Model-Driven Consistency Preservation in AutomationML

Sofia Ananieva[1] and Erik Burger[2] and Christian Stier[1]

Abstract— **Over the last decade, standards such as AutomationML have emerged to support interoperability between tools along the engineering chain of Industrial Automation Systems. AutomationML incorporates various standards for different engineering domains, and defines common interfaces between them. Evolution is an integral part of the lifecycle of an Industrial Automation System. Thus, the AutomationML model of a system is subject to numerous changes. Such changes may lead to inconsistencies between the modeling artifacts from each engineering domain. Today, the propagation of changes from the models of one engineering domain to the other affected domains is still mostly a manual process. The process requires high effort and is error prone. In this paper, we reason on different kinds of consistency constraints and propose the Vitruvius framework to enable automated consistency preservation within AutomationML.**

I. Introduction

The engineering of Automated Production Systems (aPS) spans multiple disciplines, such as mechanical, electrical, and software engineering. During the different lifecycle phases of an aPS, engineers use several tools to create individual data models that describe the same engineering concepts. AutomationML or AML (Automation Markup Language) [5] is a standardized XML-based language that provides exchange of engineering data with a focus on plant engineering exchange. With the growing size of systems, the models created in AutomationML can also become very large and complex. Thus, the evolution of these models introduces possible inconsistencies, since a change that is applied to one part of the model can affect possibly many other parts.

The AML standard defines a variety of consistency constraints regarding, for instance, attribute values and referencing mechanisms of AML objects. The definition of these constraints, the detection of inconsistencies and their automated repair pose a major challenge for the application of AML. Often, these consistency checks are carried out manually, which is a costly and error-prone process for large systems.

In this paper, we present and categorize different kinds of consistency constraints in AML 2.0 and reason about the consistency rigor of the imposed constraints. We present concrete tool support using the model-driven Vitruvius approach to automatically repair inconsistency within AML and control consistency rigor based on user interaction.

[1]Sofia Ananieva and Christian Stier are with the Department of Software Engineering, FZI Research Center for Information Technology, Karlsruhe, Germany `ananieva@fzi.de, stier@fzi.de`

[2]Erik Burger is with the Chair for Software Design and Quality, Karlsruhe Institute of Technology, Karlsruhe, Germany `burger@kit.edu`

II. Foundations

In this section, we provide the necessary background for this paper which comprises an introduction to model-driven software development, the main principles of AML, and the Vitruvius framework to preserve consistency in view-based software development.

A. AutomationML in a Nutshell

AutomationML or AML (Automation Markup Language) [5] is an open XML-based standard for describing Industrial Automation Systems. It aims at the efficient exchange of engineering data from different domains, e.g., mechanical-, software-, and electrical engineering leading to reduced time and cost. The XML-based CAEX standard is a central part of AML. CAEX supports the definition of a common plant super structure and common interfaces following IEC 62424 [6]. Additionally, CAEX references external information that describe the geometry and kinematics of plant components following COLLADA 1.4.1 and 1.5.0 [4] and their behavior following PLCOpen XML 2.0 and 2.0.1 [12]. CAEX subdivides the plant engineering information into four basic viewpoints. The Role Class Library **(RCL)** contains Role Classes **(RC)** that represent specific roles, e.g., a motor or a robot. Such roles allow for specifying vendor-independent requirements or the definition of attributes for an AML object. The explicit linking of AML object with roles enables automatic interpretation by an engineering tool. The Interface Class Library **(ICL)** defines all required interfaces to describe a plant model. The **ICL** contains Interface Classes **(IC)** that specify interfaces between the objects in an AML model. An example of this is an interface type that links the AML object hierachy with an external 3D description of a robot. The System Unit Class Library **(SUCL)** represents vendor specific AML libraries. It contains System Unit Classes **(SUC)**, each describing a physical or logical object that can be matched with a particular **RC** defining its properties. The Instance Hierarchy **(IH)** comprises individual plant objects, i.e., Internal Elements **(IE)**, in a hierarchical structure and, hence, defines the equipment for a specific plant. The internal elements contain references to all previously mentioned viewpoints.

A special property of CAEX is that it follows a prototype-based paradigm. In object-oriented programming languages like Java and C++, objects instantiate statically defined classes. Prototypes are dynamically typed. For instance, **IE**s can either be created from scratch or instantiated from existing prototypical classes, i.e., **SUC**s. The instantiated elements represent clones of the respective prototypes retaining all properties of the prototypical elements.

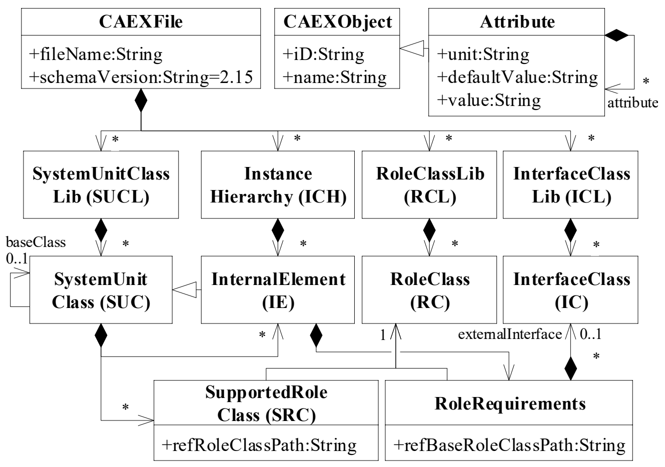

Fig. 1: An excerpt of the CAEX metamodel.

B. Model-Driven Software Development

Model-Driven Software Development (MDSD) [14] considers models as first-class engineering artefacts that formally represent a particular application domain. On those models, MDSD enables the usage of automated techniques, e.g., transformation of a model into executable code, or model validation. This differs from model-based development approaches, where such models may be used for documentation purposes only. Metamodels play a central role in MDSD as they capture a formal description of the model and define its elements and their relations. As outlined by Stahl et al. [14], a metamodel defines four artifacts: the *abstract syntax* represents the basic structure of notation elements and their relationships. The *concrete syntax* defines the notation of the modeling languages, e.g., a textual or graphical representation. The *static semantics* deals with the well-formedness of a model by specifying constraints over attributes and associations defined in the metamodel. Such constraints must always be satisfied, e.g., the static semantics in Java is concerned with an assignment of unique names for class attributes and operations. The *Object Constraint Language (OCL)* [16] is a formal language that is used to add static semantic constraints to a metamodel. Last but not least, *dynamic semantics* defines what happens upon execution of single model elements, e.g., invoking operations. A model which conforms to a metamodel is called an instance of that metamodel. Many model-driven applications are built upon the Eclipse Modeling Framework (EMF) [15]. EMF provides an in-build import mechanism for XML schemes like the XML schema definition of CAEX. EMF can generate a metamodel from the XML schema which complies with it. Figure 1 provides an overview of the CAEX metamodel. It covers the central objects of the CAEX standard, e.g., **SUCL**, **IH**, **RCL** and the **ICL** along with their relationships. The structural description of the metamodel is comparable to a UML class diagram. The conversion from XML to a metamodel enables the application of existing MDSD tooling to AML models.

C. The VITRUVIUS *Framework*

The VITRUVIUS framework is a model-driven approach for view-based software development [11, 9, 10]. *Views*, which are based on metamodels describing different aspects of the entire systems, form the central concept of the approach. These views often share common or dependent information, e.g., redundancies between the views. Views that share information need to be kept consistent to support the evolution of the system. To prevent inconsistencies, VITRUVIUS follows the idea of the *Orthographic Software Modeling (OSM)* which introduces a *Single Underlying Model (SUM)* [1]. The SUM is a monolithic model that comprises information of the entire software to avoid redundant representations of the same information. A user can access and evolve the system solely through views. The SUM conforms to an appropriate metamodel, the *Single Underlying Metamodel (SUMM)*. To reuse tool support for existing metamodels and refrain from difficulties of maintaining a monolithic metamodel, VITRUVIUS extends the idea of the SUMM by a virtual component: instead of one metamodel, the *Virtual Single Underlying Metamodel (V-SUMM)* consists of multiple existing metamodels that are interlinked through consistency preservation specifications. The *Virtual Single Underlying Model (V-SUM)* instantiates the V-SUMM, that consists of individual models representing different aspects of the system, e.g., the hierarchical plant structure, geometry,- and behavior description. Additionally, consistency specifications may be defined not only between different metamodels, but also within a single metamodel, e.g., the CAEX metamodel. To preserve consistency, views report all changes to the framework. The framework propagates the changes according to the consistency preservation specifications within the V-SUMM. For example, the changed name of a prototypical class leads to a reference update of the cloning class(es).
Furthermore, user interaction may be required to preserve consistency. This is the case if the user intent can not be unambiguously derived from an observed model change. For example, adding an **IE** to a CAEX model may represent a clone of another **IE**, of a **SUC** or not be a clone at all. To correctly preserve consistency, VITRUVIUS provides the possibility to define user interactions within consistency preservation specifications.

III. RUNNING EXAMPLE

In this section, we provide a running example of an CAEX 2İ5 model that is used throughout the paper for demonstration purposes of the developed approach.

Figure 2 illustrates an excerpt of a CAEX model that represents a simple manufacturing system. The **IH** *Manufacturing System* consists of a single electric screwdriver which, according to its required role class *Energy*, acts as a tool within the manufacturing system. The **SUCL** *Lib Of Common Tools* comprises a predefined library for a screwdriver object with two attributes: the *Revolution Per Minute (RPM)* and the torque of a screwdriver. Each attribute has a measuring unit, a default value and an

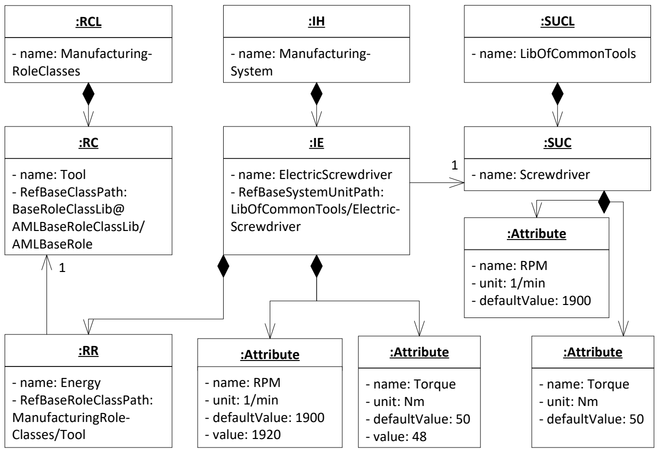

Fig. 2: The CAEX model of a small manifacturing system.

actual value. The attribute *RefBaseSystemUnitPath* of the **IE** *ElectricScrewdriver* defines the path to the prototypical **SUC** *Screwdriver*. The *ElectricScrewdriver* instantiates the *Screwdriver* and adopts its both attributes RPM and torque by additionally specifying the concrete values of both attributes. The motivating example will be used through this paper to illustrate different kinds of consistency constraints and demonstrate how consistency preservation specification can be enforced. Next, we will present an exemplary overview of consistency constraints of the CAEX model induced by the AutomationML standard.

IV. Consistency Constraints in AML

The AML and the subsumed CAEX standard define a series of consistency specifications that must be complied with when using AML [5, 6]. The publicly accessible *AML Whitepapers* [17] combine the respective consistency specifications. In this section, we present an excerpt of extracted consistency constraints from the defined consistency specifications and propose their categorization into *static* and *dynamic* constraints.

A. Static Consistency Constraints

A *static* consistency constraint is a model invariant, i.e., a condition that must hold true for the model at any point in time. Generally, the fulfillment of such a consistency constraint can be ensured by properties of a metamodel, e.g., the static semantics and abstract syntax (see Subsection II-B). Nevertheless, the static semantics of the generated CAEX metamodel only covers a subset of the identified constraints. This is, in essence, due to the limited expressiveness of the XML schema constraints compared to the static semantics of a metamodel. XML schema constraints are limited to generally define acceptable values for XML elements or attributes, but are, for example, not able to define value restrictions on a particular subset of elements, e.g., the CAEX tag *name* shall only be unique within a particular hierarchy level. Note that we refrain from manually changing the generated metamodel in order to preserve compatibility of the metamodel to the XML specification. The lack of full compliance to the AML standard additionally provides flexibility of possible tool functionality, e.g., regarding different consistency rigors (see V-B). In the following, we present an exemplary excerpt of the static consistency constraints in Table I.

Nr.	Consistency Constraint
#1	The CAEX root element *CAEXFile* of each AML top level document shall have the CAEX child element *Additional-Information*.
#2	Each AML document shall provide information about the tool which has written the AML document.
#3	The CAEX tag *name* shall identify all AML classes (**RCs**, **IC**s, **SUC**s) and be unique within the hierarchy level of the corresponding class.
#4	The CAEX tag *ID* shall identify all AML object instances (e.g., **IE**s). The identifier shall be a universal unique identifier (UUID) according to ISO/IEC 9834-8.
#5	Once created, a UUID shall never change over the life time of the corresponding project within all participating tools.
#6	References shall contain the full path to the referred class object.
#7	Cross-inheritance (e.g., an **SUC** inherits from an **IC** or an **IE** inherits from a **SUC**) is not allowed.
#8	All AML objects shall be associated directly or indirectly (via other AML objects) with the role class *Automation-MLBaseRole*.

TABLE I: Overview of exemplary static consistency constraints in AML.

The first constraint (#1) represents a static constraint on the structure of a CAEX model. The abstract syntax of the CAEX metamodel generally and, including this case, meets such structural requirements. Here, the attribute *Additional-Information* can always be created as a child element of the CAEX root element *CAEXFile*. The second constraint #2 requires to provide information about the tool which was used to create the CAEX model. The CAEX attribute *source-DocumentInformation* comprises this information. The static semantics of the CAEX metamodel specifies that the attribute has to occur at least one time within the CAEX model. The integrated validation framework of EMF checks the fulfillment of such static constraints and, hence, provides a validation error message if the respective attribute is missing. However, it does not check if the contained information uniquely identifies the tool. The remaining constraints (#4, #5, #6, #7, #8) deal with constraints for the identification and the referencing of CAEX objects. None of the constraints is ensured by the static semantics of the CAEX metamodel formalized in the CAEX schema. The listed constraints represent a illustrative subset of the extracted static consistency constraints.

B. Dynamic Consistency Constraints

A *dynamic* consistency constraint mainly relates to the prototype-based paradigm of AML: clone objects, e.g., **IE**s, instantiate a particular prototype object (i.e., **SUC**s, **IE**s). An AML object may represent either a clone or a prototype object or both during existence. Dynamic consistency constraints cannot be ensured by the properties of the CAEX metamodel. Instead, the support of such constraints has to be provided by model validation or consistency management frameworks. Table II presents an excerpt of the extracted dynamic constraints.

Nr.	Consistency Constraint
#1	If the prototype of a clone changes, this does not require a change of the mirror object characteristics. The update of mirror objects is a possible tool functionality.
#2	Changes in **RC**s shall be automatically reflected in referencing elements.
#3	The derived class inherits all attributes and features of the parent class.
#4	If an instance supports only one role, it shall be specified using the CAEX attribute *RefBaseRoleClassPath* of the *RoleRequirement* instead of using the *SupportedRoleClass*.

TABLE II: Overview of exemplary dynamic consistency constraints in AML.

The first constraint (#1) deals with prototype-clone-relations within AML. It leaves the consistency rigor between clones and prototypes to a possible tool functionality. Since the clone element copies the respective prototypical element including its properties during instantiation (see II-A), changes of the prototype may lead to changes of the clone enabling a full compliance between clones and prototypes. The second constraint (#2) represents a specific subset of the prototype-clone-relations. Unlike the previous constraint, all **RC** specifications shall be copied to **IE**s that refer the **RC** utilizing the CAEX attribute *RefBaseRoleClassPath* in their corresponding *RoleRequirement*. Specifications of **RC**s may be, for instance, the name of the **RC** or its attributes. The third constraint (#3) represents a dynamic consistency constraint regarding inheritance relations. Unlike possible consistency rigor levels between prototypes and clones, consistency between derived attributes and the respective parent attributes is not only desired but necessary during the evolution of the parent object.

V. VITRUVIUS FOR AML

In this section, we first explain how the VITRUVIUS approach can generally be applied to preserve consistency within the CAEX model. Especially, we focus on the implementation of dynamic consistency constraints as these are rarely supported by existing AML tooling and how to support compliance between prototypes and clones. Second, we reason on possible user interaction to decide on the consistency rigor between prototypes and clones.

A. Consistency Preservation in AML

So far, we have explained the VITRUVIUS framework and proposed the categorization of consistency constraints within CAEX according to their static and dynamic property. To explain how the model-driven consistency preservation of the VITRUVIUS framework can be applied within the CAEX model, we answer two main questions while referring to the running example presented in Section III:

1) How to detect and repair inconsistencies within CAEX?
2) How to create a CAEX model in the VITRUVIUS framework?

1) Detecting and repairing inconsistencies: Generally speaking, the VITRUVIUS framework monitors changes of models that instantiate particular metamodels and propagates those changes to other dependent models based on predefined consistency preservations. In contrast, consistency preservation of CAEX requires to apply the consistency preservation mechanisms of VITRUVIUS within one particular model, i.e., the CAEX model. Since consistency specifications usually define dependent or redundant information between different metamodels, the consistency specifications for CAEX define such information within the CAEX metamodel. Listing 1 represents a simplified consistency rule that will update the reference of a clone to a prototype if the name of the prototype changes.

```
 1 reaction PrototypeNameChanged {
 2   after attribute replaced at CAEX::SystemUnitClass
         [name]
 3   call {
 4     if((affectedEObject instanceof InternalElement)
         ) return;
 5     correctCloneReference(affectedEObject)
 6    }
 7 }
 8 routine correctCloneReference(CAEX::SystemUnitClass
        suClass) {
 9   match {
10     val clones = retrieve many CAEX::
          InternalElement
11     corresponding to suClass
12   }
13   action {
14      for(clone : clones) {
15         update clone {
16            clone.refBaseSystemUnitPath = CAEX.
                  updatePath(suClass)
17         }
18      }
19   }
20 }
```

Listing 1: Reaction to the change of a prototype name.

VITRUVIUS utilizes the specifically developed *Reactions Language* [11] to preserve consistency by updating a model instance after a particular model change performed by a user. Line 1–3 represents the *trigger* step that defines a consistency breaking change, and, therefore, requires an appropriate reaction to repair the consistency. In this case, the occurred change affects the name of a prototype represented by a **SUC** (e.g., the *Screwdriver*). Line 5–17 involves the *Reaction routine* that comprises consistency repair actions. Within the *match* step (line 7–10), elements that correspond to the element of the changed model are retrieved, e.g., the respective **IE**s representing the clones of the prototype (e.g., the *ElectricScrewdriver*). Next, a repair action is implemented to update the reference of the corresponding clone, particularly the CAEX path attribute *RefBaseSystemUnitPath* (line 11–15). Note, that a correspondence between a clone and a prototype has to be defined beforehand in another reaction to later retrieve the corresponding clones of a prototype. Using the *Reactions Language*, both static and dynamic consistency constraints can be defined along with respective repair routines. We provide a prototypical implementation of selected consistency constraints in the publicly available VITRUVIUS repository[1].

2) Creating the model: The EMF editor can be used to define CAEX models in a tree-based view. The editor supports the creation and deletion of model elements as well as modifications of the model structure via drag and drop. Furthermore, properties of model elements can be modified. The VITRUVIUS framework monitors the EMF editor for any kind of change that can be performed by a user on the CAEX model. If the occurred change corresponds with a consistency breaking change defined in a consistency specification, a respective routine is executed to restore the consistency of the CAEX model and updates the model respectively.

B. User Interaction regarding Consistency Rigor

Consistency preservation is not always a fully automated process. User input will be required to appropriately propagate information, if several options exist. In CAEX, this especially applies to dynamic consistency constraints regarding prototype-clone relations. As proposed by Berardinelli et al. [2], different levels of consistency rigor may be desired by a user:

- *L0: Uncontrolled compliance* occurs when a clone represents the exact same copy of a prototype only at the moment of instantiation. Afterwards, the clone evolves independently from the prototype.
- *L1: Substantial compliance* is stricter than the previous level and allows clones to extend, restrict or redefine attribute definitions of their respective prototype.
- *L2: Full compliance* is the strictest level where a clone remains the exact same copy of a prototype during the lifetime of the prototypical object.

In addition to the above mentioned levels, we propose the *L3: partial compliance* which allows a particular subset of clones to be either uncontrolled, substantial, or fully compliant to the prototype. Table III presents an excerpt of changes that can be performed a prototypical object and respective user decision options regarding the consistency rigor. The first change concerns the name of the prototypical object. Considering our running example, the **SUC** name *Screwdriver* may be changed to *AutomatedScrewdriverSystem*. The user may decide whether to update references of all clones (L2), of specific clones (L3) or of no clones (L0). In case of L2, the VITRUVIUS framework updates the reference attribute *RefBaseSystemUnitPath* of the clone to *LibOfCommonTools/AutomatedScrewdriverSystem*. The second change deals with the deletion of a prototypical object, i.e., the **SUC** *Screwdriver*. The user may either decide to delete all clones (the *ElectricScrewdriver*) or to create a new prototype in

Prototype Change Type	User Interaction Options
Name change	Update of all clone's reference Update of specific clones Leave as is
Deletion	Delete all clones Create new prototype Delete specific clones Leave as is
Attribute change	Update of all clones Update of specific clones Leave as is

TABLE III: Overview of dynamic consistency constraints in AML.

order to preserve full compliance (L2), delete specific clones (L3) or not react to the change at all (L0). The third change concerns all possible changes of the prototype's attributes which can be the addition of an attribute, its deletion or the change of an attribute value. For instance, the *Screwdriver* may receive a new attribute defining its mass. Again, the user may decide whether to update all clones (L2) with the new attribute, update only specific clones (L3) or refrain from consistency preservation completely (L0). Note, that we are currently focusing on change propagation to clones based on performed changes on prototypes.

VI. RELATED WORK

In this section, we focus on related work that targets the dynamic and static consistency preservation in AutomationML. Closely related to our work, Berardinelli et al. [2] focus on model-driven consistency preservation between prototypes and clones in AML and propose different levels of consistency rigor (see: V-B). Similiar to our work, the approach detects and repairs inconsistencies between prototypes and clones by updating inconsistent clones. Additionally, user interaction mechanisms are provided, i.e., annotation of inconsistent clones with warning messages that provide additional information to reason about the occurred inconsistency. Similiarly, VITRUVIUS provides information about the occured inconsistency and, additionally, performs a consistency repair operation based on the user's decision. As another significant difference, Berardinelli et al. utilize the *Eclipse Validation Language (EVL)* [7] to define consistency constraints for the different consistency levels. EVL, however, follows a state-based approach to determine changes that can lead to inconsistencies while VITRUVIUS follows the delta-based approach [3]. While the first approach determines the performed changes by comparing two versions of a model, the latter approach describes performed changes as a sequence of atomic edit operations. State-based consistency preservation approaches like the approach by Berardinelli are less challenging to apply since consistency constraints can be checked at arbitrary times for different model versions. Change information, however, cannot be derived unambiguously from discrete states. Our change-driven approach avoids this issue by enforcing consistency immediately after

[1]https://github.com/vitruv-tools/Vitruv-Applications-PlantEngineering

a change occurs. Kovalenko et al. [8] compare semantic web technologies and model-driven software engineering (i.e., EVL and OCL) to provide reasoning capabilities like constraint checking, data querying and data integration in AML. Additionally, the authors propose an AML ontology that reflects CAEX structures utilizing the *Web Ontology Language (OWL)* to support data analysis activities. The publicly available AML Analyzer [13] comprises the developed AML ontology and allows the above mentioned reasoning capabilities based on semantic web technologies. Although both OCL and OWL enable the definition of consistency constraints along with custom user messages and advanced reasoning capabilities, they provide no possibility to repair inconsistencies automatically or semi-automatically based on user decisions.

VII. Conclusion and Future Work

In this paper, we have presented an approach to detect and repair inconsistencies in systems modelled with AutomationML (AML). Therefore, we have extracted consistency constraints from the publicly accessible *AML Whitepapers* [17] that subsume the respective consistency specifications. We have classified the consistency constraints according to their *static* and *dynamic* properties. While the first constraint type represents invariants that must hold at any point in time, the latter constraint type mainly relate to the prototype-based paradigm of AML. We have explained how the Vitruvius framework can be used to preserve both types of consistency constraints automatically, taking user decisions into account which are especially useful in order to control the consistency rigor of dynamic consistency constraints.

As future work, we plan to compare our implementation of the consistency rules in Vitruvius with other implementations of the same rule set in other languages, e.g., the Epsilon Validation Language (EVL, [7]). The goal of such an evaluation will be to show that the rules can be expressed more efficiently in Vitruvius than in other languages or platforms. Furthermore, we will apply our approach within the INTEGRATE project[2] to ensure the consistency of a CAEX model in a collaborative working environment. As part of the project, we will also extend the inter-model consistency relations to further metamodels involved in AML, such as Collada and PLCOpen.

ACKNOWLEDGMENT

This work was partially funded by the Federal Ministry of Economy and Energy (BMWi), following a decision of the German Bundestag in context of the INTEGRATE project (grant agreement 01MA17001B).

References

[1] C. Atkinson, D. Stoll, and P. Bostan. 2010. Orthographic Software Modeling: A Practical Approach to View-Based Development. In: *Evaluation of Novel Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 206–219.

[2] L. Berardinelli, S. Biffl, E. Maetzler, T. Mayerhofer, and M. Wimmer. 2015. 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.

[3] E. J. Burger. 2013. Flexible Views for View-based Model-driven Development. In: *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*. Vancouver, British Columbia, Canada: ACM, pp. 25–30.

[4] Digital Asset and FX Exchange Schema. 2004. https://collada.org. COLLADA.

[5] IEC 62714 – Engineering data exchange format for use in industrial automation systems engineering – AutomationML. N.d. www.iec.ch. 2014.

[6] International Electrotechnical Commission: IEC 62424 – Representation of process control engineering – Requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools. 2008. www.iec.ch.

[7] D. Kolovos, L. Rose, R. Paige, and A. Garcia-Dominguez. 2010. The Epsilon Book. Eclipse.

[8] O. Kovalenko, M. Wimmer, M. Sabou, A. Lüder, F. J. Ekaputra, and S. Biffl. 2015. Modeling AutomationML: Semantic Web technologies vs. Model-Driven Engineering. In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–4.

[9] M. E. Kramer. 2014. Synchronizing Heterogeneous Models in a View-Centric Engineering Approach. In: *Software Engineering 2014 – Fachtagung des GI-Fachbereichs Softwaretechnik*. Vol. 227. Doctoral Symposium. Kiel, Germany: Gesellschaft für Informatik e.V. (GI), pp. 233–236.

[10] M. E. Kramer, E. Burger, and M. Langhammer. 2013. View-centric Engineering with Synchronized Heterogeneous Models. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. Montpellier, France: ACM, 5:1–5:6.

[11] M. E. Kramer. 2017. Specification Languages for Preserving Consistency between Models of Different Languages. PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT). 278 pp. URL: `http://nbn-resolving.org/urn:nbn:de:swb:90-692845`.

[12] PLCopen for efficiency in automation. 1992. https://plcopen.org. PLCopen.

[13] M. Sabou, F. Ekaputra, O. Kovalenko, and S. Biffl. 2016. Supporting the engineering of cyber-physical production systems with the AutomationML analyzer. In: *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*, pp. 1–8.

[14] T. Stahl, M. Voelter, and K. Czarnecki. 2006. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.

[15] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. 2009. EMF: Eclipse Modeling Framework 2.0. 2nd. Addison-Wesley Professional.

[16] J. Warmer and A. Kleppe. 2003. The Object Constraint Language: Getting Your Models Ready for MDA. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[17] Whitepaper AutomationML Part 1 – Architecture and general requirements. 2014. AutomationML consortium. URL: `https://www.automationml.org/o.red/uploads/dateien/1485867599-AML_Whitepaper_Architecture_V2.0.0.zip`.

[2] `http://www.integrate.ovgu.de`