2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-7281-2989-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ETFA45728.2021.9613164

Generative and Model-driven SDK development for the Industrie 4.0 Digital Twin

Nico Braunisch*, Marko Ristin-Kaufmann†, Robert Lehmann*, Hans Wernher van de Venn†

*TU Dresden, Institute of Applied Computer Science, {nico.braunisch, robert.lehmann}@tu-dresden.de
†Zurich University of Applied Sciences (ZHAW), Institute of Mechatronic Systems (IMS), {rist, vhns}@zhaw.ch

***Abstract*—Industrie 4.0 maps the digital twin as the Asset Administration Shell. To develop this digital twin, the Asset Administration Shell meta-model is used. A developer needs high-quality libraries in their programming languages to build the Asset Administration Shell in efficient and ergonomic way. The existing representations of the meta-model are inconvenient for moving from the specification to realizations in software development. This leads to many, often manual, re-implementations of the Asset Administration Shell in each programming language. We present an approach to automatically generate libraries based on the intermediate representation of the meta-model. The approach allows for practical and simple development process of Industrie 4.0 components, and scales across programming environments and future changes in the meta-model.**

***Index Terms*—Asset Administration Shell, Meta-Model, Code Generation, Design-by-contract, Model-driven Architecture**

I. Introduction

The concept of a digital twin is a cornerstone of the Industrie 4.0 (I4.0). A digital twin allows us to reason about a physical and/or ideal *assets* in the virtual (*i.e.* cyber) space thus providing a necessary foundation for complex and large cyber-physical systems.

In this context, an asset is any entity of value owned by or under the custodial duties of an organization. Recently, the Asset Administration Shell (AAS [1]) established itself as a standardized representation of a digital twin.

How we interact and use a digital twin depends on the context. For example, we can use it as a statical or dynamical description of the asset, use it as a shell to execute operations in the physical world, observe events *etc.* In this work, we focus on the first perspecitve, *i.e.* AAS as a data structure.

In this light, an AAS is an explicitly-formed data structure. It follows the well-defined Meta-model of the AAS which prescribes how to model all aspects of the corresponding asset as a component of I4.0. The meta-model is split in different elements which are combined to describe the asset. These elements vary from simple data elements, like properties, *via* detailed events and operation definitions to complex relations and references (*e.g.*, asset capabilities). Together with the description of these elements, their attributes and the applicable conditions, the AAS meta-model forms the foundation of interoperability in I4.0.

Practical I4.0 systems are realized in plethora of programming languages and runtime environments. Given the centrality of AAS for development of I4.0 components, it is important that we can process AAS's in an ergonomical and convenient way. Therefore the realization of the AAS meta-model should be available in a variety of software development kits (SDKs), supporting as many frameworks, programming languages and paradigms as possible. At present, the SDKs are implemented manually for each programming setting in separation. This development process involves a particularly laborous process of interpreting and codifying the specification from [1] to the individual programming languages.

The current work presents a novel hands-on tool chain based on modern software engineering to facilitate scalable creation of SDKs meant for seamless development of I4.0 components. Our contribution is two-fold:

1) We introduce a domain-specific language, *Simplified Python*, to formally specify the AAS meta-model.
2) We develop code generators to translate the meta-model and automatically generate SDKs in different languages and for different runtime environments.

II. Problem Statement

The official I4.0 meta-model is specified by the Platform Industrie 4.0 Working Group "Reference Architectures, Standards and Norms" and is published as a book [1]. The book contains the description of the model, its elements as well as the constraints accompanied by UML diagrams and tables. The constraints define dependencies, conventions, and relations within the metamodel and are specified infromally in human language. In the appendix of the document, as well as in a repository on GitHub [9], codified representations of the model are given in form of *schemas* in XSD, JSON-D *etc.*

At the moment, there exists no comprehensive set of SDKs or libraries for building an AAS. Many disconnected teams develop and maintain their own libraries. Each team needs to interprete the specification each on their own when they design their libraries. In certain cases, the schemas can be used to automatically generate the model elements. However, the schemas omit the constraints between the model elements. Thus, the constraints need to be modelled manually by each team following the specification.

This also applies to the operations on the model elements, *e.g.*, for creating, adding or removing model elements, or consistency and conventions checks. These operations are not defined by [1]. The follow-up specification [2], which also specifies the runtime API, is currently incomplete. In

2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-7281-2989-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ETFA45728.2021.9613164

particular, the API does not cover generation, traversal, or transformation of the AAS model elements based on the AAS Meta-model. This leaves again each developer with a tedious task to come up and implement his/her own set of operations.

Furthermore, a major backwards-incompatbile version of the AAS Meta-model is released every couple of years. Upon breaking changes, the developers have to manually work around or re-implement their AAS models and the respective operations. This is a laborous effort and needs to be performed minitiously to avoid unexpected inconsistencies.

Hence the current state of the ecosystem around AAS is fragmented and provides no unambiguous codification of the AAS model derived from the meta-model. The subjective interpretation of the specification and manual modelling thus lead to mismatches between two realized AAS models derived from the same meta-model. This means that multiple realized AAS models vary depending on the programming language and paradigm used, as well as the skills and understanding of the model maintainer.

III. Design

We propose a generative and model-driven approach to the problem. We first translate the domain-independent AAS meta-model from [1] to an intermediate representation in the domain-specific model. The intermediate represention is then used to generate the final usable language-specific model of an SDK or a library. Figure 1 demonstrates the process.

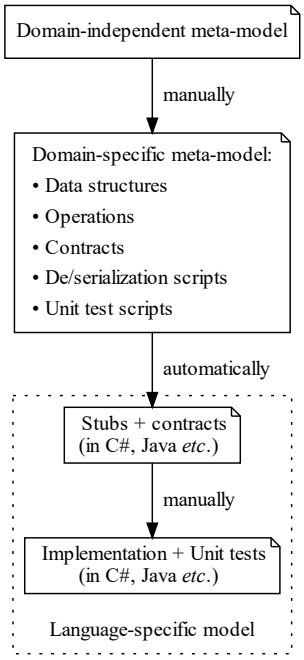

Fig. 1. Model-driven SDK generation process

A. Domain-independent Meta-model

In our approach, the domain-independent model from [1] represents the AAS meta-model with structural and functional elements along the requirements and constraints of these elements. There are no further technological, modeling, or development dependencies. However, the specification provided in UML diagrams, tables, and text is often informal.

To compensate for the ambiguity of this representation, the accompanying schemas[1] are used as a basis for a more formal representation. Unfortunately, the current schemas miss some dependencies and constraints so we consider them in our first attempt only as a reference and manually interprete the specification in [1] for the model generation.

B. Domain-specific Meta-model

To generate the specific AAS model elements and the operations on these model elements for the SDK, we need to create a formal representation capable of capturing all the dependencies and constraints. This representation also needs to include the necessary operations for querying and manipulation of model elmenets as well as consistency checks.

This model and the related operations depend on the constraints of the AAS model elements, and so it depends on the explicit modeling constraints of the AAS meta-model. At this stage, we avoid any implementation or runtime-specific constraints (such as programming language-specific functionality or operating systems support). This representation also abstracts away the constraints of the AAS model representation in volatile memory or persistent storage. We ignore the concerns related to multi-threading and (database) transactions.

Therefore, we introduce the domain-specific model as an intermediate representation of the model. We specify the operations on the model elements using the type annotations. The behavior of the operations is given in program code. The constraints are modeled as pre and post-conditions. These contracts can be simple, such as specifying a range of a property, as well as very complex expressions. For an example of the latter, consider the length of a list needs to be divisible by three and the first element needs to be larger than the sum of all the other elements.

C. Language-specific Model

In the final step, we generate the source code and the libraries for the respective language-specific SDKs based on the domain-specific model. This approach allows us to scale both in changes to the meta-model and the number of supported programming environments. The domain-specific model needs to be updated *once*, and the changes are semi-automatically propagated to the language-specific models. In contrast, the existing approaches re-interprete the meta-model *for each programming environment individually*, making the creation of SDKs much more labor-intensive.

The generated code highly depends on the language as well as the runtime environment, and is versioned appropriately. For example, the version refers to the version of the AAS, the programming language and, if relevant, the runtime environment (*e.g.,* AAS version 3.0 RC01, C# 9,

[1]The most suited appears to be the Resource Description Framework (RDF) Notation, while the XML, XMI, and JSON notations lack most of the constraints.

2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-7281-2989-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ETFA45728.2021.9613164

dotNet 5). We shape the generated code such that it employs existing toolchains to guarantee a maximum support to the developer in the respective language and IDE. The generated code also leverages paradigm-specific concepts and standard libraries of the respective programming language and available frameworks. For better readability, the documentation from [1] is intertwined in the code appropriately.

At this point of development, our generated code only provides essential operations on the elements of the AAS metamodel. The API functionality in perspective to [2] and the query language are left as a future work.

IV. Realisation

A. Assumptions and limitations

For tight development cycles with short feedback loops and quick adaptations, we impose the following **constraints** on our proof-of-concept:

1) Our main user base is the developers working on stable production systems. However, the library should be flexible enough to allow the use in their experimental code bases as well.
2) Certain design patterns, like visitor and transformer, should be supported out-of-the-box.
3) The library should minimize the dependencies for better portability. The implementations should be in native code instead of wrapping or encapsulating other code fragments. This allows language-specific power features such as reflection.
4) To prepare for future migrations, factory methods should be used to handle construction of the objects.
5) Computational efficiency is important. The library will be used to process large batches as well as on servers that need to have large throughput.

We also anticipate the following **limitations**:

1) The data model of an AAS fits in memory.
2) The thread-safe operations, persistency and database transactions are out-of-scope.
3) The management of the elements of the AAS in the perspective of storage and transmission is the responsibility of other projects such as aasx-server [7].
4) The operations on the AASX package format are handled by a separate library, *e.g.*, aas-package3-csharp [8].
5) The generated library should provide basic querying through dedicated operations based on the existing implementations, such as the data model of the aasx-package-explorer [6]. Since query language has not been standardized yet, we do not implement it at this point.
6) We support the local environment of the AAS, but not the handling and provisioning of separated AAS in a shell repository. The developer should be able to implement the repository using the generated library.
7) The library should provide import and export for JSON and XML notation of the AAS. Other formats such as AutomationML and RDF are not supported out-of-the-box. However, adding imports and exports for additional formats should be straightforward, so that users can develop their import/export libraries on the top.

B. Simplified Python

We use a simplified version of Python as the universal language of our domain-specific model to be transpiled into the respective programming languages such as C#, Java, *etc*. This *Simplified Python* supports only a certain subset of language constructs and built-in functions. The language is used for defining data structures and operations, contracts, and scripting (de-)serialization. As the scripts are only used by the developers of the code generators, we intentionally decide on the go which Python features to support. Hence, we only transpile a very limited subset of built-in functions and types. For example, we allow no nested functions, context managers, and nested classes.

We specify the data structures and operations using our Simplified Python. We use markers (in form of decorators) to annotate the data structures with special, domain-specific semantics (such as *entity* or *read-only property*).

Based on this specification, we generate the data structures and operation stubs in the respective library implementations. Many operations, such as simple add/remove commands, are trivial to script and transpile into the implementation. For such operations, we script them in our Simplified Python directly in the meta-model. Thus, their code is automatically generated and does not need to be written manually in the respective implementations.

The complex operations which cannot be scripted are marked with appropriate markers. Analogously, we also mark properties and data structures which we can not define meaningfully in Simplified Python. The developer needs to implement the marked spots manually in separate code snippets written in the corresponding implementation language.

C. Code Contracts

The specification of operations in our meta-model should include the code contracts such as pre and post-conditions. The code generators transpile contracts so that they are included in the implementation. Therefore, the implementation of the library allows for deeper testing as contracts are checked during debugging, testing, and in production. Additionally, where the contracts are tight enough, they allow for static analysis and automatic testing of the implementation. We also include the contracts in the documentation of the implementation to make it more formal and less ambiguous. This is important since the documentation in a human language tends to "rot", while contracts are continuously and automatically verified in testing or production.

Conceptually, the contracts are clearly delineated from input validation. The contracts improve the correctness of the code, while the validation of the input is part of the deserialization scripts. The deserializations check that the input is correct and conforms to the meta-model, and eventually report the errors to the user. In contrast, the contracts give us a certain assurance that the deserialized objects are internally consistent and

2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-7281-2989-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ETFA45728.2021.9613164

conform to our meta-model. In other words, the deserialization scripts verify the external data, while contracts verify the internal data. For example, we can use contracts to ensure that our deserialization works. Instead of writing many unit tests, we rely on the contracts to be verified during testing. A test boils down to simply deserializing the example JSONs.

The contracts are written in our Simplified python and transpiled by code generators into the implementation stubs. As invariants are not supported in most languages, we decide to define contracts only as pre and post-conditions of operations.

Each contract must have an identifier, a unique one if possible. In implementation languages where we cannot produce an error message, we will at least display the identifier to the user. If the identifier is omitted, it equals the body of the contract.

The contracts should also include an error message. The error message should support printf format so that we can insert the offending values in the message as well. This is important for contracts which are enforced in production, catching rare and hard-to-reproduce bugs. If the error message is omitted, the code generator will try to infer the format string based on the types of the arguments of the contract condition.

D. Validation

Since we are going to provide a variety of libraries and SDKs based on the same meta-model, the constraints of most of the unit tests will be the same across the different languages. Therefore, we script those unit tests in a common language. Similar to representation of the AAS metamodel in Simplified Python, we generate the implementations in the respective languages based on these unit test scripts.

To establish the correctness of the chain deserialization – contracts – serialization, we test for identity. The serialization of the deserialization of an input should give you the very same result. All valid test cases should satisfy this property, and all invalid test cases should report one or more errors.

We will collect multiple examples of valid and invalid AAS in serialization of JSON and XML for the test cases. The test cases also include AASs such as those which are invalid according to the schema or those which are valid according to the schema, but represent an invalid shell.

Manually constructing AAS is tedious, and we will almost certainly miss many edge cases. Therefore, we turn to automatic generation of test cases to increase the test coverage. We randomly erase required parts or flip the types of the JSON schema. Using such a defect JSON schema we generate the invalid data automatically by sampling.

Automatically generating cases which are valid according to the schema, but invalid in respect to AAS metamodel is hard as we do not know wheter the deserialization or the contracts are at fault. Therefore, we generate JSON data using JSONSchema and manually inspect (and record) the failing examples.

E. Code generators

We develop a code generator for each programming language separately. A code generator takes our domain-specific model as input and generates the stub of the library. Additionally, it generates the interfaces and data structures and transpiles the contracts as well as the de/serialization. The developer needs to fill in the stubs to finalize the library.

We aim to provide as much native support for as many programming languages and runtimes as possible. To provide a large variety in the final stage, we validate as many different paradigms and platforms as possible.

In this early stage of development, we prioritize the major programming languages in I4.0 by their share of use in industrial applications and I4.0 projects. We will start with C# and evaluate our approach in practice by integrating with other open source projects such as aasx-server [7]. Other languages such as C/C++ (for embedded software), Java (for enterprise applications), Python (for prototyping and research) and others will follow. For the backend and network development we aim to include Rust, Go and Erlang in the future. Further, we consider a IEC61131 compatible solution to be deployed on the production machines.

V. Future Work and Challanges

We have introduced a generative and model-driven approach to producing SDKs in different programming languages for development of I4.0 components. In our first attempt to create a general AAS SDK based on the AAS Meta-model, we focused on fast progress and early proof-of-concept. At the current stage, we are focusing on a subset of the AAS meta-model, including *Common Attributes* such as *Referable*. We need to model more elements of the AAS meta-model in our domain-specific model and align them with the given constraints from the specification.

To speed up this process, we further consider the automatic parsing of the domain-independent model. Processing the RDF notation of the AAS metamodel from the appendix of [1] could be a worthwhile effort here. We first need to extend the given RDF description to fill the full set of constraints. Second, we need to build a parser for the OWL and SHACL syntax from the Turtle serialization to satisfy the constraints of the AAS metamodel thus defined. We are looking positively forward to extend the given RDF representation, which could also lead to an improvement of the model back in the specification.

References

[1] Details of the Asset Administration Shell - Part 1: The exchange of information between partners in the value chain of Industrie 4.0
[2] Details of the Asset Administration Shell - Part 2: Interoperability at Runtime – Exchanging Information via Application Programming Interfaces (Version 1.0RC01)
[3] https://www.w3.org/TR/shacl/
[4] https://www.w3.org/TR/PR-rdf-syntax/Overview.html
[5] https://www.w3.org/TR/owl2-overview/
[6] https://github.com/admin-shell-io/aasx-package-explorer
[7] https://github.com/admin-shell-io/aasx-server
[8] https://github.com/aas-core-works/aas-package3-csharp
[9] https://github.com/admin-shell-io/aas-specs
[10] https://spinrdf.org/shacl-and-owl.html