Implementing reference APIs for AutomationML - A Java based walkthrough

1st Ronald Rosendahl
Otto-v-Guericke University
Magdeburg, Deutschland
ronald.rosendahl@ovgu.de

2nd Konstantin Kirchheim
Otto-v-Guericke University
Magdeburg, Deutschland
konstantin.kirchheim@ovgu.de

3rd Josef Prinz
inpro
Berlin, Deutschland
josef.prinz@inpro.de

***Abstract*—The exchange of information about and within production systems at engineering- as well as runtime of the system is a major task within recent industrial applications. Information and communication models or architectures are key enablers for this purpose. One powerful data format in this field is AutomationML. It is an XML based exchange format with meta-modeling capabilities for semantical richness. Due to its expressiveness and broad range of applications the implementation of software solutions with support for AutomationML raise high demands on system design. To help developers to get started general requirements as well as suitable architectures for the implementation of AutomationML applications are collected and discussed in this paper. Based on the collected arguments a reasonable implementation strategy for reference APIs in object-oriented programming languages fitting the specifications of AutomationML is given.**

***Index Terms*—AutomationML, API**

I. Motivation

Industrial Internet of Things, Smart Manufacturing Component, Engineering Cloud or Value Networks are just a few keywords visualizing the opportunities in a new branch of trade of the digitization of industrial production. The ambition of the actors in this field is the integration and application of specialized methodologies of different professions to optimize the value added. Common to all of them is the necessity to structurally aggregate or compute, process and apply data to business processes. When AutomationML was started the first hype of computer integration was bygone and what was left was a heavily vertical integrated technology landscape. The necessity of the horizontal integration of the plant planning process drove the mills of the AutomationML community for about 12 years now. It is remarkable that the issues and ideas raised in the AutomationML development may easily be superposed to the requirements of the emerging digitization trends in manufacturing industry. The major difference to denote is the focus on different phases of life-cycle involving a lot of new different trades with their specific requirements. To really make AutomationML yours in this current pork cycle of computer integration is to change the perspective from the initially intended file-based sink to a globally interlinked marketplace of engineering or planning data in a generalized structure. Therefor it is indispensable to have an abstraction of the underlying XML data to a suitable model for the runtime of intended application cases.

* The authors were partially supported by the INTEGRATE project funded by BMWi (support code 01MA17001D) as part of the program PAiCE

The associates of AutomationML society were aware of this demand and the first steps in this direction were taken under the phrase *Abstract API*. It was chosen as the name of an activity of the working group *Application Scenarios* in the AutomationML society started to provide a general pattern of objects conceptualized in the standard documents. This pattern would be a foundation for the automatic generation and evaluation of software modules capable of the format. Although the working group failed to provide a language-independent formal specification an important result was the semi-formal description of the business objects with a detailed description of their behavior. From software systems developer point of view the provided business object model does not serve automatic code generation but constitutes a recipe and evaluation pattern to create a die-cut software interface to an AutomationML model instance.

Within this paper we will present an implementation of the defined set of objects with their interfaces as a language specific reference API of the current results of the *Abstract API*. It will be provided along with a file-based back end called reference implementation here which will be transparent to an API user. The main effort of this work is the proof-of-concept of an object-oriented reference API, the gain of experience with the implementation to provide feedback to the activity to evolve the specification of the *Abstract API*. This paper will not present a general introduction or exploration of AutomationML but will give developers who are familiar with the data model a sound foundation for design decisions. Therefor in the following the paper is structured in four major sections followed by a conclusion. In section 2 available solutions in the software ecosystem are introduced. Section 3 provides a discussion of system architectures and programming paradigms related to possible applications. Within section 4 the implementation strategy for a Java based reference engine is being presented along with some conclusions drawn in its development. Section 5 presents the specification tiers and sections of the API developed in accordance and extension of those specified by the *Abstract API*. The content of the paper is being concluded in section 6.

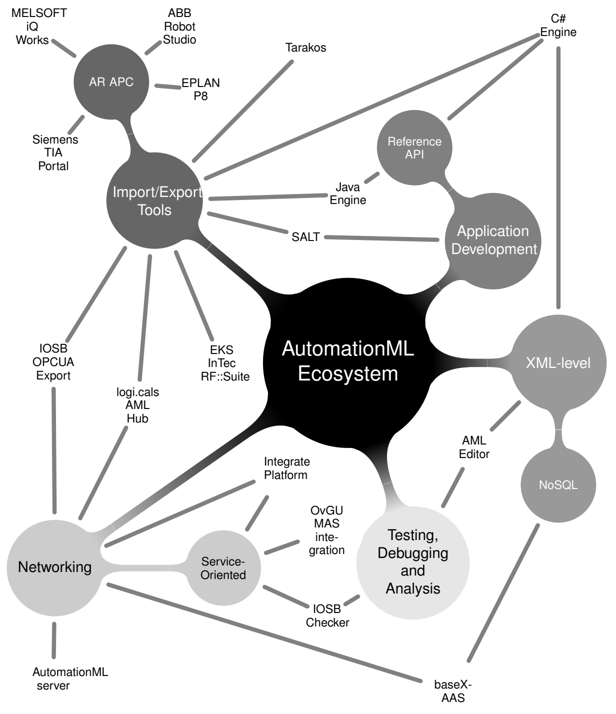

Fig. 1. AutomationML tool landscape

II. AutomationML Software ecosystem

Within the industrial application as well as in academic research several software tools capable of AutomationML were developed in the past. This section shall give a brief overview on the technologies, architectures and use cases they were applied to. The tools may roughly be categorized in 4 groups. The first one is constituted by tools that *import or export AutomationML* files or even both. These tools are usually built around their own business object model and define transformation logic from the internal data to the AutomationML file format or the other way around. Very often these implementations are focused on a single use case of transmitting a small set of artifact types from one application to another in a very narrow scope. Therefore they don't need to implement a full fledged AutomationML interface in the most cases. For example software used in simulation or virtual commissioning is built on data structures perfected for computation, feature optimization and visualization like the products of tarakos [1] or EKS InTec [2]. The standardized exchange of "Automation Project Configurations (AR-APC)" [3] is another example where AutomationML is used as an exchange format between applications with heterogeneous specific internal models realized with im-/export logic like in Siemens TIA Portal, EPlan P8, MELSOFT iQ Works or ABB Robot Studio [4].

A second group of application is utilized for *testing, debugging and analysis* of data exchange on file level. They support work-flows and operations closely related to the real content of the file. In many cases they reflect the structure of the XML file in the user interface and enable direct editing to solve or intentionally create errors in the file. These tools usually don't support the higher semantical layers of the format. Beneath the generic tooling of powerful text and XML editors that have shaped up to swiss army knifes of web- and software development there is the AutomationML Editor [5]. It has been started as a debugging tool and test case modeler but has emerged a universal modeling tool for AutomationML file content. Due to its increasing support of the object model and higher semantical layers of the standard it may also be mentioned within the next class of applications. Also to be mentioned in this class is a tool for conformance testing of AutomationML files that was developed by the IOSB presented in [6].

A third evident class is made up of tools built around an AutomationML business object model. Here many tools are created straight forward with the help of an XML-object binding library. A formal description like an XML-Schema (e.G. CAEX*.xsd) is compiled to an implementation-specific data model. The used library assists the transfer of the data from XML to the application and back. In such tools the application logic is implemented very closely to the ideas of the AutomationML models. Their output show a high degree of compliance to the standard. But as a consequence the business logic is complex and not perfected for the intended application. For example the C# based AutomationML Editor is built on the official AutomationML Engine [7] but there is also support for Java as introduced in this paper. Another implementation of this type with strong support for model driven applications is the EMF (eclipse modeling framework) back end created for several applications at the university in Vienna. [8] and [9]. A similar approach was used for model driven consistency preservation of engineering data at FZI Karlsruhe [10]. Beyond model driven approaches there are also tools implemented applying semantic web technologies to AutomationML [11], [6].

The fourth group is characterized by tools that try to overcome the limit of a directional file-based exchange of data between just a few peers. Such tools consider a high count of sometimes very heterogeneous partners in the data access or exchange. They care for huge sets of artifacts scattered across several files or AutomationML snippets often transmitted using web technologies. Such tools sometimes manage versions and variants as well as the consistency of contained artifacts as already presented in the previous tool class. Such tools support the integration of the engineering process as suggested with applications like AutomationML server [12], the AML.hub [13] or the INTEGRATE platform [4]. In the application of AutomationML within the runtime of *Industrie 4.0* use cases there are several proposals to use the data model in the asset administration shell. The export of an AutomationML model to an OPC UA model was proposed in [14]. This idea was enhanced to built a CPPS knowledge base implemented as an OPC UA server running a dynamic AutomationML model presented in [15]. A similar approach to implement a centralized asset administration shell for I4.0 but based on an XML database is presented in [16].

As we have seen AutomationML shows a broad range of application and the assignment of tools to a taxonomy is not unambiguous. Several tools have to be assigned to more than one class. Furthermore additional criteria and classification schemes may be used for example by applicability to the hardware in the different levels of industrial facilities, hardware requirements and performance measures, semantical complexity of the use cases or involved engineering disciplines and phases. Therefore the classification presented in figure 1 may neither be absolute nor complete but gives an overview on the tool landscape at the time of writing.

III. Implementation Architectures

In this section it will be discussed which considerations have to be taken into account when implementing applications above the model of AutomationML. For this purpose a shallow overview on system architectures and programming paradigms is given. Here some key findings of theoretical computer science will be applied to the topic but it is far out of scope of this paper to provide and apply the full formal background of application programming to this field. It rather shall provide a software-technical classification scheme and a foundation for a sound reasoning on design decisions within the development of AutomationML applications. Further it gives the background for the object-oriented reference implementation presented in section IV-B as well as arguments for the definition of intersections in tiers and segments of the API specification presented in section V.

For the exploration of this topic first an analogy is drawn. Looking on the textual representation of AutomationML as if it was the physical memory of a computer the discussion becomes amenable to the lines of reasoning of the different data access patterns and abstractions of programming language architectures. Based on this idea the following subsections will discus the different concepts on hypothetical AutomationML implementations. As depicted in figure 2 there are four major types of data abstractions to be mentioned which were conceptualized within the evolution of system architectures in parallell with the different generations of programming languages [17]. They were developed to manage the complexity of growing amounts of data and logic. While this evolutionary classification implies that each step overrules the former ones other schemes assume the coexistence of solutions optimized on different requirements. One very relevant classification is given in [18] that states classes of information structures oriented on the storage, processing or networking of information. This scheme is used to organize the following subsections.

A. Storage-Orientation - ”Machine Languages”

The least degree of abstraction can be found in the early days of programming within assembly and the upcoming 1st and 2nd generation languages. These languages are also referred to as machine languages due to their close relation to the hardware. Transformed to our text based representation of information a change of a register value in memory may be

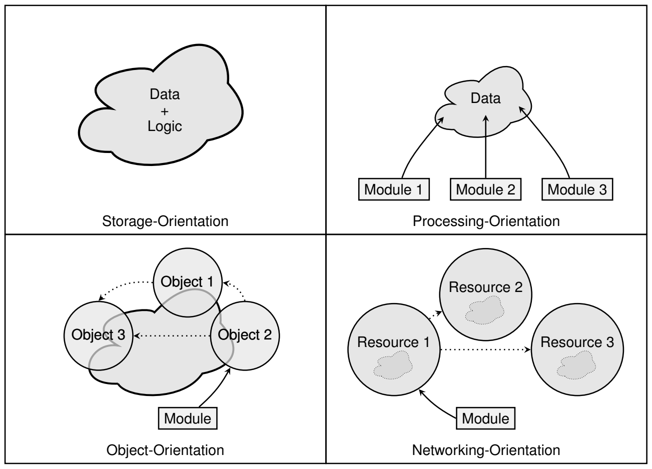

Fig. 2. Data Abstractions

compared to a change of a letter in the text file containing an AutomationML model. Implementations of low level functions have a small codebase and are straight and powerful by the immediate access to the data. But the functional entirety of an application is very hard to manage and with rising complexity of data and functions on that data it quickly gets unmanageable. The smallest change has the potential to destroy the consistency of the whole file. On higher semantical layers it gets even worse. For example renaming an attribute is a simple search and replace function but may require adjustments of references in an `AttributeNameMapping`. This may be achieved applying the change to each occurrence of the `Attribute`. But as a consequence this may unintentionally change another instance or even the `InterfaceClass` specification breaking the model contract of the toolchain. Anyway it is the most rapid solution but it has to be applied very carefully.

B. Processing-Orientation - ”Abstract Data Types”

A very important step in the development of programming languages was to relate data to its use. Distinct chunks of the data are provided with a certain behavior. This is accomplished by assigning types with a defined extent of values and operations available on that type. In theoretical computer science those types are called abstract data types and form a mathematical model of the expressiveness of the programming environment they are part of. By the combination of types to aggregate data types a structural description of the information representation in the data model is explicitly given.

Carried over to statically structured file formats one may identify a certain part of code as a structure of distinct type. By this it is possible to consistently change the content at an offset position predefined by the type in a predefined way and scope. This strategy is used by XML parsers (DOM/SAX) which convert the textual representation into a processable application-specific shape. The benefits are the generic access and manipulation of partial information as well as the preservation of consistency of the types contained in the file. A major drawback of this approach is that the structural patterns are static and have to be defined in advance. Furthermore the semantics of the structure is flat. Interrelations between different segments is entirely represented in the business logic. Hence the business logic is partially modularized by the functional approach of processing a type but still has to be designed and managed as a monolithic artifact to implement a complex application.

Applied to AutomationML this shows up for example in the process of assigning an `Attribute` change for all performers of a specific `RoleAssignment` would require the traversal of each entity to evaluate the type pattern. Then the change is eventually deployed to the structure. Sequel changes like necessary adjustments to dependent relations (e.G. `AttributeNameMapping`) have to be integrated in the same code. This renders the operation single purpose. An adjustment of the code becomes necessary not only if the semantic of the operation changes but also if the data it operates on has changed. Consequently if there are many operations defined on a certain data structure this may require a huge refactoring spread across the whole codebase.

C. Object-Orientation - "Model Driven"

The next major step in data abstraction was done within the refinement of 3rd generation languages (3GL). To maintain code more efficiently the idea was raised to tightly integrate the data with its behavior. This way the code that depends on a data structure may easily be identified and adjusted. Guided by this idea a complete new programming paradigm emerged under the term object-orientation. Since AutomationML states to be object-oriented the means of object-oriented programming (OOP) should perfectly fit the demands of the format. For this reason the discussion of this paradigm will be elaborated here in more detail. Due to the high degree of abstraction this architecture is very independent of the underlying physical representation of the data. This is one of the main intended benefits but as a consequence there is a broad variety of implementation options. Some of them will be discussed in section IV-B but to stay focused here the discussion will be organized along the four basic concepts of object-oriented programming.

1) Encapsulation: Certainly the most fundamental concept of object-oriented programming is object encapsulation. The internal structure of an object is hidden to the outside system. The functionality of an object is controlled interactively. Therefor an object exposes a so called client interface. Through this client interface the behavior of the object is triggered. This means that any business logic doesn't have direct access to manipulate certain data structures but knows the object and its available behavior to execute an operation on the data. Since operations of an object are processed only by the object itself there may not be any unmanaged interference of two logical operations. Further model state and behavior of the data are kept together. Dependencies between those characteristics are obvious and in case of any changes the extent of necessary refactoring may easily be estimated and managed.

Adopting our use case to the principles of object-oriented programming it has first to be clarified that AutomationML does not specify any dynamics of data. The behavior of each object has to be defined with the help of the standard definition by identifying any consistency constraints given in the normative documents affected by the intended change of each operation. The constraints are then interpreted as pre- and postconditions as well as invariants of the behavior. In several cases it is even possible to derive operational semantics for the operation.

For example adding an AutomationML object to another one would require the operation to ensure that the new object caries a UUID to be uniquely recognized in this system and preserving the consistency of the whole model. How this is achieved is part of the logic of the object and encapsulated from the superordinate implementation. Due to the complexity of interrelations of structural components within the AutomationML model there are several layers of operations with several degrees of consistency to distinguish. Their definition is one of the most challenging tasks in the specification of the reference API presented in IV-B and V.

2) Abstraction: The second basic concept is abstraction. As already discussed with abstract data types abstraction hides the internal structure and details of data and logic to transparently enable access to the true business logic of the system. Along with the concept of encapsulated objects abstraction renders the business model of the application a system of entities defined very closely to the structure humans intuitively conceptualize complex systems. The model is built up out of entities with certain characteristics and (sometimes interactive) relations between them.

To build a business object model for an application it is suitable to translate the domain knowledge represented in functional specifications into object definitions. A best practice of agile development for this purpose is the definition of user stories. These stories are broke down in an activity called story decomposition (which basically is an application of the divide and conquer concept). Thereby the story is split up into subordinate partial stories the superordinate story may be composed of. This is repeated until it reaches a specific level least abstract to be reasonable implemented as a single module. This strategy helps to identify necessary subroutines in the business logic. In case of OOP it identifies the different objects to be developed on common levels of abstraction.

Applied to AutomationML for example implementing an application for part 5 of the standard one would prefer to work with objects called network node and network connection instead of struggling with all the details behind (`InternalElements`, `AssignedRoles`, related `ExternalInterfaces` and `InternalLinks` in between). Section IV-B will present the example and the conceptual layers that have been identified thereby in more detail.

3) Inheritance: As the third concept of object-orientation inheritance enables the modeling of commonalities between objects. Therefor an object type declares to inherit behavior of another object type. This strategy enables a high level of code reuse. Mission proven code segments may be exploited by objects of several object types. There are different strategies of how to implement inheritance. Since AutomationML is based on CAEX it is defined in the realms of prototypic inheritance. Here an object has a backreference to it's prototype. A popular form of prototypic implementation is by only declaring behavior deviation from base type on an inheriting object. If no deviation is specified the inheriting object defaults its behavior to the prototype. This way equivalence classes of object behavior are established. For AutomationML there are several limitations defined to this concept. First of all the deviations are limited to be supplementary. This will be explained in detail in subsection III-C4. Second the type-instance relation is defined in the standard not to be a class inheritance. Further it is interfered by the mirror concept as well as the changed object identification strategy.

Practically spoken an `InternalElement (IE)` serves an instance while a `SystemUnitClass (SUCL)` (in case of CAEX another `IE`) serves the type. An `IE` must provide the property `RefBaseClassPath` which may provide a value. Here the path to the prototype for the instance may be given. But in AutomationML the object identification strategy was changed from location to identity key using UUID while the uniqueness of element names was dropped at the same time. Therefore paths lost the potential of identifying an instance uniquely. The reference to the unique identifier instead was chosen to mark the instance to be a mirror of the element given by the UUID. In consequence an `IE` may not inherit another `IE` anymore loosing the mechanism of prototypic inheritance. Remains the possibility to inherit a `SUCL` but this is explicitly defined to be a class instance relation for informative purpose and not an inheritance relation. This was tailored to the initially intended use cases of AutomationML. Tool vendors that define exchange between tools should negotiate on side contracts for type preservation. But the changed demands with the advent of Industrie 4.0 rendered it an issue. Distributed applications depending on a reliable type system have to reintroduce an explicit type instance relation within the data. Strategies to solve this issue should be subject to further discussions. In the case of a `SUCL` the inheritance is still up but limited. If a `SUCL` refers to a another `SUCL` it implicitly inherits all behaviors of the base class. Those have to be explicitly specified in (copied to) an `IE` that is created from a `SUCL`.

4) Polymorphism: The fourth basic concept of object-oriented programming is polymorphism. In this scope it would mean that an object could redefine a behavior of an inherited base object. It would behave different to the behavior of the equivalence class. But as mentioned in section III-C3 behavior deviation may only be defined supplementary. The structure of the inherited class is always implicitly derived and shall not be modified.

D. Networking-Orientation - "Cloud"

Networking-oriented information structures are applied in different architectures from simple statically spread network clusters up to cloud computing architectures. For this purpose they raise high demands on abstraction of data and processes on the data. This way they enable a distributed, asynchronous and redundant information retrieval, processing and storage. According to NIST under the term cloud computing a model is described for ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. These resources may rapidly be provisioned or released with minimal effort. [19]. This definition describes a model closely related to the vision of the information architecture of the factory of the future.

In the context of AutomationML such an information architecture may be described as a service-oriented processing of object-oriented distributed information models. Objects are clearly identifiable and linked via literals and URIs across the network. A communication technology exists that enables the exchange of objects and processing information between the network components. Architecture examples are Client-Server Architectures, Publish-Subscribe Networks, Cloud Architectures and IOT Platforms.

Data centric network-oriented communication technologies with an object-oriented modeling approach are for example OPC-UA (Open Platform Communications Unified Architecture) and DDS (Data Distribution Service), for communication in the field of industrial automation and machine to machine (M2M) communication.

Examples that have already been implemented were introduced in section II. There are applications of distributed engineering where engineering artifacts in form of AutomationML are synchronized within a repository on a network server. Different views on that integrated data are provided by server applications. In other use cases AutomationML is used as a telegram type to transmit partial model data like device configurations. Also the use of AutomationML models at the runtime of a system as a distributed online data structure hosting an asset administration shell in the context of I4.0 has been proposed. The concepts of AutomationML fit these demands but details on the dynamics of the model have to be elaborated. An open question is how changes in the model are logically expressed or how versions and variants are explicitly specified in AutomationML using for example the built-in `ChangeMode` elements. Another open issue is how time-critical communication based on AutomationML information models may be enabled.

IV. IMPLEMENTATION STRATEGY

In this section a reference API and a file back end for AutomationML developed at the Otto-von-Guericke University will be presented. It was created as a proof of concept to evaluate the *Abstract API* and provide feedback to evolve the specifications. Additionally it should introduce support of a new implementation language to the pool of AutomationML backends. Further it should fit the requirements raised in the academic context (platform independent, open and free libraries and IDEs). It should be small, modular and built on standard approaches.

A. AutomationML engine for Java applications

Java was chosen as the implementation language of the planned reference engine and API as it addresses all the mentioned requirements. In order to build a Java based API from the specifications of the abstract API working group, interfaces were created according to the list of identified business objects. To create the file-based back end a very popular approach was selected. AutomationML is entirely expressed in the syntax of CAEX which is provided with a formal specification in form of an XML-Schema file. Due to the popularity of XML there are free XML support libraries available for most programming languages.

According to Vinosky [20] its preferable to exploit the power of binding libraries over low-level access like (DOM or SAX) parsers to minimize impedance mismatch. Therefor Java classes are annotated to instruct the binding support to properly execute the transformation between XML and Java objects called (un-)marshalling. Given an XML-schema it is very easy to use a compiler shipped with the binding library to automatically generate such an annotated object model. This model implements all core functions for the manipulation of CAEX files right away. To fit the demands of AutomationML this model just needs to be extended by additional business logic and objects conceptualized in the standard, as elaborated in the *Abstract API*.

At this stage of development all core concepts identified within *Abstract API* (e.G. `RoleClassAssignment`, `MirrorObject`, ..) are represented as interface specifications with methods to read and write the values of their features. However, the object signatures were incomplete and did not yet cover all use cases (obviously given by the standard). Following the agile software development approach user stories were gathered and recursively decomposed down to technical stories that can be addressed on the API level created in prior steps. Assuming that each of this user stories corresponds to a concept it should be represented as an object. All of these objects are serious nominees to provide additional signatures to the API.

B. Engine Realization

In table I the results of the decomposition of the story *"modeling an ethernet conection"* is sketched. It shows that the behavior is disintegrated to a hierarchy of several reusable partial behaviors. For example the sub-stories for modeling a graph fit the concept presented in [21] and may be applied to several other stories (e.G the modeling dependencies, logistic networks or processing orders). This behaviors may be implemented in particular logical blocks. But to make use of the effort spent in other stories the hierarchical dependencies may directly be translated to inheritance relations. In case of Java language there is the concept of derivation or the application of the delegation pattern that fit this purpose best. The delegation

ID	Story	Dependencies
User Story		
1	modeling an ethernet conection	2,3,4
Consistent Edits		
2	Create a network container	5,6
3	Create a node	5,6
4	Create an edge	7,11
5	Add InternalElement to InstanceHierarchy	8,13
6	Instantiate a SystemUnitClass	8,12,14, ...
7	Link two ExternalInterfaces	9,14
Utilities		
8	Check if an ID is already assigned	
9	Get highest common parent of two Elements	10
10	Get parent of an Element	
11	Get ExternalInterfaces of a specific type from InternalElement	
12	Get ID of an Element	
Object Model Modifications		
13	Add InternalElement to an InstanceHierarchy	—
14	Add InternalLink to InternalElement	—
⋮		

TABLE I
STORY DECOMPOSITION EXAMPLE

was favored for the reference engine because it implements a loose coupling to maximize modularity. Also it helps to model and implent types and there inheritance that are not known at compile-time but at load-time or run-time of the system [22]. Therefor the adapter pattern was used.

Adapter types are offered to the application at run-time by dynamically linked factories. Each object is wrapped by suitable adapter to perform its behavior depending on the context it is used in. Especially the application of roles may be implemented in a semi-static way (type safe but post compile-time). The drawback of this architecture is the necessity of complex management code. For example the *self* pointer `(this)` does not implicitly refer to the business object and has to be forwarded from adapter to the adapted object. This effects a lot of functions part of the standard libraries that base their implementation on object identity (e.G. `compare(..) or equals(..)` method).

Another issue in the implementation was the freedom to leave out subtrees (containments) of AutomationML elements. This does not imply that the containment has been deleted. But the type system of Java presumes the existence of containments in an instance that are declared by a type. Therefore the absence of an entity has to be asserted. The Java concept of exceptions was chosen to represent the unexpected absence of a containment. This way absence and removal of an entity may be distinguished.

It also works out that three different types of functionalities emerge within an engine:

- **Consistent Edits** modifying the content in the model through the objects in a consistent way,
- **Utilities** that process complex queries on the model to support the behaviors of the objects, and

- **Object Model Modifications** that allow direct access to the object model without any further consistency checks.

Within the implementation process it was found that the *Complex Edits* merely extend the *Object Model Modifications*, as they wrap all functions enforcing consistency constraints. Therefore, method signatures in these packages are identical but show different semantics. For example, the edit for adding an `InternalElement` to an `InstanceHierarchy` (story 5) wraps the *Object Model Modification* with the same functionality (story 13) and enforces the unique-ID constraint (story 8). The *Consistent Edits* also introduce higher level functionality like creating an instance from a `SystemUnitClass`. To represent the different semantics preserving the same signatures different namespaces for the Java package where used. An alternative solution would be the definition of some kind of consistency level indicator. Setting the value of this indicator would orthogonally integrate the consistency preservation logic into the behavior of an object. Since this is one of the key concepts targeted by aspect-oriented architectures it was voted out of the object-oriented reference implementation but should definitely be discussed as a new implementation strategy for AutomationML.

V. Specification Tiers

The following section will dive deeper into the nature of the *Abstract API* specification and its implementation in the Java reference API.

A. Packaging

Table II lists all components of the AutomationML standard currently available to the public [23]. The AutomationML standard follows a layered architecture, where different tiers of the specification depend on previous parts. For example, the Communication whitepaper utilizes AutomationMLs capabilities to model graphs [21] in order to represent communication networks. Graph modeling for its part utilizes the Whitepaper Part 1 to represent graph structures.

Developers implementing an AutomationML engine based on the *Abstract API* should not be forced to implement every single aspect, but should rather be able to implement only a specific subset of the specification in order to optimize the code for the intended use case and avoid unnecessary overhead. Additionally, developers should be able to use existing (tested and possibly certified) code to build additional tiers of the specification on top of it. To take these design requirements into account, the *Abstract API* groups different interrelated parts of the standard into packages that can be built on top of each other. As elaborated through the story decomposition described in the previous chapter, at least three distinct types of functions can be identified. For each tier, these functions are grouped into separate sub-packages, namely:

- the **Core**-package, which provides basic access to all AutomationML business objects of a specification tier,
- the **Edit**-package which allows modifications of the object model consistent with the AutomationML standard, and

Document Identifier	**Core**	**Edit**	**Utils**
Whitepaper (WP)			
Arch	✓	✓	✓
Lib	✓		
Geo			
Logic			
OPCUAAML			
Comm			
eClassAML			
Best Practise Recommendation (BPR)			
RefDes			
Container			
DatVar			
RefVersion			
MlingExp			
MLA			
EDRef			
CstrRegExp			
Application Recommendation (AR)			
APC			
MES ERP			

TABLE II
AutomationML specification tiers

- the **Utilities**-package that contains query-like and miscellaneous functions that are out of scope of the individual business objects.

The current status of the Java engine is depicted in table II.

B. Additional Requirements

It can be stated that Java is incapable of expressing all requirements of the specification directly on language level. These inexpressible characteristics of the *Abstract API* collected as “soft constraints” have to be annotated in natural language to the code. Such constraints include for example the computational complexity, which can vary among different implementations, but must not exceed a certain limit in order to fulfill certain timing requirements. For example, a function providing a role-based service discovery mechanism for AutomationML on a project level can be implemented very efficiently using caching. However, when working with distributed models, the operation can not be guaranteed to terminate at all, as it might be unknown how many resources exist at runtime that have to be discovered and involved.

VI. Conclusion

In this paper the current state of developments with AutomationML were outlined. In the following programming paradigms and architectures were discussed to show the diversity of possible implementations. This makes up a good starting point for the design of an applications architecture. On the example of a Java implementation it is shown how a general implementation for AutomationML might look like and how it relates to the specifications of the *Abstract API*. Beneath feedback to the *Abstract API* a major effort of this paper is the identification of new challenges with emerging scenarios of usage. Especially for the distributed web-based operation of a ”living model” it has to be worked out how time critical information exchange may be realized or how division of labor and release processes in engineering and operation of systems may be implemented on AutomationML. This issues will be part of future work of the authors.

References

[1] E. Yemenicioğlu and A. Lüder, "Implementation of an automationml-interface in the digital factory simulation," October 2014.

[2] H. Hämmerle, A. Strahilov, and R. Drath, "Automationml im praxiseinsatz," *atp magazin*, vol. 58, no. 05, pp. 52–64, 2016.

[3] A. Lüder, N. Schmidt, and M. John, "Lossless exchange of automation project configuration data," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, Sept 2016.

[4] K. Stark, T. Goldschmidt, J. Doppelhamer, P. Bihani, and D. Goltz, "Cloud-based integration of robot engineering data using automationml," in *2018 14th IEEE International Conference on Automation Science and Engineering (CASE)*, 2018.

[5] J. Prinz, A. Lüder, N. Suchold, and R. Drath, "Design & engineering–automationml–integriertes engineering durch die standardisierte beschreibung mechatronischer objekte durch merkmale," *VDI/VDE: Automation*, vol. 12, 2011.

[6] M. Schleipen, "A concept for conformance testing of automationml models by means of formal proof using ocl," in *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)*, pp. 1–5, Sept 2010.

[7] R. Drath, *Datenaustausch in der Anlagenplanung mit AutomationML: Integration von CAEX, PLCopen XML und COLLADA*. Springer-Verlag, 2009.

[8] L. Berardinelli, S. Biffl, E. Maetzler, T. Mayerhofer, and M. Wimmer, "Model-based co-evolution of production systems and their libraries with automationml," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–8, Sept 2015.

[9] T. Mayerhofer, M. Wimmer, L. Berardinelli, and R. Drath, "A model-driven engineering workbench for caex supporting language customization and evolution," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 2770–2779, June 2018.

[10] S. Ananieva, E. Burger, and C. Stier, "Model-driven consistency preservation in automationml," in *14th IEEE International Conference on Automation Science and Engineering*, August 2018. accepted, to appear.

[11] M. Sabou, F. Ekaputra, O. Kovalenko, and S. Biffl, "Supporting the engineering of cyber-physical production systems with the automationml analyzer," in *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*, vol. 00, pp. 1–8, April 2016.

[12] S. Makris and K. Alexopoulos, "Automationml server - a prototype data management system for multi disciplinary production engineering," *Procedia CIRP*, vol. 2, pp. 22 – 27, 2012. 1st CIRP Global Web Conference: Interdisciplinary Research in Production Engineering (CIRPE2012).

[13] D. Winkler, S. Biffl, and H. Steininger, "Integration von heterogenen engineering daten mit automationml und dem aml. hub: Konsistente daten über fachbereichsgrenzen hinweg," *develop3 systems engineering, 3*, pp. 62–64, 2015.

[14] A. Lüder, M. Schleipen, N. Schmidt, J. Pfrommer, and R. Henßen, "One step towards an industry 4.0 component," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pp. 1268–1273, Aug 2017.

[15] R. Rosendahl, A. Calá, K. Kirchheim, A. Lüder, and N. D'Agostino, "Towards Smart Factory: Multi-Agent Integration on Industrial Standards for Service-oriented Communication and Semantic Data Exchange," in *Proceedings of the 19th Workshop"From Objects to Agents"*, pp. 124–132, Jun 2018.

[16] M. Wenger, A. Zoitl, and T. Müller, "Connecting plcs with their asset administration shell for automatic device configuration," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pp. 74–79, July 2018.

[17] M. Mezini, *Incremental Variations in Object-Oriented Programming*, pp. 1–41. Boston, MA: Springer US, 1998.

[18] J. Ameling, *Gestaltung der Wissensbasis von Unternehmen*, pp. 121–189. Wiesbaden: Deutscher Universitätsverlag, 2004.

[19] P. Mell, T. Grance, *et al.*, "The nist definition of cloud computing," 2011.

[20] S. Vinoski, "The more things change," *IEEE Internet Computing*, vol. 8, pp. 87–89, Jan 2004.

[21] A. Lüder, N. Schmidt, and S. Helgermann, "Lossless exchange of graph based structure information of production systems by automationml," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–4, Sept 2013.

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[23] AutomationML e.V., "Publications of the automationml association." available at https://www.automationml.org/o.red.c/publications.html (16.10.2018).