Implementing MIMOSA Standards

Johannes Drever[1], Helmut Naughton[2], Michael Nagel[3], Andreas Löhr[4] and Matthias Buderath[5]

[1,2,3,4] *Linova Software GmbH, München, 80805, Germany*
johannes.drever@linova.de
helmut.naughton@linova.de
michael.nagel@linova.de
andreas.loehr@linova.de

[5] *Airbus Defence and Space Deutschland GmbH, Manching, 85077, Germany*
matthias.buderath@airbus.com

ABSTRACT

A common challenge to Prognostic Health Management (PHM) systems is the management of data across different organizations based on a standardized format and meaning. The Open System Architecture for Condition-based Maintenance (OSA-CBM) and the Open System Architecture for Enterprise Application Integration (OSA-EAI) are complementary reference architectures for domain-independent asset and condition data management. In previous papers, we reported on our experiences with implementing a data integration layer based on these two architectures. In this paper, we report on our experience implementing code generators for binary OSA-CBM and OSA-EAI Tech-CDE (Compound Document Exchange), and the utilization of the resulting components within the OMAHA project. OMAHA aims towards an overall management architecture for health analysis, incorporating manufacturers, operators and maintainers of fleets of aircraft. The OSA-CBM standard specifies a message structure but leaves the assembly and disassembly of OSA-CBM data up to the implementor. Our solution is a builder/reader Application Programming Interface (API) for a binary OSA-CBM message codec which we have implemented under the constraints of a real-time computing environment. The required C code is automatically generated from the provided technical documentation for OSA-CBM. We discuss the properties of the resulting codec and point out future improvements for the OSA-CBM binary protocol to improve consistency and to add the capability of streaming. Using the same generative approach we have implemented a code generator for a Tech-CDE-compliant middleware system, consisting of client libraries (currently C++ and Java), a network layer, a server portion, and a database backend. Analogously to OSA-CBM, the code generator processes the documentation provided for Tech-CDE, creating both productive and testing code. We discuss the properties of the resulting system, report specific limitations of the Tech-CDE protocol and suggest mitigations. The paper concludes with an experience report from utilizing our work in the OMAHA project. While Tech-CDE was generally found sufficient, we identified areas of improvement, including protocol properties and entity coverage. We were able to make customizations using our generative coding approach and present these as suggestions for future standard extensions.

1. INTRODUCTION - MIMOSA STANDARDS

Introducing PHM systems imposes a paradigm shift from prevention of failure towards prediction of failure. Besides the challenge of creating the enabling diagnostics and prognostics technologies, the management of data is crucial to the successful application of PHM. What is required is a commonly accepted standard for data representation, data communication, and data storage across different organizations and stakeholders. In our work we focus on data management middleware based on MIMOSA[1] (MIMOSA, n.d.), an organization which performs standardization work by defining reference architectures for specific aspects of PHM-related data management.

1.1. The OSA-CBM standard

The Open System Architecture for Condition-based Maintenance (OSA-CBM) is an implementation of the ISO-13374 functional specification and defines six layers. Each layer hosts different functions of a data processing chain (see Figure 1). The standard focuses on the definition and communication of PHM data in distributed systems, including the monitored devices themselves. Since the inclusion of a bi-

[1]MIMOSA: Machinery Information Management Open System Alliance

nary transmission format OSA-CBM is suited for being used on embedded systems (Löhr & Buderath, 2014).

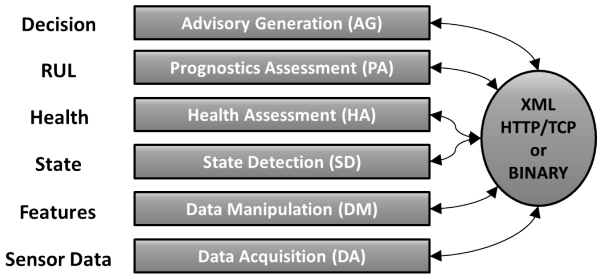

Figure 1. OSA-CBM Overview.

1.2. The OSA-EAI standard

The reference architecture OSA-EAI complements OSA-CBM with a comprehensive data storage architecture for asset and configuration management. The architecture includes a physical relational data model called Common Relational Information Schema (CRIS) and an XML-based messaging format called Tech-CDE (Compound Document Exchange). Via the XML messages, a remote client is able to perform Create, Retrieve, Update and Delete (CRUD) operations for all entities defined in CRIS, provided there exists a bidirectional mapping between Tech-CDE XML and SQL.

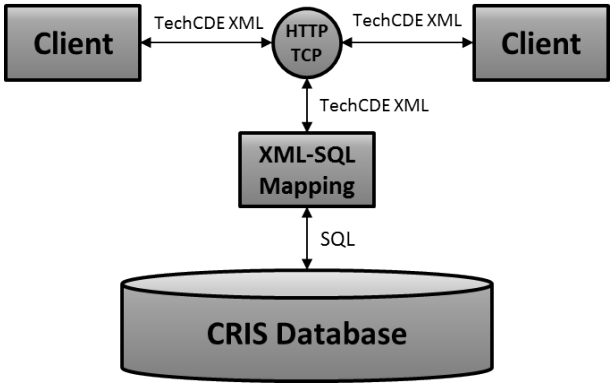

Figure 2. OSA-EAI Tech-CDE Overview.

2. ENVIRONMENT

The German national research project OMAHA (Overall Management Architecture For Health Analysis) is a joint effort of aircraft manufacturers, airlines, aviation industry suppliers and aviation-related research disciplines, such as artificial intelligence. Motivated by the upcoming increase in air traffic over the next decade, the project seeks to optimize operational cost by increasing aircraft availability and by reducing maintenance expenses. A key enabler for this endeavor is the introduction of system-wide diagnostics and prognostics on the basis of physical or virtual sensor data. The OMAHA project specifically focuses on the required system architecture, which shall facilitate data exchange between OEMs[2], operators and MROs[3], as well as the necessary process- and tool chains which have to be deployed. The project partners have decided to evaluate and use the MIMOSA Standards (see section 1) for their purpose, and Linova contributes to the project by providing required middleware stacks and data modeling skills.

3. OSA-CBM IMPLEMENTATION

In this section we report about our experience from implementing the OSA-CBM binary protocol in the context of the OMAHA project.

3.1. Motivation and Previous Work

From previous work we have gathered experience with implementing the OSA-CBM protocol XML-based (Löhr, Haines, & Buderath, 2012) and binary (Löhr & Buderath, 2014), and continued our work towards OSA-CBM for embedded systems based on the binary transmission spec. Our previous implementations showed the general feasibility of binary OSA-CBM but incorporated coding constructs which were cumbersome to use for developers, and lacked completeness. In this paper we present a generative approach for implementing the full OSA-CBM binary spec and providing a simple to use API.

3.2. OSA-CBM Message Builder API

We encountered two major issues while designing the message builder API. The first issue is that our target environment requires C, which prohibits using object-oriented programming to represent messages and subtypes. Therefore, we had to build the API as a large flat set of functions, which we arranged into groups using name prefixes. The second issue was that the target environment forbids dynamic memory allocation. Thus, we perform all encoding and decoding operations on a fixed-size buffer that resides either on the stack or is statically allocated.

Our API for the OSA-CBM subtypes follows a common pattern. For each subtype, there is a function that starts encoding the subtype, and one that marks the subtype as finished. There are simple setter functions for primitive attributes (numbers, strings, etc.). For subtype attributes, we provide functions to enter and leave the respective subtype attribute, which frame the start and finish calls for the respective subtype. Arrays require yet more functions in order to correctly handle the array counter. The API automatically enforces that subtype attributes are set in the correct order as defined by the subtype specification. The API user only needs to state in ad-

[2]OEM: Original Equipment Manufacturer
[3]MRO: Maintenance, Repair and Overhaul

vance if the subtype length is guaranteed to stay below 255 bytes[4]. The API manages aspects of the subtype header automatically, including length field and optionality bit mask.

The following listing encodes a minimal DMReal subtype (mandatory attributes only) using our API.

```
1 OSACBM_encoder_DMReal_start(encoder,
      true);
2 OSACBM_encoder_DMReal_id_set(encoder,
      0xAFFF);
3 OSACBM_encoder_DMReal_value_set(encoder,
      354.356897);
4 OSACBM_encoder_DMReal_finish(encoder);
```

3.3. Code Generator

Our OSA-CBM message builder API consists of two parts: a manually built set of functions that manage the overall message structure, header, information blocks etc., and an automatically generated part that encodes all currently defined subtypes. The binary encoding specification package contains three machine-readable files that define subtypes and attributes (all.members), inheritance relations between subtypes (all.inheritance), and enums (all.enums). Our API generator reads and processes these files, builds an internal model of the specification, and flags inconsistencies such as missing referenced subtypes, unexpected primitives, missing referenced enums, cyclical structural dependencies etc.
In the next step, the generator writes one .h and one .c file for each subtype and enum that implement the API described above. All aspects are generated automatically, including member order enforcement, optionality bit handling, packing group management etc. The generated code performs extensive error checking at every step to ensure the encoded message is consistent and complete. We also generate test suites for all subtypes along with JSON encoders/decoders to allow manual test case specification.

3.4. Lessons Learned

During our work on the message builder API, we noticed an inconsistency in the spirit of the specification regarding length fields. Subtype headers are required to specify the length of their subtype block, while the overall message blocks are not. The length of a string is impossible to determine before actually processing it, whereas the length of an array must be written before the array itself.
There are certainly arguments for both styles: knowing all lengths in advance makes it easier for recipients to allocate memory and perform checks on the incoming message; allowing lengths to be unspecified in advance eliminates a considerable part of housekeeping duties and may significantly lower memory requirements on the encoder side. If lengths do not need to be specified in advance, a memory- and time-constrained message sender can stream messages instead of fully assembling them in memory prior to sending. For example, a sender that creates long arrays of numeric value subtypes currently needs to hold the entire message in memory to continuously update the length field while members are added. If the length were not required in advance, the sender could simply send out array element after array element and eventually finish the message, without having to keep anything in memory in the meantime.
Therefore, we recommend that a future version of the specification consistently allows both specified and unspecified lengths for all mentioned instances (headers, strings, arrays). For example, a length value of 0 (zero) for a subtype length could indicate that the length of the subtype block must be calculated from its expected content as per the specification. Strings could be prefixed with a 1/4 byte length indicator before the actual string content, where a length value of zero means that the string is null-terminated.
Getting arrays to not require a length in advance is a bit more complex, but also manageable. A straightforward option would be to require a 1-byte indicator before each array element for arrays with unspecified length. This indicator could be 1 if another array element follows, and 0 if the end of the array has been reached. The downside here is the added space consumption. This can be addressed through a slightly more complex scheme: an array end marker byte (e.g. 0xFF). If the array end marker byte is encountered where a new array element would begin, it marks the end of the array instead. Of course, it must still be possible for an array element to begin with the actual value of the array end marker. An array element marker byte (e.g. 0xFE) can be used for that. If an array element starts with either the end marker value or the element marker value, an element marker value would be written, and the actual content afterwards. For example, an element starting with 0xFF would be encoded as 0xFE 0xFF, and an element starting with 0xFE would be encoded as 0xFE 0xFE. All other values in the first array element byte (here: 0x00-0xFC) do not need special treatment and would be handled as the beginning of an array element.
Another issue that we encountered is that there is no hard limit for the depth of nested subtype trees. It is theoretically possible to assemble a subtype tree of arbitrary depth, because there are many possible cyclical structural dependencies between subtypes. The shortest example is that a Data subtype contains an array of Data subtypes, which means it is possible to nest Data subtypes to any depth. In total, our generator currently detects 758 unique circular structural dependencies, with circle lengths from 1 (Data → Data) up to 9 (e.g. ExplanationDataSet → ExplanationData → DataEventSet → PADataEvent → AmbiguityGroup → OrConnector → NotConnector → AndConnector → LogicalConnector → ExplanationDataSet). If there were no circular structural dependencies, there would be a maximum depth, and our encoder

[4]OSA-CBM defines a different header for subtypes greater than 255 bytes.

could allocate a fixed depth subtype management structure stack that cannot be exceeded.

4. TECH-CDE IMPLEMENTATION

In this section we report about our experience from implementing the OSA-EAI Tech-CDE protocol in the context of the OMAHA project.

4.1. Motivation and Previous Work

A core goal of the OMAHA project is to build a demonstrator for physical health management in a real time closed loop with scheduling algorithms. The demonstrator simulates a fleet of passenger aircraft based on scheduling data provided by a major airline. The aim is to demonstrate that the tight loop between real time data monitoring and maintenance scheduling leads to more efficient planning. There are several parties responsible for the simulation of the fleet, the physical simulation of aircraft parts, and the scheduling of maintenance activity. These parties work with heterogeneous data formats in different domains. The OSA-EAI layer is introduced in order to unify this data in the Common Relational Information Schema (CRIS). Further, the Tech-CDE network layer is provided to allow data manipulation in CRIS. The specific domains of the participating parties, in particular the flight and maintenance scheduling, are mapped to the generic CRIS schema (4.6). We extended the CRIS schema since it was not possible to map all scheduling related entities (4.7).

In previous work we have presented an OSA-EAI Client/Server Application implementation (Löhr et al., 2012; Löhr & Buderath, 2014). The choice of the Tech-XML format brought several draw-backs. The identification of Tech-XML request types with numeric values (e.g. `mim_5005_req` for asset) imposed additional mental burden on the developer and led to difficulties in readability and maintainability in the resulting client code. Further, Tech-XML does not provide range queries, grouping, ranking and aggregation queries (Löhr & Buderath, 2014). The Tech-CDE Client/Server Application allows using the CRIS schema for data exchange in a more direct fashion. There are 431 query and write requests for the 431 entities specified in the CRIS schema. We hoped this to be more suitable than the Tech-XML architecture for the data integration use case in OMAHA.

The Tech-XML implementation was realized by a generative approach using the XSD specification. The specification was parsed with JAXB (JAXB, n.d.) in order to generate object representations and XML-serialization methods. This approach limited the client to Java and only allowed a 1:1 mapping of protocol elements to Java objects. In the current work we use a more flexible approach to code generation. First, since there is a one to one correspondence between CRIS instances and Tech-CDE read/write requests, we used the SQL DDL specification of CRIS as the basis for code generation - not the also provided XSDs. Second, we implemented a whole SQL DDL analysis and Java/C++ code generation framework (4.5). The code generator allows for more flexible design choices and the implementation of more complex features such as partial updates in the client API (4.3). The generated code has entity-specific methods for XML-serialization. Thus, run time reflection as in the JAXB implementation is not necessary and may lead to better runtime performance.

4.2. System architecture

The extended CRIS schema is hosted as a MySQL database. The database is accessible via a server which handles Tech-CDE read and write requests. The server may be queried either by standard Tech-CDE messages (cf. 4.9 for limitations) or via a Java or C++ client API. The server can be accessed via HTTP/S, optionally enforcing authentication via SSL client certificates. The client API provides factory methods for a convenient configuration of the HTTP/S connection using an optional proxy server.

According to the MIMOSA specification, the Tech-CDE client and server schema was developed as a means of transferring aggregate, related sets of CRIS data in XML format using one query schema and one write schema. It is independent of specific "connect" or "disconnect" methods between client and server. The server may combine data from different databases.

4.3. The client side API

The Tech-CDE Client API allows to issue Tech-CDE queries from either Java or C++[5]. The API provides data access object (DAO) interfaces for each entity, as shown in figure 3. The DAO interfaces provide CRUD operations for the corresponding entity. The provided CRUD operations can be used with several different parameters, depending on whether a single object or a bulk of objects is manipulated. Single object manipulations are specified with primary keys, object instances or template instances. Bulk manipulations are specified with lists or entity specific *Filters* and *BaseParams*. The following example illustrates how a list of assets is queried.

```
1 DaoAsset dao = factory.getDaoAsset();
2 List<Asset> assets = dao.getAssets(
3   Arrays.asList(
4     new AssetFilter()
5       .name(Comparator.LIKE, "aircraft")
6       .criticality(Comparator.MIN, 3L)),
7     new AssetBaseParam()
8   .orderBy(AssetAttribute.ASSET_ORG_SITE)
9   .order(Order.ASC));
```

[5] In this section only the Java implementation is presented. The C++ implementation provides analogous concepts with a similar implementation.

The list of assets is defined by a filter on the name and on the criticality of assets, resulting in all assets whose names contain "aircraft" and which have a minimal criticality of 3. The result set is ordered ascending by the `asset_org_site`. The API is designed using method chaining (Fowler & Parsons, 2010) to facilitate the combinatory build of comprehensive queries. Further properties may be added to the query by appending further method calls. The example API call results in the following Tech-CDE request:

```
<mim_query>
 <mim_query_req>
  <header include_non_active_rows="0"
      session_id="2016-03-30T11:50:17"/>
  <base>
   <base_params count_only="false"
       order="ASC"
       order_by="asset_org_site"/>
   <base_rows>
    <asset>
     <filterTYPE column_name="name"
        like_value="OMAHA"/>
     <filterTYPE
        column_name="criticality"
        min_value="3"/>
    </asset>
   </base_rows>
  </base>
 </mim_query_req>
</mim_query>
```

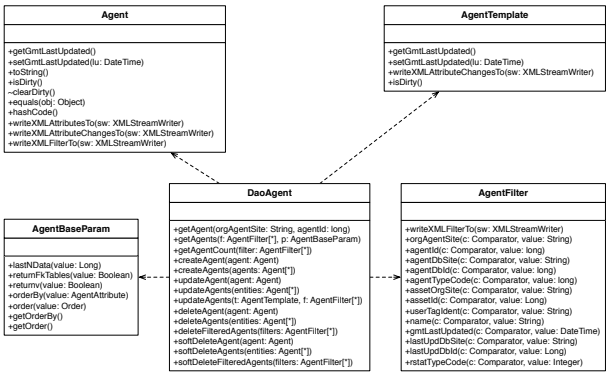

Figure 3. The client side API.

Handling of updates to existing entities

In order to let the client library user interact with the entity objects as they would naturally, we track for each attribute of the entity object whether it has been changed on the client since it was retrieved from the server. This way, if the user wants to transmit the updated entity to the server, we can send only those attributes which actually need updating, minimizing the risk of lost updates, which might have happened in the meantime on the server via another client.

Handling of updates with filtering

To allow for updates to all entities corresponding to specific filter condition, we need to be able to specify the attributes to be updated, and their values. The entity classes themselves cannot be used for this purpose, since they differentiate between mandatory and optional attributes of the entity and thus require values for all mandatory attributes.

4.4. Generic Tech-CDE server

Providing a client library that allows to interact with an object representation of CRIS entities requires a code generation approach such as the one described in 4.5 to generate classes and helpers for each entity.

The server can be structured less complex. Since the XML representation of objects in the Tech-CDE directly map to CRIS tables and their attributes, we have implemented a generic mechanism for translating Tech-CDE XML requests directly into SQL statements.

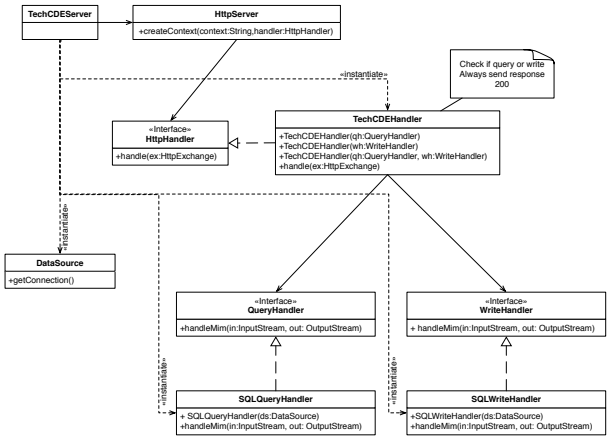

Figure 4. Tech-CDE Server.

The high-level structure of our Tech-CDE server implementation is shown in Figure 4. The server communicates with the clients via HTTP/S and implements separate handlers for query and write requests. The Tech-CDE Client/Server Version Application Specification leaves network layers[6] 5 and below up to the implementor. We have chosen to use HTTP POST requests for querying the server and transmitting the response back to the client, as most corporate networks allow for this traffic in and out of their perimeter.

We use two separate handlers for query and write, because the Tech-CDE Client/Server Version Application Specification allows for servers to implement these features separately, e.g. a server could only allow for querying but not for writing data. The Tech-CDE handler is used to read the first 100 bytes

[6]ISO-OSI Network Model

of the incoming request stream and determine the Handler to be used to process the request.

Requests and responses are both handled as streams, so even for large XML files (e.g. requests inserting many objects at once into the database, or responses containing a large number of objects) the server does not keep the entire object hierarchy in memory.

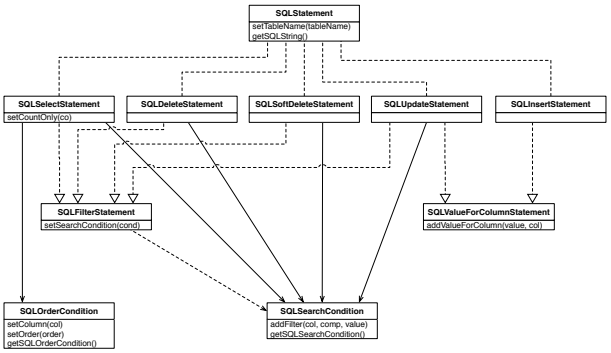

Figure 5. SQL statements.

Figure 5 shows how SQL statements are represented. There are five classes of SQL statements, corresponding to SELECT, INSERT, UPDATE, and DELETE, as well as a SOFT_DELETE, which just updates a specific attribute of an entity to mark it as not being in use. Search conditions (WHERE clauses) can be specified for SELECT, (SOFT_)DELETE, and UPDATE. INSERT and UPDATE can set values for specific columns in the entity's table. SELECT can specify an ordering for the results with an ORDER BY clause.

The query and write handlers each instantiate the appropriate SQL statement class, and populate the order and search conditions as necessary. After the end of the request is reached, the SQL statement class is used to create the SQL query string. The results are directly written into the HTTP response stream.

Consider the following sample DELETE request from the Tech-CDE Client Server Application Primer (adapted for this paper by reducing the number of filters to two):

```
<mim_write
  xmlns="http://www.mimosa.org/TechCDEV3-2">
<mim_write_req>
<header session_id = "36" />
<param action="Hard_Delete"
  trans_class="Atomic" />
<mimosa_rows>
<segment_chr_data>
<filterTYPE column_name="segment_site"
  equal_value="000003F900000001" />
<filterTYPE column_name="segment_id"
  equal_value="21" />
</segment_chr_data>
</mimosa_rows>
</mim_write_req>
</mim_write>
```

The server parses the XML as it is streamed to the server. On encountering the `mim_write_req` tag, the stream is forwarded to the write handler to handle the rest of the query. The `param` tag specifies the type of write – in this example the `action` is DELETE, so an `SQLDeleteStatement` is created. Since the `trans_class` is "Atomic", the write handler is set to execute the statements atomically, i.e. as a transaction. Next, the parser encounters `mimosa_rows` which signals that the next opening tag defines the name of the table to be operated on, in our example "segment_chr_data". This tag also contains a list of `filterTYPE` tags. Each such tag is parsed and added to the delete statement's `SQLSearchCondition` via the `addFilter` method, e.g. for the first filter:

```
addFilter(Comparator.EQUAL,
  "segment_site", "000003F900000001");
```

The search conditions form a `WHERE` clause, with additional conditions added with `AND`. The closing tags signal the end of the statement. The `SQLDeleteStatement` now contains all information needed to create a valid SQL statement performing the delete: the table name is "segment_chr_data", the first part of the where clause is a check for equality of the "segment_site" column and the value "000003F900000001", and the second part of the where clause is a check for equality of the "segment_id" column and the value "21". The resulting SQL statement is constructed as follows:

```
DELETE FROM segment_chr_data
  WHERE segment_site =
      '000003F900000001'
  AND segment_id = '21';
```

In order to prevent clients from accessing arbitrary, non-CRIS tables, we generate a whitelist from the CRIS model which contains valid targets for any read or write query. This whitelist is the only piece of server code which contains specific information about CRIS entities, the rest is generic.

4.5. New approach to code generation

The CRIS model, and consequentially the Tech-CDE protocol, provides 431 different entities. These entities may be subject to future change or may be extended - for example with the scheduling extension described in 4.7. In order to facilitate the tedious work of implementing the communication infrastructure for the large amount of entities and in order to be responsive to further change we chose a generative approach. The generative approach requires parsing of source data and generation of source code in possibly multiple target languages. We chose Haskell to implement the code generator because it has a rich ecosystem of libraries for parsing and

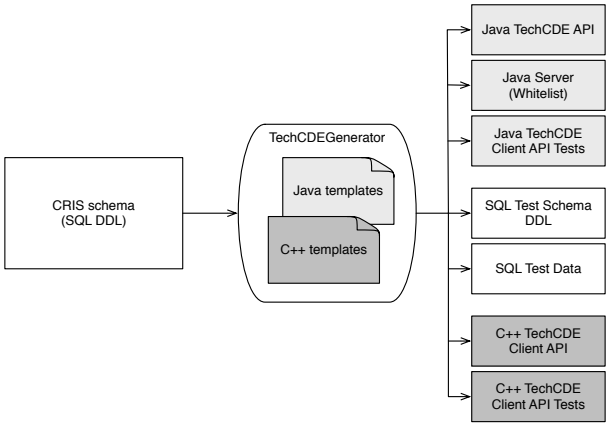

Figure 6. Tech-CDE code generator.

pretty printing (Gonzalez, 2015).

The overall architecture of the code generation process is shown in figure 6. The dynamic source code[7] is provided as templates and compiled with the source code generator. The source code generator parses the CRIS database schema and transforms it into an intermediate representation. From the intermediate representation the Java code, the SQL code and the C++ code are generated. The Java code comprises the Tech-CDE API, the server whitelist and the test code. The SQL code comprises the test schema and the test data. The C++ code comprises the Tech-CDE API and the test cases.

Parsing input data

The CRIS database schema is provided in the form of a SQL DDL, i.e. as *CREATE TABLE* statements. The CRIS schema is published by MIMOSA with the OSA-EAI in ORACLE format. The parsing library[8] is agnostic to the SQL format, since the main differences are related to the data types. We map the ORACLE data types to internal data types. Additionally we map MySQL types to the internal data types. Thus, it is possible to convert ORACLE to MySQL entities and vice versa.

Source code generation

Most of the Java and C++ code is static code. The remaining dynamic code is represented in the form of templates. The templates are actually Haskell code which is compiled as part of the generator. The Java code is represented as an abstract syntax tree (AST) which ensures that only syntactically correct Java code is generated. This eliminates errors in the Java code during the compilation phase of the generator. However, it is still possible to generate Java code with compile errors, e.g. by using variables which are not in scope. The representation of the Java code as an AST provides additional benefits since it is possible to manipulate this data structure. The generator adds import statements to the source code, similar to Eclipse's "Organize Imports" feature.

[7] We refer to code which is generated as dynamic code. Code which is stored in static files is called static code.

[8] HsSqlPpp: https://hackage.haskell.org/package/hssqlppp

4.6. Mapping of OMAHA data model to CRIS

In order to ensure interoperability with existing domain models, we had to map OMAHA domain entities to their corresponding CRIS entities. For this purpose, we created tooling to ingest the domain model and the CRIS model both in the form of SQL create statements. We then let domain experts map domain entities to their closest corresponding CRIS entity, and the domain entity's attributes to the CRIS entity's attributes. Any attributes not mappable to CRIS columns were mapped via the CRIS data attachment mechanism shown in Figure 7.

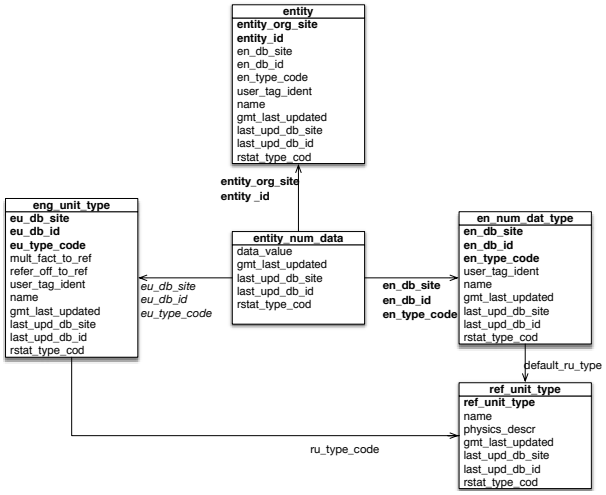

Figure 7. The CRIS data attachment mechanism.

The tooling then maps any domain attribute which cannot be mapped directly to a CRIS entity to one of the CRIS entity's attached `data` tables instead (`*_num_data`, `*_chr_data`, or `*_blob_data`). The name of the attribute is saved in the `*_*_dat_type` table, while the value is stored in the `*_*_data` table itself. The entries of the `*_*_dat_type` tables, which consist of the attribute names and unique type codes, can be generated by our tooling.

At all times, the tooling can provide a report, which domain entities have already been fully mapped, which are still missing some attribute mappings, and which are still entirely unmapped. For the OMAHA domain, we found that CRIS was sufficient to represent a majority of the domain entities, however, a significant amount of entities or attributes was left un-

mapped. OMAHA is a project within the aviation domain and one group of unmapped entities deals with the scheduling and allocation of aircraft to flight plans. In order to provide suitable data storage for the project we designed a *scheduling extension* to CRIS, presented in the following section.

4.7. CRIS scheduling extension

For the analyses to be performed as part of the OMAHA project, it is necessary to collect information about the usage of assets, in particular with regards to their scheduling. Since the CRIS data model does not provide entities for this type of information, we propose a scheduling extension to CRIS.

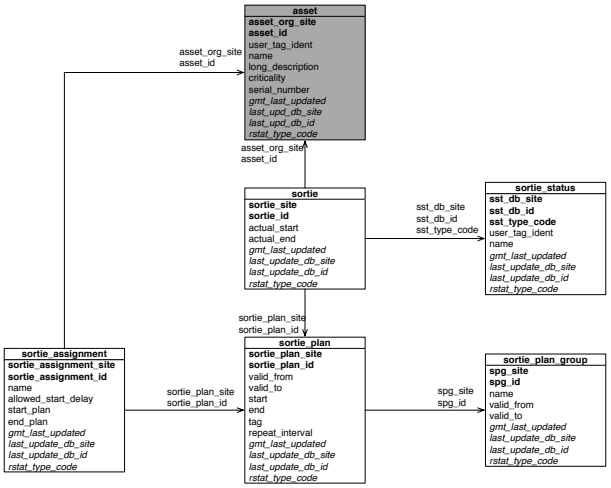

Figure 8. The CRIS scheduling extension.

Figure 8 shows the entities used to represent scheduling in the context of CRIS. A *Sortie* is a concrete usage of a specific *Asset* (defined as part of CRIS). It specifies an actual start and end date, as well as an *SortieStatus*. *SortieStatus* is a user definable success or failure status. A *Sortie* is associated to a specific *SortiePlan*, which specifies the scheduling of the *Sortie*. It has a time interval, during which it is valid, specifies scheduled start and end of the *Sortie*, as well as a repetition interval. Multiple *SortiePlan*s can be grouped into a *SortiePlanGroup*, which also has a period of validity. Finally, rotation planning can be represented via a *SortieAssigment*, which connects a *Sortie* to a *SortiePlan*. The *SortieAssigment* specifies an allowed start delay as well as start plan and end plan.

The domain specific entities from the scheduling context were mapped to a more generic representation in the CRIS extension. For example, a *Flight* represents the flight connection between two airports. This entity is mapped to a *SortiePlan*. The start and end times can be mapped directly to the attributes *actual_start* and *actual_end*. However, the departure and destination airports are not represented in the *SortiePlan*, because we followed the domain-agnostic spirit of CRIS. These attributes are mapped with the attached `data` tables, e.g. the departure airport is mapped to *SortiePlanNumData* with the code 16003 and the user tag identifier DEPARTURE_SITE.

4.8. Testing and evaluation

We generated several integration tests for each entity. The test cases cover the CRUD operations for all entities and their attributes extensively. We deployed a randomized testing approach, inspired by QuickCheck (Hughes, 2007). The code generator generates arbitrary data which is compliant to the database schema. This data is available during code generation time. Thus, it can be used to generate *assert* statements for the test cases and *INSERT* statements for data used during testing, as illustrated in figure 9. This approach made it easy to generate a high number of diverse test cases (about 30000). The broad coverage helped to identify edge cases - such as problems induced by the CRIS entity *Test* - early and greatly increase the confidence in the correctness of the implementation.

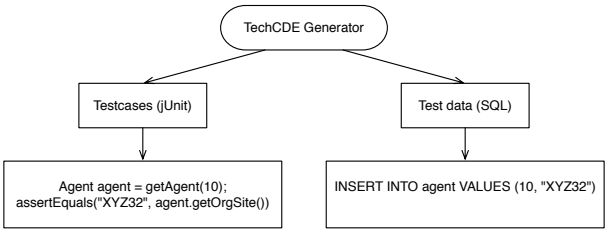

Figure 9. Test generation.

Our middleware was field-tested involving some of our partners in the OMAHA project. We deployed the Tech-CDE server at our company, secured with an Apache Web-Server enforcing SSL client certificates. We distributed both the Java and the C++ library, including client certificates. Two partners required to setup our client libraries with an authenticating HTTP proxy. While this setup was qualitatively feasible we hope to generate qualitative test results, especially regarding performance and scalability.

4.9. Limitations of the OSA-EAI Tech-CDE specification

We came across a number of ambiguities and areas of missing information.

Missing protocol specification

MIMOSA does not specify the protocol to be used when transmitting the XML. Possible solutions include HTTP GET or POST requests, socket communication, or file transfer to name a few. In order to better facilitate interoperability between Tech-CDE client and server solutions by different vendors, it would be beneficial if MIMOSA specified the protocol to be used. We chose to implement HTTP using POST to cater for

large payloads. Also, HTTP connections can be secured via SSL, and can be routed through complex proxy chains.

Specifying primary keys for entities

The Tech-CDE Client/Server Specification does not specify whether the client or the server defines the primary keys of the entities. A comment in the XML schema seems to suggest that it is the client, since it has to provide all required fields when performing an insert request. It is unclear, however, how the client should determine, which keys to use. There is no way of querying the server reliably for this sort of information, so the only possible solution seems to be for the client to try with a certain key and try again with a different key in case of an error, until the insert request succeeds.

On the other hand, it also does not seem feasible to let the server define the keys, since the protocol does not allow to communicate them back to the client. Because of this, the client would have no way to know the keys of the entities it just inserted.

Handling of base attributes

The four attributes `gmt_last_updated`, `last_upd_db_site`, `last_upd_db_id`, and `rstat_type_code` are present in all CRIS objects. Their purpose seems to suggest that they should be set automatically by the server, but no such requirement can be found in the Tech-CDE Client/Server Specification.

Handling of `null` values for strings

Some attributes can be set to `null`, but there is no concept of `null` in XML. This leads to a problem with strings, since the empty string is a valid value for string, so it cannot be used to substitute for `null`. For update requests, the client has to send all updated attribute values to the server, so omitting a string attribute which was set to `null` is also not possible, since this would be the same as not updating it at all. Since the Tech-CDE Client/Server Specification offers no guidance on this, we chose to disallow the empty string as a valid value for strings and instead treat it as `null`. In order to allow updating only some optional values, we generate template classes in addition to the entity classes themselves, which allow setting an arbitrary combination of values, regardless if they are optional or mandatory.

Error handling

The Tech-CDE Client/Server Specification is very sparse on details on error handling. It defines a list of example error messages, but does not specify the corresponding error codes. The range of responses for the server is also very limited, it can only signal success or failure, or the result set in case of a query. More detailed information on successful writes cannot be communicated. If HTTP is used as communication protocol, it is also not specified, whether HTTP error codes are also used to communicate failure.

4.10. Summary and Future Work

We implemented an OSA-EAI Tech-CDE client/server Application with a generic Java server (4.4) and an entity specific, generated C++/Java Client API (4.3) as a middleware solution for data integration of the OMAHA demonstrator. Further, we mapped aviation related entities to CRIS (4.6) and extended the CRIS schema with entities where necessary (4.7). Our automated integration tests and the deployment for project partners (4.8) show that we have provided a usable and scalable solution for data integration based on the OSA-EAI standard. However, the Tech-CDE standard is not yet fully implemented. Recursive queries of compound structures are not implemented and foreign key tables are not returned when the `return_fk_tables` attribute is set. We plan to implement these on the server in order to be fully compliant with the Tech-CDE specification, and for performance and usability reasons. A minor security improvement on the server side would be to introduce an additional whitelist for entity attributes. We motivate the addition of JSON or a binary format to the standard, instead of XML, for decreasing communication overhead. We plan to evaluate JSON with our architecture, knowing that it would break the standard compliance.

5. CONCLUSION

Our generative approach to implement a simple builder API for binary OSA-CBM messages revealed protocol inconsistencies regarding the specification of length fields, for which we presented a solution that makes binary OSA-CBM streaming-capable. We further showed that the protocol specification allows for assembling message subtrees of infinite length due to the presence of 758 unique circular structural dependencies. We recommend to add a maximum depth definition. Otherwise, implementations for real-time systems have to add an artificial (i.e., non-standardized) maximum depth to prevent memory or stack overflows.

We implemented a network layer for data integration following the specification of the Tech-CDE client/server application. Deviating from our previous implementation of the Tech-XML specification we found advantages in the simple mapping from CRIS entities to Tech-CDE requests. The direct representation of CRIS entities on the protocol layer lead to an easier association of requests to entities and more readable client side code. The generative approach with a custom generator allowed to develop a flexible client API for which correct Java code translates to correct Tech-CDE requests. Further, the generative approach made it possible to cover a broad range of test cases. These tests and the usage of the

Client API by our project partners suggest that the Tech-CDE client/server application is ready to be tested in a realistic and challenging environment. The roadmap for further improvements of the server is clear. In the following years we will be able to report on production usage of the system.

Our attempts to map the scheduling related tables to the CRIS schema show that it does not follow naturally how this domain should be mapped. Further, we introduced new tables in order to map this domain, because our analysis suggested that a mapping to existing tables was not possible. We tried to follow the philosophy of CRIS by only introducing domain agnostic entities and by retaining the soft attribute mechanism. This extension is a suggestion and could be used as a starting point to incorporate the scheduling domain into CRIS.

ACKNOWLEDGMENT

Financial support from the German Federal Ministry of Economic Affairs and Energy through project OMAHA, contract 20Y1302G, was essential for enabling the work, and is gratefully acknowledged.

REFERENCES

Fowler, M., & Parsons, R. (2010). *Domain-specific languages*. Addison-Wesley Professional.

Gonzalez, G. (2015). *State of the haskell ecosystem.* Retrieved 2016, from `https://github.com/Gabriel439/post-rfc/blob/master/sotu.md`

Hughes, J. (2007). Quickcheck testing for fun and profit. *Practical Aspects of Declarative Languages*, 1-32.

JAXB. (n.d.). *Jaxb project website.* Retrieved 2016, from `https://jaxb.java.net/`

Löhr, A., & Buderath, M. (2014). Evolving the data management backbone: Binary osa-cbm and code generation for osa-eai. In *Second european conference of the prognostics and health management society.*

Löhr, A., Haines, C., & Buderath, M. (2012). Data management backbone for embedded and pc-based systems using osa-cbm and osa-eai. In *First european conference of the prognostics and health management society.*

MIMOSA. (n.d.). *Mimosa organization website.* Retrieved 2016, from `http://www.mimosa.org`

BIOGRAPHIES

Johannes Drever received his M.Sc degree in Computer science from the Technical University of Munich in 2009 (Informatics, Diplom) and his PhD at the medical faculty of Ludwig-Maximilians-Universität in 2011. He worked for 4 years at Linova Software GmbH. His interests are functional programming in Haskell, data visualization, randomized testing and source code generation.

Helmut Naughton received his M.Sc. degree in Mathematics from Ludwig-Maximilians-Universität of Munich (Mathematik, Diplom) in 2005, and his M.Appl.Inf. degree in Applied Informatics from Technical University of Munich (TUM) in 2008. He worked at the Chair for Applied Software Engineering at TUM as a researcher in the field of software engineering for 6 years. Today, he works for Linova Software GmbH where he specializes on requirements engineering and developing information systems.

Michael Nagel received his M.Sc. degree in Computer Science from the Technical University of Munich in 2003 (Informatics, Diplom) and earned his PhD degree in Computer Science from Technical University of Munich in 2013. Today, he works for Linova Software GmbH with a focus on mobile application architecture and wireless hardware/software interfaces.

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.

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. 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.