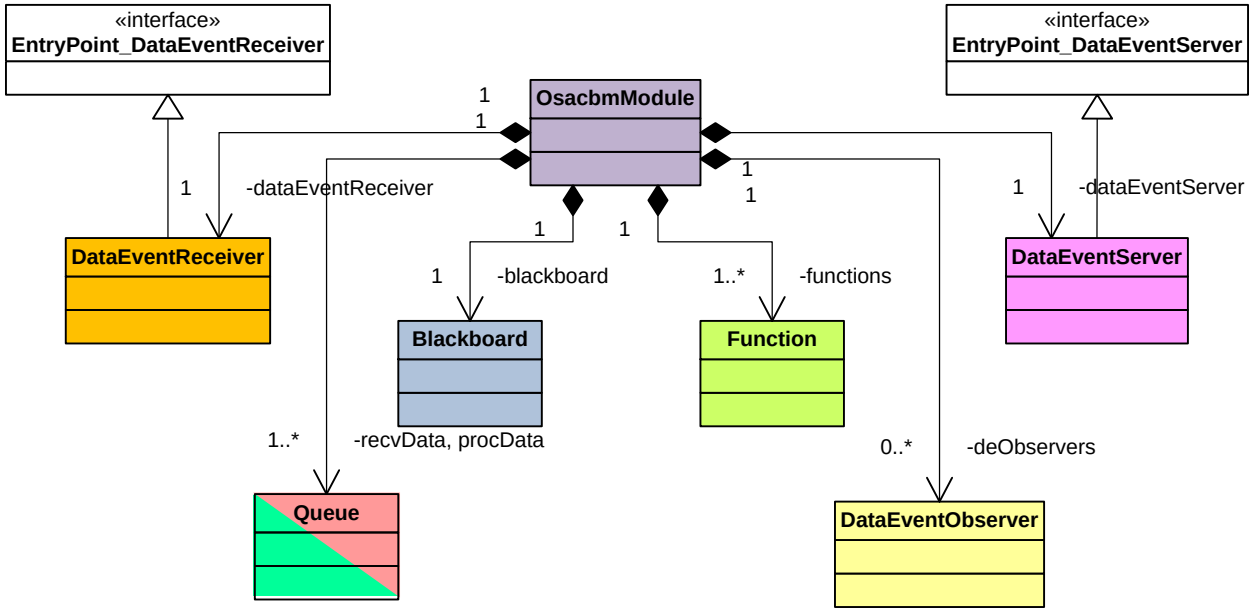


Fig. 7. UML Diagram of OSA-CBM Module.

Message Queues

The Received Data and Processed Data message queues, which are an instantiation of the Queue class (`recvData` and `procData` shown Fig. 7), provide asynchronous communication between OSA-CBM modules. A function call to update data returns immediately after the data event has been inserted in the received data message queue. The processed data message queue acts as a data buffer store before being sent to the subscribed OSA-CBM modules.

Function

A Function is a self-contained software wrapper for the data processing algorithm so that it can be used within an OSA-CBM module. Functions communicate only with the OSA-CBM module's internal components, reacting to Blackboard data events and publishing results to the Blackboard. More details of Function are described in III.D.

Blackboard

The Blackboard provides the Functions with the ability to specify and interact with data events of specific interest to that Function, see also [13]. It acts as an OSA-CBM module's shared data space. The Blackboard tracks all changes in the data event and distributes the updates to all interested/subscribed Functions. It allows the instantiated Function objects to be uncoupled.

These few generic software components (Entry Points, Message Queues, Blackboard and Functions) act as building blocks of an OSA-CBM module. The components are dynamically loaded and connected together at application start time using given configuration information. In terms of OSA-CBM application developments, developments would only have to focus on specialized algorithms and data flow, rather than being concerned about the details of software implementation and module communication.

C. Communication Model

In a distributed IVHM system, both intra- and inter-module communications are required to process different IVHM data events. In this paper, design patterns are extensively used to facilitate software component interaction. The patterns are suggested solutions, and their use leads to robust and maintainable software code. Three distribution and concurrency patterns are used in the framework; these are Publish/Subscribe, Observer and Producer/Consumer patterns.

The model of communication between Blackboard and Functions (intra module) is data-driven via the Publish/Subscribe mechanism [14] (see Fig. 8) and is used to facilitate the intra-module IVHM data processing. The Blackboard manages data flow between Functions inside an OSA-CBM module. It tracks changes in the data events and distributes the updates to all interested Functions. In this framework, the id of a data event is used as a subscribed topic. The Blackboard allows the instantiated Function objects to be uncoupled. There is only one object that the

Functions need to known – the module Blackboard. The Blackboard minimizes the work required to configure a complex dependency relationship between Functions. The Publish/Subscribe pattern enables software reuse and configurability of IVHM data processing algorithms.

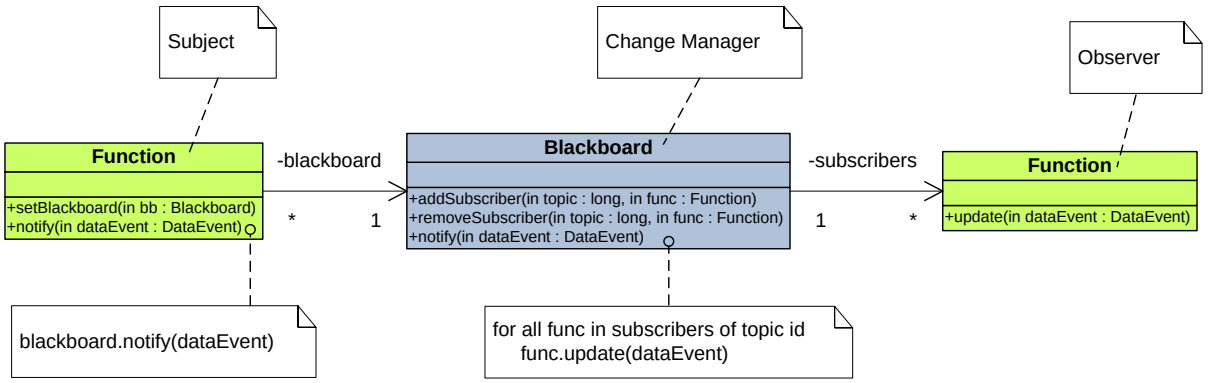


Fig. 8. Publish/Subscribe Pattern.

In this framework, inter-module communications are carried out through the OSA-CBM DataEvent Server interface. This interface implements the Observer Pattern [14] (see Fig. 9). DataEventServer provides an interface for adding and removing observer objects. A DataEventServer can have any number of dependent DataEventObservers. DataEventServer notifies its observers whenever new data is ready. After being informed of an updated data event, a DataEventObserver then notifies its associated DataEventReceiver if the data event id is in the list of interested topics.

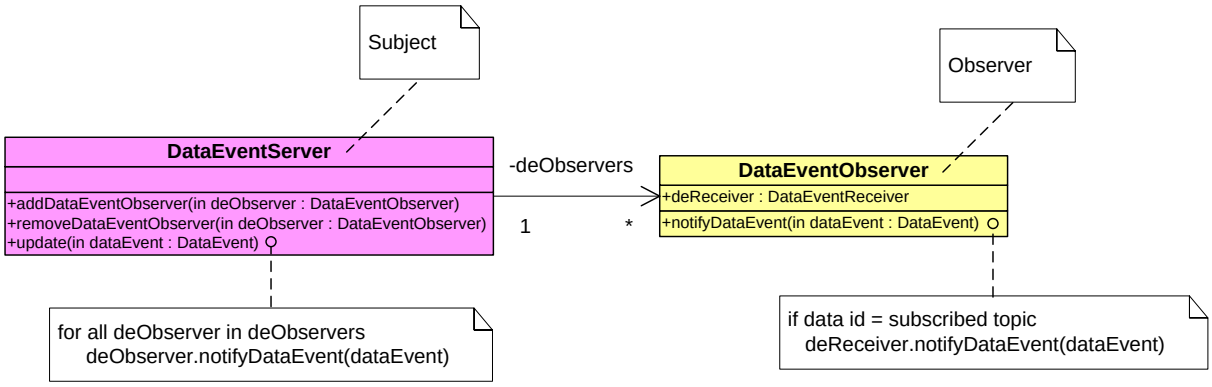


Fig. 9. Observer Pattern.

The intra-module IVHM data processing depends on the data received from other OSA-CBM module. The inter-module data communication likewise depends on the data generated by the IVHM data process within the module. Despite these dependencies, the tasks in receiving, processing and sending data can (and should) be executed in parallel. In this framework, the Producer/Consumer pattern [15] is used to solve call blocking issues (see Fig. 10). Queues are used as buffers for the received and processed data. The message queues decouple the intra-module data processing from the inter-module data communication. The module's DataEventReceiver can receive new incoming data while the IVHM algorithms (i.e. Blackboard and Functions) are still processing previously received data. Similarly, the algorithms can keep generating new processed data while the DataEventServer is still trying to send completed processed data. Hence, the CPU can be fully utilized for both tasks.

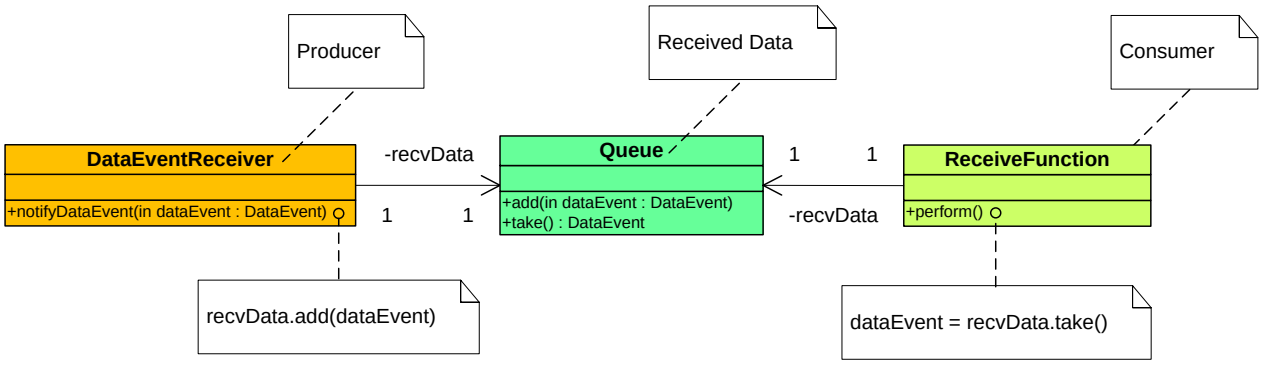


Fig. 10. Producer/Consumer Pattern.

Fig. 11 shows an example scenario of intra- and inter-module data communication. In this scenario, function #1 and #2 are wrappers for two IVHM data processing algorithms. The input and output of function #1 are data events of id 0 and 3, respectively. function #2 processes two input data events of id 1 and 2. Its output is a data event of id 4. The sequence diagram illustrates the working mechanism of the combined Publish/Subscribe, Observer and Producer/Consumer design patterns. Notice the parallelism among receiving, processing and sending tasks. For example, while the data event of id 0 is being processed, the incoming data event of id 1 is received and buffered in the recvData queue. It is similar for the data processing and sending tasks. How an algorithm handles multiple inputs is also illustrated in the sequence diagram.

Observe how function #2 buffers and synchronizes the input data (i.e. ids 1 and 2) before being processed. Various user-specific techniques can be used to synchronize multiple input data events. To this end, the Rendezvous pattern [16] is used to synchronize data for Functions. It is simple and sufficient for most of our case studies, where all the inputs of an algorithm have single data rate (SDR) and there is no significant latency related to the input data events (i.e. the inputs are kept relatively synchronized).

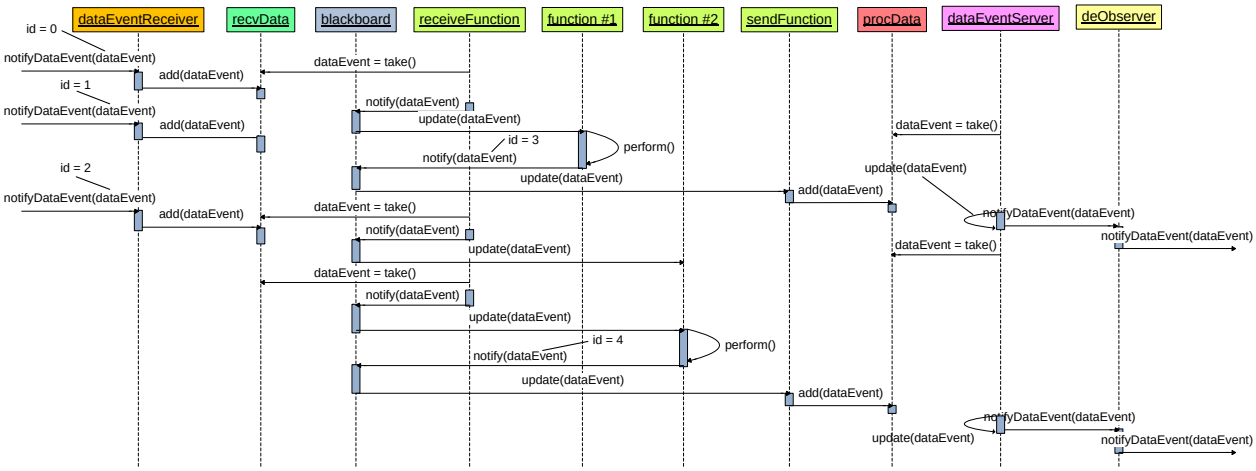


Fig. 11. Data Communication and Processing Sequence Diagram.

D. Function Development

To allow rapid development and software reuse, an OSA-CBM module's functionality is based around an object-oriented concept (see Fig. 12). The base classes of Function, TransducerFunction, DataProcessingFunction, ReceiveFunction and SendFunction provide common attributes and methods needed for basic functionalities and communications between software components within an OSA-CBM module. A specialized Function class is constructed by extending/inheriting from one of the base Functions, e.g. NiDAQmxFunction, FilterFunction, DcOffsetFunction, TsaFunction, KurtosisFunction, ResidualFunction, FftFunction, Ena4Function and BayesFunction in Fig. 12. The class colors highlight the OSA-CBM abstract functionalities described in II.A. Its algorithm-specific behaviors are implemented by overriding the pre-defined methods of the inherited Function.

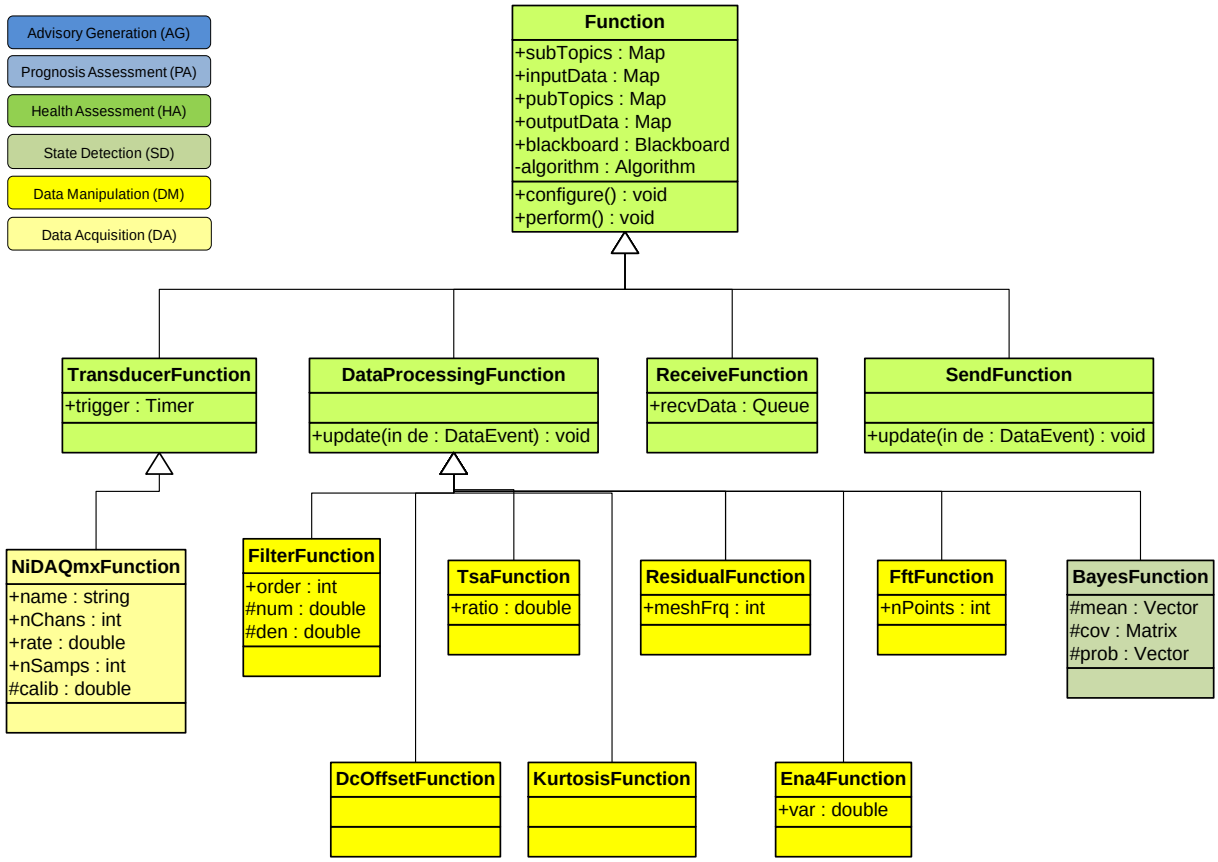


Fig. 12. Base and Gearbox-Related Specialized Functions.

Fig. 13 gives an example of how an Infinite Impulse Response (IIR) filtering functionality is implemented in this OSA-CBM framework. FilterFunction is constructed by inheriting the DataProcessingFunction. Filter characteristics are determined by coefficients of a filter's numerator and denominator [17]. The configure() method reads a given configuration (i.e. algorithm) which is encoded in the OSA-CBM Algorithm-type data format [11] and then appropriately set the filter parameters order, num and den to the required specific values. The perform() method reads an input data event, executes the filtering process VibUtils.filter(…) and packages its output data into the OSA-CBM format. In this example, a generic VibUtils.filter(…) algorithm is wrapped to create a configurable FilterFunction component.

```
Import ivhm.VibUitls;

class FilterFunction extends DataProcessingFunction {
   private int order;
   private double num[];  // Numerators
   private double den[];   // Denominators

   // filter-specific configuration, i.e. order, numerators and denominators
   // are the algorithm parameters
   @override public void configure() {
   order = algorithm.inputInts.get(0).value;  // setting filter order

   // setting filter coefficients - numerators
   num = new double[order+1];
   for (int i=0; i<(order+1); i++) {
      num[i] = algorithm.inputReals.get(i).value;
   }
   ...
   }

   // filter-specific data process, i.e. IIR filtering
   @override public void perform() {
   ...
   // filter input waveform, num and den are the filter coefficients
   wf = VibUtils.filter(num, den, ((DAWaveform)de).values);
   ...
   // package output waveform into an OSA-CBM data format
   ((RealWaveform)filteredDE).values = wf;
   ...
   }
   ...
}
```

Fig. 13. Implementation Example of Specialized Function.

In this way, Functions bring data processing functionality to an OSA-CBM module; they together are the essential compute engine of an OSA-CBM module. This approach allows a data processing flow or a complex algorithm to be built using a combination of simple specialized Functions. This flexibility allows the developer to easily reuse pre-developed Functions in OSA-CBM module construction or customize a IVHM application to meet a required IVHM functionality.

IV. Example

A. Gearbox Health Monitoring

A gearbox vibration monitoring example is chosen to demonstrate the adequacy and effectiveness of the distributed IVHM software development framework. The purpose of the IVHM system is to monitor the vibration signals produced by the gearbox and determine the level of damage (or fault) of its pinion gear. Fig. 14 shows the experimental platform and the related IVHM data processing chain. In this setup, an accelerometer is mounted on top of the gearbox to measure the vibration in the axial dimension, and an optical sensor is used to provide a one pulse per revolution signal used for measuring the speed of the gear shaft. A data acquisition board is used to continuously collect 20 s batch data for the vibration analysis at the rate of 25 kHz.

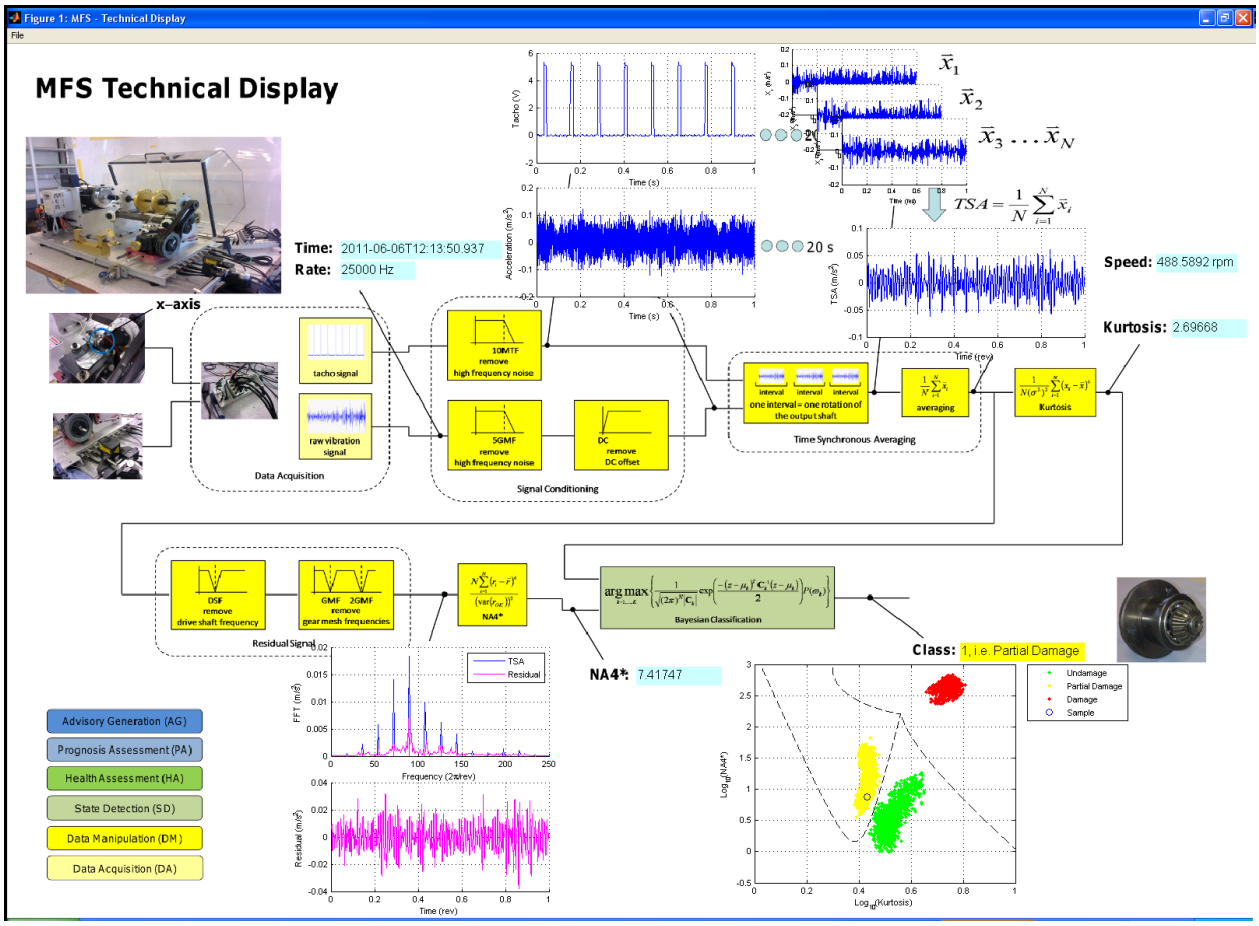


Fig. 14. Screenshot of Gearbox Health Monitoring Technical Display.

The data processing algorithms used in this paper range from data manipulation to state detection. Based on collected data and an extensive analysis, a combination of Kurtosis and NA4* (or ENA4) of the axial vibration is sufficient for accurate fault detections in this particular system. Note that we only outline one possible example of data processing flow in this paper, an extensive discussion of the gearbox vibration analysis can be found in [18]–[20]. To determine the condition of the gearbox, the raw tacho and acceleration signals are first conditioned by passing through the low-pass filters and the DC offset remover. The random vibrations and external disturbances are further removed by averaging multiple data segments, each segment equals to one revolution of the gear shaft. This technique is known as Time Synchronous Averaging (TSA). In addition, the shaft rotation speed can also be obtained during the TSA computation. The Kurtosis and NA4* features are then extracted from the TSA and residual signals, respectively. The residual is computed by removing the shaft frequency and gear mesh harmonics from the TSA signal. The aim is to have an analytic signal (i.e. the residual signal) which is less dependent on the speed and load of the gearbox. The Bayesian classifier processes the Kurtosis and NA4* input features and determines the gearbox condition based on the statistical properties of the training data.

B. Applying the Framework

The development framework described in III is employed in building the OSA-CBM based embedded IVHM system. The first step of the implementation is to categorize the underlying algorithms according to the ISO 13374 abstract functionalities shown in Fig. 1, i.e. DA – NI DAQmx, DM – Filter, DC Offset, TSA, Kurtosis, Residual, NA4*, FFT and SD – Bayesian Classifier. In this paper, the NI DAQmx C library and Java Native Interface (JNI) are used for low-level communication with the NI data acquisition board, and the vibration processing algorithms are implemented based on the Java Apache Commons Mathematics Library. These generic vibration algorithms are then wrapped to form an OSA-CBM algorithms library by extending/inheriting either the TransducerFunction or DataProcessingFunction, and the resulting gearbox vibration analysis specific Functions are shown in Fig. 12. Note that the implemented algorithm wrappers (i.e. Functions) are generic and configurable (see Fig. 13). The configuration parameters can be the number of inputs and outputs, expected input data type and algorithm

parameters.

Fig. 15 gives a concrete example of how a Function can be implemented to have non pre-defined number of inputs and parameters' dimension. Bayesian classifier is a classification algorithm which is generalized for *n*-dimensional space [21]. The classifier parameters (i.e. mean vectors, co-variance matrices and prior probability vectors) can be any dimensions depending on the dimension of input vector (or feature vector) and the number of enumerated classes. In BayesFunction, the classifier characteristics are determined by nFeatures, nClasses, mean, covariance and priorProb parameters. The configure() method reads a given configuration and then appropriated set the parameters to the required specific sizes, dimensions and values. In configure() method, the mean vectors are initialized to the required number. Each vector is then configured to have a required dimension and finally its vector components are one-by-one set to the specific values. The configuration steps are similar for covariance and priorProb which the details are omitted to simplify the figure. The perform() method unpacks input OSA-CBM data events and re-packages the data into the format required by the classification algorithm. The number of input data events is determined by the configurable nFeatures parameter. VibUtils.bayesClassify(...) is then called to execute the classification process, and its return output is finally packaged into the OSA-CBM data format. In this example, a generic algorithm (i.e. VibUtils.bayesClassify(...)) is wrapped to create a highly generic configurable BayesFunction component which can be reused in different IVHM data processes.

```
import ivhm.VibUitls;
import org.apache.commons.math.linear.*;

class BayesFunction extends DataProcessingFunction {
   private int nFeatures;  // dimension of input  vector
   private int nClasses; // number of classes
   private RealVector[] mean;    // mean vectors
   private RealMatrix[] covariance; // covariance matrices
   private RealVector[] priorProb; // prior probabilities

   // Bayes-specific configuration, i.e. dimension, number of classes, mean,
   // covariance and prior probabilities are the algorithm parameters
   @override public void configure() {
      nClasses = algorithm.inputInts.get(0).value;// setting vector dimension
      nFeatures = algorithm.inputInts.get(0).value;  // setting number of classes

      // setting mean of each classes
      mean = new RealVector[nClasses];
      for (int i=0; i<nClasses; i++) {
         mean[i] = new ArrayRealVector(nFeatures);
         For (int j=0; j<nFeatures; j++) {
            int idx = i*nFeatures + j;
            mean[i].setEntry(j, algorithm.inputReals.get(idx).value);
         }
      }
      ...
   }

   // Bayes-specific data process, i.e. classification based on probabilities
   @override public void perform() {
      ...
      // unpackage OSA-CBM input data into a feature vector
      RealVector features = new ArrayRealVector(nFeatures);
      for (int i=0; i<nFeatures; i++) {
         features.setEntry(i, ((DMReal)dataEvents[i]).value);
      }
      ...
      // determine which class based on given features and statistical properties
      // of the based data.
      int damageLevel = VibUtils.bayesClassify(features, mean, covariance, priorPob);
      ...
      // package output state into an OSA-CBM data format
      ((SDInt)levelDEfilteredDE).value = damageLevel;
      ...
   }
   ...
}
```

Fig. 15. Implementation Example of Algorithm Wrapper – BayesFunction.

Fig. 16 shows the network setup of the distributed IVHM system. The objective of this system is to partition the IVHM data processing chain described in IV.A and to distribute the algorithms to form a hierarchical IVHM system. In this setup, the OSA-CBM module #1 hosts the ReceiveFunction, NiDAQmxFunction, FilterFunctions, DcOffsetFunction, TsaFunction and SendFunction. It runs on the sensor node which is an Intel Atom D510 single-board computer attached to NI data acquisition board and has Windows XP Embedded, Java SE 6 and NI DAQmx C library installed. This OSA-CBM sensor node is for acquiring the tacho and vibration signals and to perform low-level data processing from filtering up until TSA. In this phase of data processing, for each 20 s batch data, the vibration information is compressed from 500,000 samples of tacho signal and 500,000 samples of acceleration signal to a TSA signal of 4,096 double data points.

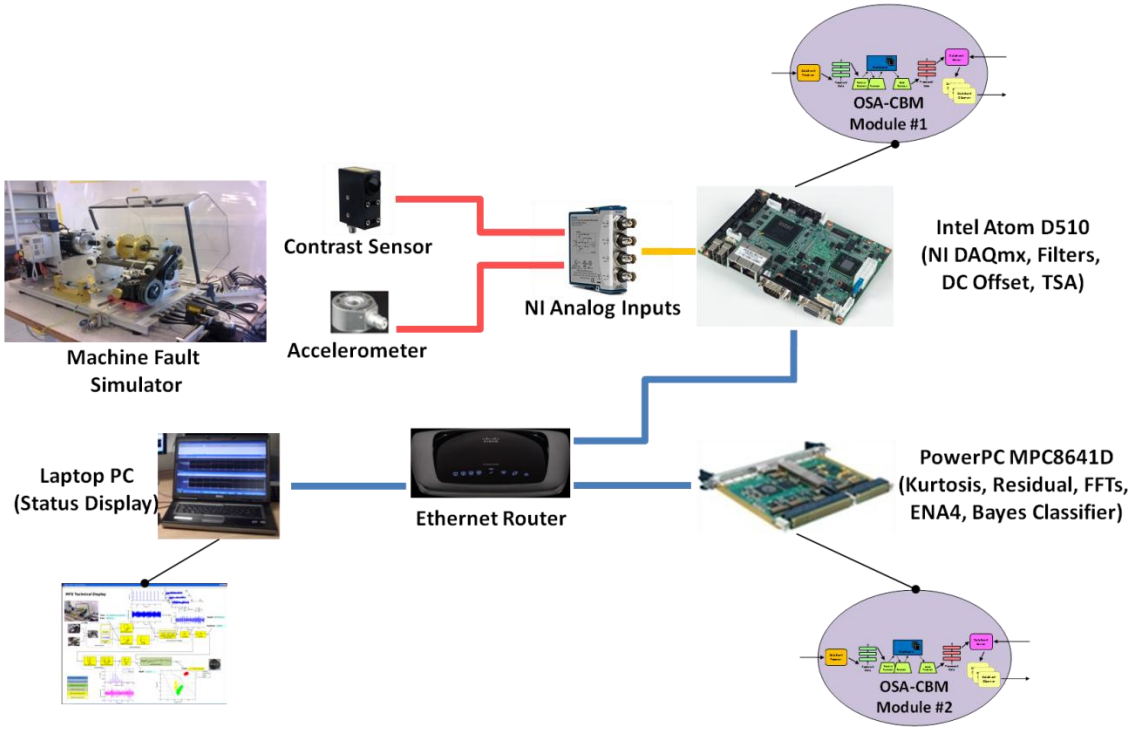


Fig. 16. Distributed Gearbox Health Monitoring System.

The OSA-CBM module #2 hosts the ReceiveFunction, KurtosisFunction, ResidualFunction, Ena4Function, FftFunction, BayesFunction and SendFunction. It runs in the health reasoner which is a Freescale MPC8641D flight- worthy single-board computer and has a pre-compiled

Linux file system from the Embedded Linux Development Kit (ELDK) version 4.2 and OpenJDK 6 installed. The health reasoner extracts the Kurtosis and NA4* features from the TSA data generated by the sensor node and determine the current health state of the gearbox.

At each OSA-CBM module start time, the module specific binary configuration file is loaded and the module is created by means of runtime instantiation and configuration of the pre-compiled Functions. The configuration file contains the module description, a description of algorithms (incl. parameters) used for data processing and lists of individual algorithms' inputs and outputs.

Note that the current experimental setup can be extended by having a high-level IVHM system take more pre-processed data from other local sensor nodes (e.g. another current sensor and processing node attached to the motor). The high-level module can then fuse all the data available and perform a more thorough system-level condition monitoring, for example, an overall drive-train system condition monitoring.

V. Demonstration

To display the output data from the OSA-CBM embedded nodes, we create a small MATLAB GUI displays shown in Fig. 14. The m-files are compiled into deployable Java classes using MATLAB Builder JA (for Java language). The Java display module integrates the deployable Java classes (i.e. compiled MATLAB GUI components) for the graph plotting and text updating functionalities and also implements the standardized OSA-CBM DataEventReceiver interface for interoperability with the OSA-CBM modules.

In this setup, a static executive module (a small Java console program) is used to manage the data subscription or un-subscription between the OSA-CBM module #1, OSA-CBM module #2 and display program. The subscription and data updating mechanisms follow the design pattern shown in Fig. 9.

The CM data process operates at the fixed rate schedule of 20 s timeframe. The mean elapsed time between the point when the raw tacho and acceleration data are acquired and the condition level of the gearbox is computed is <1 s, which is specifically less than the timeframe required

for real-time operation. Note that this level of computational power is typical for the embedded single-board computers used in aerospace vehicles.

The data subscription or un-subscription between the OSA-CBM module #1 and #2 can be established dynamically at any time during the operation, through the addDataEventObserver(...) and removeDataEventObserver(...) APIs. Once a connection is established, the module #1 pushes the updated data to the module #2, and the module #2 responses to the arriving data events generated by the module #1. On the other hand, once the un-subscription is requested, the module #1 stops sending the updated data to the module #2. The module #2 effectively becomes idle, waiting for the incoming data events.

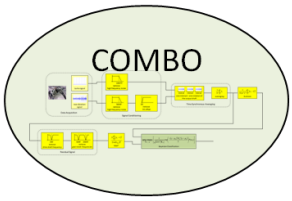
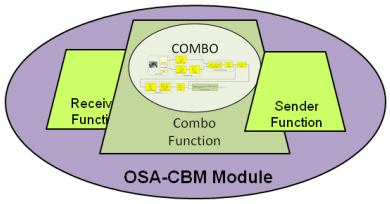
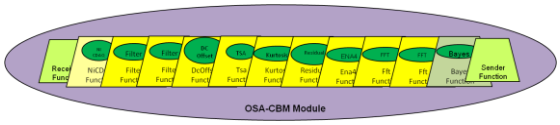
VI. Performance Overhead

In this framework, the requirements of interoperability, rapid prototyping and data processing reconfiguration are met through the use of common software structure, configurable software components and OSA-CBM interface. However, despite the benefits of the framework, what is the performance overhead added by using such a framework? This question is often asked by IVHM implementers when considering alternative approaches to developing IVHM applications.

One fundamental measure of the computation platform is speed. In this paper, speed means the elapse time to perform an IVHM data processing task. The IVHM data process shown in Fig. 14 is used in the test. Three configurations are considered (see Table 1): 1. The IVHM data process is implemented as a Java program. 2. The whole IVHM data process is wrapped in one Function and then packaged in an OSA-CBM module. The ComboFunction used in the configuration #2 cannot be reused and configured. 3. Wrappers are created for the individual algorithms and then packaged in an OSA-CBM module. This configuration gives maximum reusability and configurability.

Table 1

CPU Overhead

	Configuration	<Elapse Time>	<Overhead>
1.		427.075 ms	N/A
2.		418.136 ms	0.003 ms
3.		416.817 ms	0.199 ms

Note: Trapezoidal – an instantiation of Function class shown in Fig. 12. <Elapse Time> are results from averaging over 1,000 data processing cycles. <Overhead> are results from averaging over 1,000,000 iterations of (un-)packaging between raw and OSA-CBM data formats.

The speed test results shown in Table 1 are obtained using the Intel Atom D510 single-board computer running Windows XP, but this result is found to be counter intuitive. The elapse time for configuration #1 should be the lowest followed by configuration #2. Nevertheless, the differences are very small, i.e. ~2.5% of each configuration. This indicates a very small overhead added by the framework. In this case, the fluctuation caused by the JVM garbage collection and XP OS have more effect on the computation than the framework.

To determine precisely the size of the overhead, a program to package and un-package data and metadata between their raw and OSA-CBM formats was created. For the configuration #2, only the gear health level (an integer) is packaged into an SDInt data. For configuration #3, data are packaged when the Functions output the data and unpackaged when the Functions receive the data. The overhead are 0.003 and 0.199 ms for configurations #2 and #3, respectively. It only costs 0.05% of the total CPU time to obtain maximum reusability and configurability, i.e.

configuration #3. The CPU overhead added by the framework is negligible in this case.

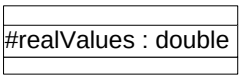
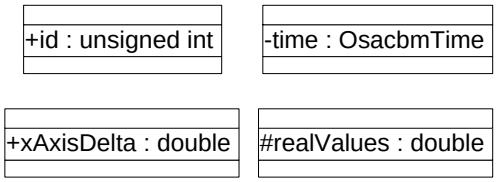
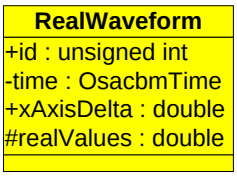
Note that the effort to package/un-package between raw data and metadata and OSA-CBM data format will depend on the data structures used to store the information. For example, RealWaveform uses an array of double data type to store waveform data. If an algorithm also produces a double array as an output, then we can expect a very small overhead. For a contrary example, CmplxFrqSpect uses two double arrays to store a spectrum data. If an FFT algorithm generates an array of Complex (a data structure for complex numbers used in Apache Commons Mathematics Library), then we have to un-wrap the real and imaginary double values from each Complex value in the array and then store them in two arrays of double data. We then expect a higher overhead (but not excessive) for this example.

Other important measures of performance are bandwidth and latency. In this paper, latency means the amount of time it takes for a client to invoke a data notification method in a server. The sending of TSA data events from the OSA-CBM module #1 to the OSA-CBM module #2 is considered. Three data notification settings are used in this test (see Table 2): 1. Only TSA waveform information, which is an array of 4096 double data points, is being sent. 2. TSA waveform and metadata are being sent. 3. RealWaveform data containing both TSA and metadata is being sent, i.e. data communication between OSA-CBM modules. This setting gives interoperability to the data.

The bandwidth and latency test results shown in Table 2 are obtained using the Intel Atom D510 and Freescale 8641 PowerPC single-board computers. The client-server configuration is as shown in Fig. 16. For the setting #1 and #2, a client and server programs is created to perform the task. For the setting #1 and #3, the average bandwidth and latency are similar. The OSA-CBM overhead is 0.086 kbits/s for the bandwidth and 0.003 ms for the latency, which are equivalent to 0.66% and 2.59% respectively. However, the overhead is significant higher if the metadata to be sent separately. The setting #2 has the overhead of 0.215 kbits/s bandwidth and 0.084 ms latency, which are 1.64% and 72.41% respectively. In each distributed method invocation, there is a header added to the data packet and a connection to establish. Hence, a significant overhead is incurred for the second setting. It can be seen that the bandwidth and latency overhead of data interoperability is very small. In fact, the OSA-CBM data model used in our framework helps to reduce bandwidth and latency if having to send both data and its metadata.

Table 2

Bandwidth Overhead

	Data	<Bandwidth>	<Latency>
1.		13.032 kbits/s	0.116 ms
2.		13.247 kbits/s	0.200 ms
3.		13.118 kbits/s	0.119 ms

Note: The experiments were carried out on a Gigabit Ethernet link. <Latency> are results from averaging over 1,000 iterations. <Bandwidth> are captured using WireShark. The results are average bandwidth of continuing runs of 20s data notification cycle.

In addition to latency and bandwidth, how responsive of an OSA-CBM module is to an incoming data event is another performance related metric. However, this is not an OSA-CBM performance overhead but it is worth discussing for completeness. The responsiveness will depend on computational load of the OSA-CBM module. In this example, ~0.417 s is the time that takes to compute a 20s batch data event. The data process can be completed ~19.5 s before a new data event arrives, and hence ~0 s waiting time is expected. In our case, the available CPU resource could allow other 40 more of similar IVHM data processes to be packaged in the OSA-CBM module. In this framework, data events are processed in a sequential manner. While a data event is being processed, other incoming data events will be buffered in the message queue. For simplicity, let consider an OSA-CBM module with 40 data processes, each of which takes 0.417 s to complete. In this framework, data events are processed on a first come, first serve (FCFS) basis, therefore on average it will take

$$\langle \text{Waiting Time} \rangle = \frac{T(N-1)N}{2N} = \frac{0.417(40-1)40}{2\times 40} = 8.132\,\text{s} \quad [22]$$

for an incoming data event to be processed. The maximum waiting time however will simply be (40-1)×0.417 = 16.263 s. In this simple example, it can be seen that how responsive a module is will depend on a number of data processing tasks and their required CPU loads. However, for non-identical data processes, scheduling and how to configure an OSA-CBM module in a limited resource environment will require substantial further research which can be an area of expansion in the future.

VII. Conclusion and Discussion

A standard like OSA-CBM benefits in easing integration of multiple vendors' IVHM software components. The interoperability issue is addressed through the common standardized interface and data model. To the IVHM application developers, OSA-CBM saves considerable time and effort required to develop an architecture and related data classes. The developer can make use of the already well designed API and data model.

To enable reuse, data process partitioning with configurable and rapid deployment, a development framework like the one described in III is required. The proposed component model eases the IVHM implementation process as the common/generic IVHM tasks are handled by the framework and the developed OSA-CBM algorithm wrappers can be reused in a new application. Developers can concentrate on the application logic, i.e. IVHM data processing and resource allocation, rather than being concerned about the details of software implementation and data communication. With pre-existing OSA-CBM algorithm libraries, the task of creating a new IVHM application could be simplified to a matter of writing the OSA-CBM module configuration files.

Interoperability and rapid prototyping are key requirements in this paper. The framework addresses the interoperability requirement through its underlying OSA-CBM data model and OSA-CBM remote interface. In this framework, algorithms are formatted into standardized configurable Functions, which form a reusable OSA-CBM algorithm library. Moreover, OSA-CBM modules developed using this framework share a common software structure. Since the

software structure is known in advance and its components are standardized, hence a rapid prototyping is possible. Based on a given configuration information, a configuration software can be used to dynamically configure/instantiate an OSA-CBM module (as used in this paper) or auto-generate static code to be further compiled into an executable OSA-CBM module. In the gearbox example, the interoperability and rapid prototyping are demonstrated. For this particular test example, the empirical evaluation shows small performance overhead in terms of CPU, bandwidth and latency. In particular, bandwidth and latency are actually improved if the data and metadata are packaged and sent in the OSA-CBM data format.

In this paper, the framework is demonstrated through a gearbox example. However, this is only one example, and hence the question that follows would then be "How generic can the framework be?". In generic sense, most (if not all) IVHM data processes can be constructed using multiple discrete generic/specialized algorithms. These algorithms are connected forming a data processing flow which will probably be similar to the diagram shown in Fig. 14. In this way, specialized Functions can be created (or reused) for the algorithms in the data process as similarly shown in Fig. 12. These Functions are then packaged into an OSA-CBM module and configured according to the specified input/output and parameter information. Therefore, if an IVHM data processing flow is transparent in terms of algorithms and relationships between them, then it is likely that the proposed framework will be applicable.

If an IVHM network is relatively fixed, then connections between OSA-CBM modules are usually pre-configured and run-time data (un-)subscriptions are then redundant. However, IVHM is still a relatively immature area, and hence insertion of IVHM technologies is likely to be incremental based on availability of resources or technologies. Moreover, proof-of-concept activities are expected to be integrated and carried out with an existing IVHM system. In a dynamic changing IVHM network, if a need of rework for affected OSA-CBM modules is to be minimized, then dynamic data subscriptions will become a necessary feature. The remote interface like OSA-CBM facilitates proof-of-concept activities and dynamic connections of a new OSA-CBM module or other IVHM-related devices (e.g. Portable Maintenance Aid). Note that IVHM systems gear towards maintenance purposes. The requirements can be less stringent than what is required in the safety critical systems (SCSs). Hence, dynamic data subscriptions, which do not exist in SCSs, could be a possible attribute in the IVHM systems.

In this paper, Java is used to implement the proposed framework primary due to its

productivity. However, C++ is a widely used programming language in the development of embedded applications. Hence, "Can this framework be implemented using C++?" is worth discussing. Within OSA-CBM module, data events (or references) are passed between Entry Points, Message Queues, Blackboard and Functions. There is no centralized component that will keep track the usage of data events. This will in general create a memory leak in C++ unless a garbage collection is used. However, in order to utilize the garbage collection, additional related smart pointer classes must be implemented either by coding manually or auto-generated depending on the employed middleware. If ICE is the underlying middleware technology, then the associated smart pointer classes are auto-generated together with the OSA-CBM C++ classes. Hence, in terms of memory, it will be straightforward to implement this framework in C++.

In addition, reflection is not supported in C++. In Java, reflection enables Class information of an object to be easily identified. Now, let consider the remote method notifyDataEvent(in dataEvent : DataEvent) in Fig. 3. "How do we make correct type-casting of a DataEvent?" is another issue if C++ is to be used. Port is a class within OSA-CBM's Configuration data classes for storing module (or algorithm)'s input/output information (e.g. id, OsacbmDataType) [11]. An OSA-CBM module (or a Function object) can appropriately typecast a receiving DataEvent object using the pre-supplied input/output configuration information. This way, a highly generic Function class can be developed to allow multiple input/output data event types.

Acknowledgment

This work was funded by Integrated Vehicle Health Management Centre, Cranfield University, UK. The authors would like to thank the IVHM Centre's industrial partners (BAE Systems, Boeing, Meggitt, Rolls-Royce and Thales) for giving numerous feedbacks and suggestions throughout the development of this work and also for helping in many other ways. The authors would also like to thank the reviewers and associated editor for their comments which essentially help to improve the manuscript.

References

[1] K. Swearingen and K. Keller, "Health ready systems," in *Proceedings of the IEEE AUTOESTCON*, Baltimore, MD, 2007, pp. 625–631.

[2] IEEE, *IEEE 1451.1 Standard for a Smart Transducer Interface for Sensors and Actuators – Network Capable Application Processor (NCAP) Information Model*. The Institute of Electrical and Electronics Engineers, Inc., 1999.

[3] K. B. Lee and R. D. Schneeman, "Distributed measurement and control based on the IEEE 1451 smart transducer interface standards," *IEEE Transactions on Instrumentation and Measurement*, vol. 49, no. 3, Jun. 2000, pp. 621–627.

[4] K. B. Lee and R. D. Schneeman, "Internet-based distributed measurement and control applications," in *IEEE Instrumentation & Measurement Magazine*, Jun, 1999, pp. 23–27.

[5] E. A. Batista, L. Gonda, A. C. R. da Silva, S. R. Rossi, M. C. Pereira, A. A. de Carvalho and C. E. Cugnasca, "HW/SW for an intelligent transducer network based on IEEE 1451 standard," *Computer Standards & Interfaces,* vol. 34, no. 1, 2012, pp. 1–13.

[6] N. Kularatna and B. H. Sudantha, "An environmental air pollution monitoring system based on the IEEE 1451 standard for low cost requirements," *IEEE Sensors Journal*, vol. 8, no. 4, Apr. 2008, pp. 415–422.

[7] K. Swearingen, W. Majkowski, B. Bruggeman, D. Gilbertson, J. Dunsdon and B. Sykes, "An open system architecture for condition based maintenance overview," in *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT, 2007, pp. 3717–3724.

[8] D. Espindola, L. Fumagalli, M. Garetti, S. Botelho and C. Pereira, "An adaptation of OSA-CBM architecture for human-computer interface through mixed interface," in *9th IEEE International Conference on Industrial Informatics*, Lisbon, 2011.

[9] L. H. Xia, L. Q. Rong, M. Zhao, L. X. Wang and Q. Man, "Research on open system architecture for equipment health management based on OSA-CBM," in *IEEE International Conference on Intelligent Computing and Intelligent Systems*, Xiamen, 2010.

[10] L. Li, Y. L. Qian, K. Du and Y. M. Yang, "A fast development framework for condition-based maintenance systems," in *the 2nd International Conference on Mechanical and Electronics Engineering*, Kyoto, 2010.

[11] MIMOSA, *OSA-CBM UML Specification 3.3.1 Release*. Machine Information Management Open Systems Alliance, 2010.

[12]ISO, *13374-2 Condition Monitoring and Diagnostics of Machines – Data Processing, Communication and Presentation – Part 2: Data Processing.* International Organization for Standardization, 2007.

[13]A. Helsinger and T. Wright, "Cougaar: a robust configurable multi-agent platform," in *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT, 2005.

[14]E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

[15]M. Grand, Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. Wiley. 1998.

[16]B. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley. 2003.

[17]S. Stearns and D. Hush, *Digital Signal Processing in MATLAB, second edition*. CRC Press. 2011.

[18]M. Lebold, K. McClintic, R. Campbell, C. Byington and E. Song, "Review of vibration analysis methods for gearbox diagnostics and prognostics," in *the 54th Meeting of the Society for Machinery Failure Prevention Technology*, Virginia Beach, VA, 2000, pp. 623–634.

[19]P. Vecer, M. Kreidl and R. Smid, "Condition Indicators for Gearbox Condition Monitoring Systems," *Acta Polytechnica*,vol. 45, no. 65, 2005, pp. 35–43.

[20]L. Gelman, I. K. Jennions nad I. Petrunin, "Detection of chipped tooth in gears by the novel vibration residual technology," *International Journal of the PHM Society*, vol. 2, no.2, 2011.

[21]F. van der Heijden, R. Duin, D. de Ridder and D. Tax, *Classification, Parameter Estimation and State Estimation: An Engineering Approach using MATLAB*. John wiley & Sons. 2004.

[22]J. Saltzer and M Frans Kaashoek, *Principle of Computer System Design: An Introduction*. Morgan Kaufmann. 2009.

https://dspace.org/

Software framework for prototyping embedded integrated vehicle health management applications

Sreenuch, Tarapong; Tsourdos, Antonios; Jennions, Ian K.

2014-02-28T00:00:00Z

http://dx.doi.org/10.2514/1.I010046

T. Sreenuch, A. Tsourdos and I. K. Jennions, Software framework for prototyping embedded integrated vehicle health management applications, Journal of Aerospace Information Systems, Volume 11, Number 2, 2014, Pages 82-97.

Downloaded from a DSpace repository. https://dspace.org/