Evolving the Data Management Backbone: Binary OSA-CBM and Code Generation for OSA-EAI

Andreas Löhr[1] and Matthias Buderath[2]

[1]*Linova Software GmbH, München, 80805, Germany*
andreas.loehr@linova.de

[2]*Airbus Defence & Space, Manching, 85077, Germany*
matthias.buderath@cassidian.com

ABSTRACT

Integrated system health monitoring and management (ISHM) is a field of research and development where both academia and industry is highly focused on. Airbus Defence & Space has recognized that simulation is a key capability for developing ISHM technologies and is therefore in the process of developing a comprehensive simulation framework in that area. One significant building block is to invite 1st class technology providers, e.g. Universities and SMIs, to provide innovative technologies and support their integration into the simulation framework. This paper is a joint presentation of Airbus Defence & Space and Linova Software GmbH, an Airbus Defence & Space preferred software provider. The Open System Architecture for Condition-based Maintenance (OSA-CBM) and Open System Architecture for Enterprise Application Integration (OSA-EAI) are complementary reference architectures and represent an emerging standard for application domain-independent asset and condition data management. The architectures address several challenges in building Prognostic Health Management (PHM) systems, which are commonly composed of disparate and distributed hard- and software components. Therefore, a common challenge to PHM systems is to be confronted with vast amounts of data which are exchanged over a heterogeneous collection of communication channels. Any such system's success depends upon an open, uniform, and performance-optimized solution for data management. A solution that includes: data definition, data communication, and data storage. We will follow up on previous work and report on our experiences from implementing our second generation data management backbone based on binary OSA-CBM transmission. We also aim at implementing a fully OSA-EAI compliant database. We confirmed the general feasibility of OSA-CBM and OSA-EAI by previous work. We have now migrated our data management backbone to the current release of OSA-CBM, which includes a standard binary transportation format. We report on our experience from implementing this format and discuss issues regarding message handling and Meta data overhead. In previous work we used a simplified and stripped-down implementation of OSA-EAI and our current goal was to be fully compliant with the OSA-EAI standard. In order to reach this goal, we have created a code generator which receives OSA-EAI-provided documentation artifacts as input. It produces compileable source code for a Java-based 3-tier OSA-EAI information system. We have identified issues with the OSA-EAI standard regarding completeness and handling, which we discuss, and suggest means for mitigation or enhancements to the standard. To underline the feasibility of our solutions, we provide empirical evidence drawn from our work. The conclusion is a summary of our experience and the direction of future work in the area of PHM system design for aircraft maintenance. In total, our contribution to the community is best seen from a practitioner's perspective.

1. INTRODUCTION – MIMOSA STANDARDS

The paradigm shift from prevention towards prediction, which PHM systems impose to maintenance and operational processes of technical system, promise higher availability and higher operational capability, coupled with a reduction of overall maintenance costs. The challenges, which programs to introduce PHM systems in any application domain must face, are twofold. The *enablers challenge* deals with developing enabling technology, such as novel sensors, state detection, and health assessment methodologies and models for determining future life of (possibly deteriorated) components. The *data challenge* deals with integrating heterogeneous data from disparate and distributed sources into consolidated information and dependable decision support. It has therefore been recognized by the community that efficient data management solutions are crucial to success of PHM. Such a solution should introduce a commonly accepted framework for data representation, data communication, and data storage. In other words, all solutions should be based on a commonly accepted and open standard in order to allow for seamless integration. In this writing we focus on the data challenge, i.e., the realization of a highly productive and standardized data management middleware

The organization MIMOSA is a "*non-profit [...] industry association, focused on enabling industry solutions leveraging supplier neutral, open standards, to establish an interoperable industrial ecosystem for Commercial Off The Shelf (COTS) solutions components provided by major industry suppliers*" (MIMOSA). The organization performs standardization work by defining reference architectures for PHM data management, respectively, aspects of PMH data management. We have chosen to base our data management backbone on two of MIMOSA's proposed standards, which are introduced in the following.

1.1. OSA-CBM

The Open System Architecture for Condition-based Maintenance (OSA-CBM) is an emerging reference architecture which has a chance of becoming the de facto standard for exchanging data in a condition monitoring system. Being an implementation of the ISO-13374 functional specification, the architecture defines six functional layers. Each layer is allocated different and unique functions of the data processing chain in a condition monitoring system (see Figure 1).

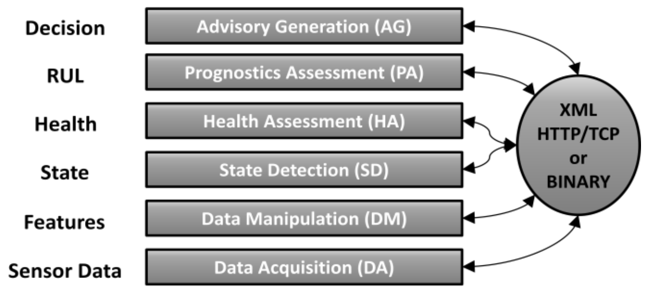

Figure 1.OSA-CBM Reference Architecture

This architecture focuses on the definition and communication of PHM data. Specifically, on the question as to which data entities and events can be exchanged between the layers during operation and the communication interfaces used for this purpose. The standard recommends the usage of XML messages, which are transported over HTTP, and for this purpose, a thorough collection of specifications for XML messages is provided. Recently, a binary transmission format for OSA-CBM messages has been added to the standard, and it is recommended to be used in embedded systems, or systems with limited computing resources (Löhr, Haines & Buderath 2012). In this writing, we will report about our experience in implementing the binary OSA-CBM format.

1.2. OSA-EAI

The reference architecture OSA-EAI is complementary to OSA-CBM and specifies comprehensive data storage architecture for asset management and configuration management systems. This architecture consists of: a physical relational data model (Common Relational Information Schema, CRIS), a corresponding logical object model (Common Conceptual Object Model), and CRUD interfaces (Create, Retrieve, Update, Delete) for all defined entities, as depicted in Figure 2. The data model is harmonized with OSA-CBM to facilitate storing data coming from all six OSA-CBM layers. Analogously to OSA-CBM, it is recommended that clients exchange XML messages transported via HTTP. For this purpose, the authors of the OSA-EAI standard provide a multitude of CRUD XML message specifications.

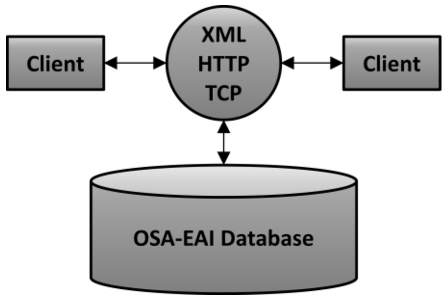

Figure 2. OSA-EAI Reference Architecture

The XML message specifications have been provided in XSD format. In this writing, we describe a Java code generator, which processes the XSD files and generates a fully functional client- and application tier upon the CRIS relational data model provided by MIMOSA.

2. ENVIRONMENT

Airbus Defence & Space is developing a comprehensive simulation framework for research in the areas of condition monitoring and prognostic health management. The framework includes airborne functions hosted on embedded systems, as well as ground-based functions hosted on PC-based systems. The primary objective is to interconnect both airborne and ground-based systems using a uniform data management philosophy and, as far as possible, uniform communication protocols. The simulation environment consists of airborne and ground-based functions which are connected by a data management backbone upon OSA-CBM and OSA-EAI.

In the following section, we provide a brief technical overview, whereas a more detailed description can be found in Löhr, Haines & Buderath, 2012. The air segment of the simulation framework models systems and associated sensors for which IVHM capabilities shall be developed. At the core of the framework is a central IVHM data processor to which data gets pushed by OSA-CBM. The IVHM data processor calculates IVHM information according to the OSA-CBM layer specifications, up to the health assessment layer (refer to Figure 3).

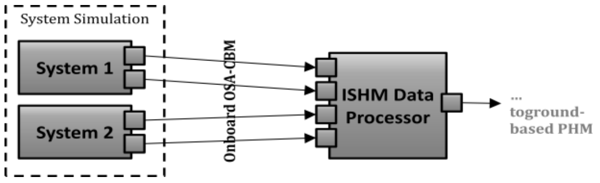

Figure 3. Air Segment of Simulation Framework

The central data processor supports download of data, which has been collected and calculated on board the aircraft, to the ground-based environment for further processing (e.g. during the aircraft's turnaround). Once downloaded, the data is stored in a central data management component, which we call the CBM data warehouse (refer to Figure 4).

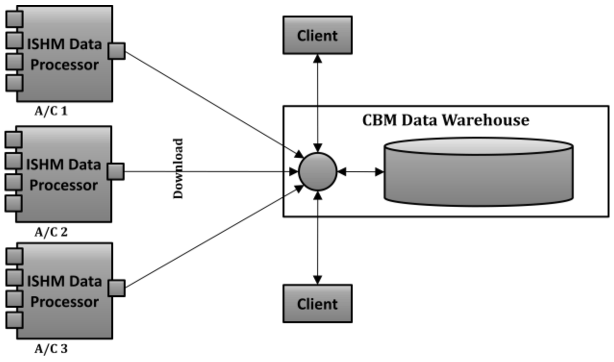

Figure 4. CBM Data Warehouse

The CBM data warehouse is based on the OSA-EAI reference architectures and it serves two major purposes: first, it hosts all current (i.e. short timeframe) and historical (i.e. long timeframe) condition data. Second, it provides services to distributed client applications that are involved in the PHM process.

In our context, data management includes the entire data set life cycle: from initial instantiation of a sensor value, transportation to the IVHM data processor, downloading to the ground-based environment, on through to storage and further processing.

3. OSA-CBM ENCODING IN THE AVIATION DOMAIN

When implementing OSA-CBM for an on-board embedded system one has to consider the software certification context for in-flight software. In this regard, our implementation deviates from MIMOSA's recommendation of transmitting OSA-CBM-encoded messages via a HTTP/TCP stack. Instead, we transport OSA-CBM messages via a UDP/IP stack. In our work we apply OSA-CBM messaging from data acquisition layer up to health assessment layer and in the following sections we report about our experience in implementing the binary OSA-CBM messaging standard in the C programming language under specific restrictions.

3.1. Programming Environment

When fielding OSA-CBM compliant applications on embedded systems certified for in-flight usage, several issues are brought to the fore. Ultimately, two aspects defined the unique structure of our solution: resource limitation and non-dynamism. Computing hardware for avionics, due to qualification requirements, are generations behind present off the shelf computing hardware. Implementation rules for applications hosted on real-time operating systems (such as VxWorks) typically forbid dynamically allocating memory resources, as these operations are potentially non-deterministic and lead to memory leaks if not used carefully. This environment imposes further constraints on the solution space: due to qualification or certification requirements (depending on the risk class of the final system) all embedded code must be written in the C programming language. Furthermore, UDP must be used as the sole protocol for network communication.

3.2. Starting Point

In order to make our current work comparable to prior work, we transmit the same OSA-CBM event instances as described in Löhr, Haines and Buderath 2012. This is, a *heavy load data event set* which contains four heterogeneous OSA-CBM `DMDataSeq` events at individual sample rates of 160Hz, 360Hz and 1 kHz (in total 2520 floats which corresponds to 10080 raw bytes). Additionally, we want to transmit a *light load data event set*, containing a single `DMDataSeq` event recorded at 20Hz (80 raw bytes). Both data event sets will be transmitted with a frequency of 1Hz. We have previously used these use cases to compare the standard XML-based OSA-CBM messaging protocol against a custom binary OSA-CBM messaging protocol, which we had designed at a point in time, where the standardized MIMOSA binary messaging protocol was not yet available to us. The ratio between transmitted data and usable payload, which shall act as a benchmark for the standardized binary messaging protocol, is given below.

	MIMOSA XML	Prop. Binary	Ratio
Heavy Load	165 345 bytes	40 792 bytes	4.1
Light Load	1 827 bytes	576 bytes	3.2

Table 1. Data Transmission Size Comparison

As seen in Table 1 there is a significant reduction in the volume of data from XML-based transmission compared to binary transmission, ranging up to a factor of four. Also, processing the messages is less costly in binary mode (Swearingen, Kajkowski, Bruggeman, Gilbertson & Dunsdon, 2007).

3.3. Previous Work

For the implementation of our use cases in standardized binary OSA-CBM, we expected no significant deviation from the ratio of transmitted data versus usable payload, as for our custom binary approach. We were however unsure if it would increase or decrease. Our custom binary format is not prepared for all optional or dynamic elements of an OSA-CBM message. We provided maximum boundaries for dynamic elements and implemented the subset of optional elements, which we need, as static fields. In contrast, the standard binary OSA-CBM format can deal with all optional and dynamic fields, but has to include metadata fields which control the interpretation of the byte stream.

In our previous work we modeled OSA-CBM data as structs and captured their memory footprint for direct transmission to the receiver side. An example can be found in Figure 5. We transmitted OSA-CBM messages from a 32-bit Ubuntu sender system on an Intel processor to a 32bit VxWorks receiver system on a PowerPC. In order to overcome the platform differences we worked with artificial padding bytes so that the internal in-memory arrangement of our transmission structs was equal on both platforms. Also, we performed byte-swapping on the receiving platform to deal with high- and little-endian issues. This allowed us to easily cast the UDP package payload into the required structures (including pointer remapping) with a minimum of marshalling and un-marshalling effort.

```
typedef struct {
    long id;
    Site site;
    OsacbmTime time;
    int alertStatus;
    OsacbmDataType_ENUM osaCbmDataType;
    int noEvents;
    char* events;
} DataEventSet;
```

Figure 5. Data Event Set as Custom C Structure

However, our approach was highly platform- as well as use case-dependent and did not cater for the full spectrum of OSA-CBM features. The standard OSA-CBM binary protocol is platform independent as it

- defines endianess
- introduces a limited set of primitive data types with specified width and defines signdness
- strictly serializes the OSA-CBM classes into a flat byte stream. Here, it benefits from the fact that no multiple inheritance is used in the OSA-CBM classes

3.4. Design and Implementation

The high level design of our implementation consists of the following three core parts:

- a representation of all OSA-CBM data types, Meta data elements, enumerations, constants, etc. as C language elements, such as enums, structs and defines. We modelled inheritance as already shown in Löhr, Haines & Buderath, 2012.
- an encoder library, which receives an instance of an OSA-CBM data event (struct instances) as input and transforms it into an OSA-CBM-compliant binary byte buffer
- a decoder library, which receives a binary byte buffer as input. It interprets the buffer form left to right, and instantiates and wires respective structs from left to right into an OSA-CBM compliant event structure

We have chosen to not include the actual network transmission layer into our implementation. It depends on the deployment of how the encoded bytes are actually transmitted or the encoded bytes are received. Our C code implementation is subject to the restrictions pointed out in section 3.1, according to which we do not have dynamic memory allocation available. We require that the caller of the decoder or encoder provides a chunk of (statically) allocated memory, on which the en- or decoder operations work. All structs will be allocated within this static piece of memory, of which the allocation we assume to be external to the library.

We model OSA-CBM data elements as C structs an enums, and have restricted the scope of implementation to OSA-CBM data event elements. Other elements, such as Configuration, can be implemented analogously. Having the user of our library modifying the data event struct and its children directly is possible, but error prone. Meta data fields could be missed, or the structure might simply be incomplete or incorrect. To avoid such errors, we provide a comprehensive set of wrapper functions for creation of events in the correct structure and for setting attributes on this structure. The creation functions operate on the pre-allocated static buffer. With these functions the user can create and populate an OSA-CBM data event set without having to deal with implementation details (such as pointer handling or OSA-CBM Meta data management). Also, the functions assure that the event is in a valid state at any time. An example will be given in the following.

1. `osacbmCreateNewDataEventSet`: provided with a chunk of statically allocated memory, the function will create an empty OSA-CBM data event and return a handle for further manipulation
2. `addDMRealToDataEventSet`: provided with a handle to an existing data event set, the function will add a new `DMReal` event to the given event set, hereby hiding all memory handling details. Also, the function returns a handle to the new

DMReal event for further manipulation (i.e., setting its value).

3. `addDMDataSeqToDataEventSet`: analogously, this function will add a new `DMDataSeq` event to a given data event set. Using the returned handle, the `DMDataSeq` can be populated
4. `addValueToDMDataSeqEvent`: given a handle to a `DMDataSeq` event, this function can populate the `DMDataSeq` event with a potentially infinite number of values
5. `setNumAlertsForDMDataEvent`: example for one or many functions which set specific attributes on a given event structure

Having constructed the required event structure as described above, the actual encoding is just one additional function call. Additionally to the actual data event that should be encoded, our encoder's entry function takes a pre-allocated buffer to which the resulting encoded byte stream shall be written. The encoder inspects the given struct and serializes it to the byte buffer. This approach is straight forward as it means implementing the pre-defined OSA-CBM specification.

The resulting binary and OSA-CBM compliant content can then be transmitted with any medium, such as UDP/Ethernet, a serial line or AFDX, to only name a few examples. On the receiver side, the decoding process is essentially the inverse of the encoding process. The function `osacbmDecodeOSACBMBinaryDataPacket` receives the transmitted bytes and a handle to a statically allocated working buffer. Additionally to the decoded data even struct, the function returns a handle to an object modeling the OSA-CBM message properties. Using our wrapper functions, the user can inspect the content of the just received data event set; for example, for passing the data into a state detection or health assessment layer.

3.5. Discussion of Results

As described in section 3.2, we transmit a data event set containing four heterogeneous OSA-CBM `DMDataSeq` events at individual sample rates of 160Hz, 360Hz and 1 kHz. The overall data event set has a frequency of 1Hz. The resulting data push represents 2,520 individual measurements being sent across the system every second. The second sample is a light load data event set, containing a single `DMDataSeq` event recorded at 20Hz; the corresponding overall data event set has a frequency of 1Hz. The heavy load event transmits 10080 raw bytes and the light low data set transmits 80 bytes.

	Prop. Bin.	Std. Bin.	Ratio gross/ net Std.	Delta Prop.
Heavy Load	40 792	16 972	1.7	58%
Light Load	576	568	7.2	-1%

Table 2. Performance of Binary OSA-CBM

The figures in Table 1 illustrate the performance of our binary OSA-CBM implementation. For the heavy load event a reduction of 58% of total raw bytes has been achieved. For the light load data event set the number of raw bytes increased by 1% -- obviously, Meta data has less impact for large payloads than for small payloads. We explain the significant reduction for the heavy load data event by the possibility to allocate dynamic data sections in the binary OSA-CBM protocol. Our custom implementation used fixed blocks with a maximum length for dynamic data fields (e.g., strings, arrays) and left unused space populated with initialization data, whereas in binary OSA-CBM the length as well as the actually transmitted number of bytes may vary.

4. OSA-EAI-COMPLIANT CBM DATA WAREHOUSE

The ground segment of our simulation framework includes a central repository for data and information, called *the CBM data warehouse*.

4.1. Motivation

Design of the CBM data warehouse was driven by the following high-level requirements.

1. act as a central information system
2. provide a uniform and standardized interfaces
3. maintain full traceability for in-service data

The MIMOSA reference architectures define a uniform data management philosophy that allows for full traceability of virtually any sensor value and its derived information. Earlier work (Gorinevsky, Smotrich, Mah, Srivastava, Keller & Felke, 2010, and others) demonstrated the feasibility of using these architectures as a reference to build a comprehensive information system for the aerospace domain. We consequently considered the selection of OSA-EAI and OSA-CBM as guidelines for the design of our CBM data warehouse as a promising approach to satisfy our high level requirements.

4.2. Previous Work

We have implemented a subset of the OSA-EAI standard for our initial version of the CBM data warehouse, as described in Löhr, Haines & Buderath, 2012. The subset was derived with the aim of providing data management for diagnostics and prognostics on our candidate systems. We concentrated on the ability to express system breakdowns (Assets, Segments, and Parent/Child relations) and the ability to associate data from the data acquisition, data manipulation, and state detection layers. Additionally, each asset was to have an active history of health assessments and remaining useful life estimates. We customized the utilized OSA-EAI tables in a way that would simplify the generation of test and reference data. We made further customizations to map specific features of the aerospace domain and stripped the composite primary keys of each entity down to a single dataset id, allowing us to strip down foreign keys as well. This approach was shown to be feasible by Mathew, Zhang, Zhang and Ma Lin (2006). Finally, we only considered those columns of any table which we really required. As a result, our CBM data warehouse was fully compatible to OSA-EAI, as it represented a subset of the standard. However, it was not compliant to OSA-EAI and we began work towards implementing the OSA-EAI standard to its full extent.

4.3. Approach and Architecture

The OSA-EAI standard defines a magnitude of documents and IT-specific artifacts of which the core artifacts – at least for our work – are briefly described here:

- CRIS: Common Relational Information Schema, a heavily normalized relational database schema. The standard provides CREATE statements for Oracle and other databases in the form of text files
- XML Request Specification: a set of XSD document type definitions which represent the entirety of XML-based requests that a client can send to an OSA-EAI compliant database. Also, the responses are defined.

The information sources above are the technical entry point for implementing an OSA-EAI database. Considering the proposed architecture from Figure 2 the implementation effort can therefore be summarized as follows. Instantiate the provided CREATE statements (porting the statements to the utilized RDMBS might be necessary). Create a server application which consists of a top layer listening for incoming XML messages via HTTP. The next layer inspects the parsed XML and routes the request to a more specific request processor. The request processor translates the XML content into an SQL statement (SELECT, INSERT or UPDATE) and executes the SQL statement against CRIS. Then, the result form the SQL statement, if any, is captured again by the request processor. If there is resulting data, the result set is worked off and the data it is wrapped into an XML document according to the XSD specification that corresponds to the initial request. The resulting XML response is finally serialized and appended to the output stream of the originating HTTP request – and as such received by the client.

The implementation of the server application cannot be done without significant effort by boldly implementing the partially very complex XML request for up to 300 XSD documents, which have to be mapped against up to 400 individual tables from CRIS. Instead of implementing the server application "by hand", we thought of a tool that would generate the source code for the server application from the available artifacts (CRIS CREATEs and XSD documents). The architecture of such a tool is depicted in Figure 6. The code generator receives all available XSD documents as well as the CRIS description as input, parses and analyzes them, and finally generates code for any layer of the described server application. Also, the code generator is able to generate unit tests for the server.

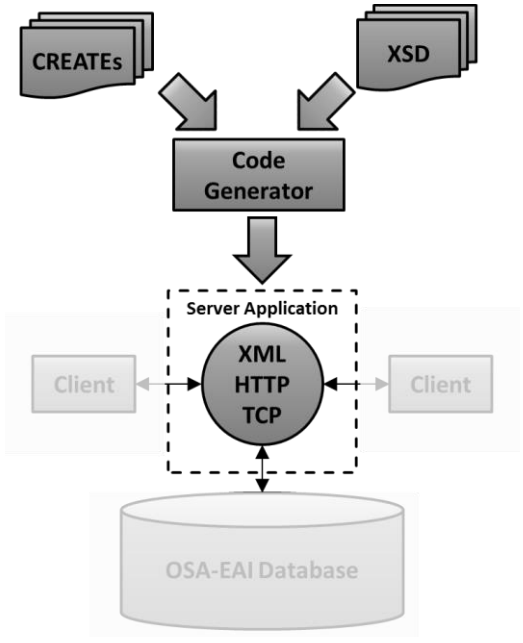

Figure 6. Code Generator for OSA-EAI

4.4. Realization

The OSA-EAI XML request specification is roughly grouped into the following three categories:

- Tech-Doc: facilitate data exchange between an application with information which it needs to publish periodically
- Tech-CDE: entity-centered, simple CRUD (create, update, retrieve and delete) operations
- Tech-XML: region-centered complex query, update, and create operations

For our work we initially focused on Tech-CDE and Tech-XML because our motivation was to create the required services for managing the information content in the database. For this purpose, CRUD operations only are required. We started with an analysis of the XSD documents provided in order to infer the required steps and architecture of the code generator. Both request groups provide a central XSD file which defines all XSD types referenced by the request definitions. In addition, Tech-CDE provides one file which contains all available retrieve requests (Tech-CDE Query) as well as one file which contains all available create, update, and delete requests (Tech-CDE Write). In contrast, Tech-XML provides a magnitude of files for each specific Tech-XML request. In total, there are 256 request files, of which the majority is Query definitions, followed by a few Create and Update definitions. The core difference between Tech-CDE and Tech-XML is that a Tech-CDE request is focused on one specific entity only. Relations to other entities are only considered by the respective foreign keys and entity-references are not fully resolved. It can therefore be seen as a relation-centric way of interacting with the database – just that one is not talking SQL, but XML. In contrast, Tech-XML defines requests which are based on a core entity upon which all provided filter parameters shall be applied. In addition to Tech-CDE the Tech-XML requests also resolve the entities which are referenced form the core entity. A response to a query request therefore not only contains the core entity's data, but also the resolved attributes of any referenced entity. As a result from this analysis we chose to focus on Tech-XML only, at least for the first iteration, since we believed Tech-CDE being a virtual subset of Tech-XML – at least from the perspective of what is required to talk to the underlying database. A significant result of our analysis was – as we hoped to confirm in the first place – that the request/response definitions all follow a common structure. This was the key prerequisite for designing the code generator. For the first iteration we made an important assumption: our aim was to only query or manipulate the core table that a specific Tech-XML is directed to. Although foreseen by the standard, we did not intend to resolve foreign key relations and thus we treated Tech-XML like Tech-CDE. We considered foreign key resolution as just an implementation effort.

Our code generator produces Java code: a server application which performs the XML request handling and the mapping of XML to SQL and vice versa, and a Java client library which provides an interface for client applications. The client library encapsulates the XML-messaging and leaves transparent that the client is actually talking to a remote database via the network. We will describe the four major phases of code generation:

Phase 1: Generate Model Classes

In this phase, the code generator parses the XSD files and generates Java POJO (plain old java objects) classes which correspond to the type hierarchy imposed by the XSD definitions. These classes do not implement any business logic and act as model objects for marshalling and un-marshalling XML or SQL result sets. For this task, we utilized the JAXB framework (refer to JAXB in the references section) which is an implementation of the Java API for XML Binding. The framework was able to create respective Java POJO code from the XSD files provided by MIMOSA. The generated model classes will both be utilized by the client library as well as the server application and thus act as common interface between the two parties. For example, from the XSD type `asset_healthTYPE` the class as depicted in Figure 7 will be generated (getters and setters have been omitted in the figure).

Phase 2: Generate Client and Server Interfaces

The interface that the client library exposes is not explicitly defined by the provided MIMOSA artifacts. We therefore chose to infer suitable method names from the request types as provided in the Tech-XML XSDs. For example, using the inherent substructure of the request element `mim_6002`, the interface method for this request type can be generated as follows:

```
public Mim6002Ack query(Mim6002Req query);
```

Phase 3: Generate Client/Server XML Transmission Code

In this phase the generator "implements" the client interface methods by generating code that serializes the given request object to an XML string, wraps the string into an HTTP POST request and opens the server URL. At this stage, the client instance can already perform request validation based on the multiplicity and optionality information declared in the XSD.

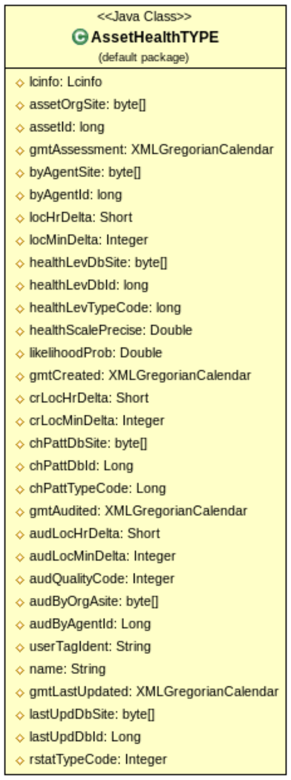

Figure 7. Example Generated POJO Class (getters and setters have been omitted for clarity)

For the server side, the generator creates code which receives the incoming HTTP POST request from a HTTP server socket, inspects the XML prefix in order to instantiate the correct XML parsing method (specific to the Tech-XML request type) for obtaining an object tree corresponding to the request. At this stage, the generator also produces code which serializes the object tree of the response into XML and streaming it to the output stream of the incoming request.

Phase 4: Generate Database Queries

The request object tree is inspected for the purpose of generating an SQL SELECT, INSERT or UPDATE statement depending on the content of the request. The type information of the objects themselves as well as Java annotations provided by JAXB provided enough information, to create syntactically and semantically valid SQL. For SELECT-type queries, the code generator produced code which inspects the result set from the database and – again, by utilizing the POJOs type and Meta information – which is able to translate the result set into a Tech-XML response object tree. As already explained in phase 3, this response object tree can then be serialized to XML and pushed to the requesting client.

4.5. Discussion of Results

The generation of POJOs was done by hooking up a single file into the JAXB framework. All dependent types and files were well referenced. We noticed however that the resulting class naming is somewhat odd – as already seen in Figure **7** all POJOs have a `*TYPE` postfix, which is introduced by JAXB. We found this issue to be of cosmetic nature only. The generation of interface method names seemed straight forward at first. The multitude of method names however made the client interface rather confusing as the request number (e.g., 6002) has to be mapped mentally to what the request is actually doing (e.g., asset health).

Compared to Tech-CDE it is not clear how the current catalog of Tech-XML requests was motivated. During the utilization of our generated application we were missing several request types for accessing specific areas of the OSA-EAI database model, such as the steps of solution packages. We were able to work around the missing requests by adding to the catalog following the already existing philosophy. We do not see this as a significant drawback of the OSA-EAI standard as the requests which are missing in Tech-XML can be found in Tech-CDE. It is however a strong indication that a productive application should implement both Tech-XML and Tech-CDE requests. We encourage the standardization committee to include the possibility of defining customer Tech-XML requests on the basis of standardized XSD request specifications.

Tech-XML requests provide support for equality filtering (‘=’) on the attributes on an entity. It is also possible to get the N latest (chronologically) instances of an entity, which is necessary for PHM applications, e.g., the latest asset health assessment of a specific asset. What’s missing is support for filtering beyond attribute equality. We were missing the general possibility for range filtering (left-, right- and left-right-bounded) on numeric or date attributes (range filtering is possible for specific date attributes), and filtering by regular expressions on character attributes, or at least wildcards. The latter is a matter of interpreting the already existing search criteria on the server-server side, and does not require structural modification, but explicit conventions of how to populate the search criteria. The former can be realized by enhancing Tech-XML XSDs to include optional left and right bounding attributes for each Tech-XML search criteria. A negative filter (“get all entities which do not match”) is missing, but can be integrated analogously.

We were also missing the possibility for grouping, or ranking, and aggregation within a single query. Our application requires retrieving the latest health and RUL for each asset and both information types are stored as time series per asset in respective tables. It is possible to write a single SQL query on table `asset_health` which returns the latest health grade per asset. In Tech-XML this is currently not possible, and one has to make one Tech-XML query per asset. Again, a solution to this issue is the enhancement of the Tech-XML query XML to include indicators on a search attribute, whether the result should be grouped or ranked by this attribute, and which aggregation functions should be used on the result columns.

The “core” table of a Tech-XML request, i.e., the entry point into the data, is not highlighted in the request specification and cannot be uniquely inferred from the list of parameters. However, for the majority of requests, the first entity of the response specification corresponds to the core table that shall be queried. Here, we suggest a more explicit way of specifying the request parameters, i.e., which entity should actually be queried.

For SQL code generation, the mapping between entity names to table names as well as entity attribute names to column names is not 1:1. For our generator we exploited the fact that both XML elements and database elements followed a specific naming schema that could be used for a bidirectional mapping – however, the naming schema is not documented, thus, subject to be changed (accidentally). We suggest to standardize a bidirectional mapping function (e.g., the current pattern) for entity/table and attribute/column names. We found that for the sake of request validation the consideration of the CRIS CREATE statements was needed for synchronizing multiplicity information from the XSD with primary and foreign key definitions on the database level.

4.6. Summary & Outlook

Assuming that any graphical user interface will take the role of a client application we have shown that a fully functional OSA-EAI information system for Tech-XML requests can be generated from the provided MIMOSA documentation as is. The compiled system is able to run the entire request-response cycle, starting from the assembly of the request on the client side, transmitting the request, issuing SQL against the CRIS database and sending back the results. Our implementation does not yet resolve foreign key relations but provides the foreign keys themselves for later referral. Since Tech-XML provides a different focus than Tech-CDE we conclude that a productive application must provide both Tech-XML and Tech-CDE interfaces to provide access to the full information content. Tech-CDE provides CRUD access for every entity and can therefore be considered as the “Swiss Army Knife” for OSA-EAI interaction. Tech-XML provides convenience functions and the ability to resolve dependent entities in a single request. In further work we will extend our code generator to Tech-CDE request types. Given the more complex nature of Tech-XML, we do not expect any significant issues for this endeavor.

5. CONCLUSION

We presented our experience from the realization of our next generation data management backbone for a simulation framework for PHM systems in the aerospace domain. For the airborne segment OSA-CBM-based communication was chosen. From previous work, where we evaluated XML-based transmission, we were motivated to use binary transmission and defined a custom protocol. Recognizing the drawbacks of our approach we switched to the new available binary transmission standard of OSA-CBM 3.3.1. We have shown that the standard can be implemented in the C programming language under the restrictions of airborne software development. Furthermore, for this special environment, we have suggested a layered approach which provides simple creation as well as manipulation functions for OSA-CBM data, which hide the details of the underlying implementation. The ratio of transmitted event size to usable payload is about 25% of the XML-based approach (overhead for HTTP and TCP not included).

The ground-based part of our data management backbone is centered on an information system, which we call the CBM data warehouse. It is designed compatible to the OSA-EAI reference architecture. Confirming the feasibility of OSA-EAI by a prototype implementation of a stripped-down instance of OSA-EAI in previous work we describe here our experience from realizing a Java code generator for a fully functional OSA-EAI client-server application system. We could successfully show that the MIMOSA-provided artifacts provide enough suitable information to generate executable code in an automatic way. We further found that Tech-CDE and Tech-XML should be both implemented in a productive information as both request categories cover different aspects (however, they also have common areas). During the implementation of our code generator we found several issues regarding object naming and object mapping which we do not consider critical. We found that the request specification lacks comprehensive support for extended filtering and aggregation when assembling a request. By providing such support in a standardized way the response times and the network traffic could be reduced significantly.

REFERENCES

Dunsdon, J. & Harrington, M. (2008). The Application of Open System Architecture for Condition Based Maintenance to Complete IVHM. *IEEE Aerospace Conference*, March

Gorinevsky, D., Smotrich, A., Mah, R., Srivastava A., Keller, K., &Felke, T. (2010). Open Architecture for Integrated Vehicle Health Management. *AAIA Infotech@Aerospace Conference*, April20-22

JAXB. JAXB Project Website. *https://jaxb.java.net/*

Löhr, A., Haines, C., & Buderath, M. (2012). Data Management Backbone for Embedded and PC-based Systems Using OSA-CBM and OSA-EAI. *AAIA Infotech@Aerospace Conference*, April20-22

Mathew, A. D., &Ma, L. (2007). Multidimensional schemas for engineering asset management. *Proceedings World Congress on Engineering Asset Management*, Harrogate, England

Mathew, A. D., Zhang, L., Zhang, S., & Ma Lin (2006).A review of the MIMOSA OSA-EAI database for condition monitoring systems. *Proceedings World Congress on Engineering Asset Management*, Gold Coast, Australia

MIMOSA. Mimosa Organization Website. *http://www.mimosa.org*

Swearingen, K., Kajkowski, W., Bruggeman, B., Gilbertson, D., &Dunsdon, J. (2007). Multidimensional schemas for engineering asset management. *Proceedings IEEE Aerospace Conference*

BIOGRAPHIES

Matthias Buderath Aeronautical Engineer with more than 25 years of experience in structural design, system engineering and product- and service support. Main expertise and competence is related to system integrity management, service solution architecture and integrated system health monitoring and management. Today he is head of technology development at Cassidian. He is member of international Working Groups covering Through Life Cycle Management, Integrated System Health Management and Structural Health Management. He has published more than 50 papers in the field of Structural Health Management, Integrated Health Monitoring and Management, Structural Integrity Programme Management and Maintenance- and Fleet Information Management Systems.

Conor Haines received his B.Sc. degree in Aerospace Engineering from Virginia Polytechnic Institute and State University in 2003 and his M.Sc. degree in Computational Science from the Technical University of Munich in 2011. For 3 years Conor was a test engineer supporting the NASA Near Earth Network, providing simulation support used to guide system development. At his current post, he is focused on developing IVHM and Computer Vision technologies as a Software Engineer for Linova Software GmbH.

Andreas Löhr received his M.Sc. degree in Computer Science from the Technical University of Munich in 2001 (Informatics, Diplom) and earned his PhD degree in Computer Science from Technical University of Munich in 2006. For 6 years he worked as a software engineer at Inmedius Europa GmbH in the area of interactive technical publications and researched in the field of wearable computing. He founded Linova Software GmbH in 2008 and at his current post as managing director he focuses on development of maintenance information systems and data management architectures.