Maturity Evaluation of SDKs for I4.0 Digital Twins

Nico Braunisch, Robert Lehmann, Martin Wollschlaeger
Institute of Applied Computer Science
TU Dresden
Dresden, Germany
{nico.braunisch, robert.lehmann, martin.wollschlaeger}@tu-dresden.de

Marko Ristin, Hans Wernher van de Venn
Institute of Mechatronic Systems
Zurich University of Applied Sciences (ZHAW)
Zurich, Switzerland
{rist, vhns}@zhaw.ch

Björn Otto
Institute for Automation & Communication
Otto von Guericke University
Magdeburg, Germany
bjoern.otto@ifak.eu

Tobias Kleinert
Chair of Information and Automation Systems
for Process and Material Technology
RWTH Aachen University
Aachen, Germany
kleinert@plt.rwth-aachen.de

2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA) | 979-8-3503-3991-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ETFA54631.2023.10275719

***Abstract*—Digital twins, the virtual representations of physical assets, processes or systems, are becoming increasingly important in cyber-physical systems. They are a foundational block of Industry 4.0, the movement to digitalize industrial processes.**

The Asset Administration Shell (AAS) has been established as the preferable model to capture interoperable digital twins in the context of Industry 4.0. As asset administration shells become more prevalent, so does the need for specialized software development kits (SDKs) to manage them.

The intertwined Industry 4.0 value chains are highly dependent on interoperability. The SDKs for AAS are often the direct interface between the components, and the malfunctioning of the SDKs leads to the breakage of the value chain. It is therefore important that we have the tools to determine how individual SDKs behave, and detect when they malfunction. In this work, we implement an approach to assessing the maturity of the SDKs for managing asset administration shells, and perform a thorough survey and evaluation of the existing SDKs.

***Index Terms*—Industrie 4.0, Digital Twin, Testing, Verification, Test Tool, Asset Administration Shell**

I. Introduction

Digital twins are reflections of physical assets, processes or systems in virtual space. They are an essential part of Industry 4.0 (I4.0), the fourth industrial revolution, characterized by the integration of digital technologies into industrial processes. The use of digital twins in I4.0 offers several benefits including improved efficiency, reduced costs and increased reliability. For example, digital twins can be used to monitor machine performance in real time, to predict failures before they occur and to optimize maintenance schedules to reduce downtime and increase productivity. They can also be used to help simulate different scenarios and identify potential improvements.

The Asset Administration Shell (AAS) has been developed and established as the main representation for digital twins in I4.0 [1]. AAS is thus the core component for providing interoperability between the partners in the I4.0 value chain.

The AAS models follow an agreed-upon meta-model by the Industrial Digital Twin Association (IDTA, [2]), one of the main organization behind I4.0. The meta-model defines the data structures, constraints, and semantics of the AAS models, providing the foundation for interoperability. The meta-model is published as a series of books [3], and there exist formalizations to a machine-readable format [4].

The information captured by AAS models can be serialized for exchange to different formats such as JSON and XML. Writing such serializations manually is possible, but is tedious and error-prone given the complexity of the AAS meta-model. For example, the formalized meta-model in Python programming language [4] includes 55 classes and the 89 AAS constraints (prefixed by "AASd-" in the specification) defined as invariants.

While the corresponding data schemas can be used to validate the data, such as XSD [5], the schema languages are limited and can not enforce many of the constraints specified in the AAS meta-model. Many humans errors slip in, resulting in unreadable data or even misleading information exchange.

To address that complexity, various *Software Development Kits* (SDKs), *i.e.*, extensive software libraries, have been developed to read, manage and serialize AAS models. The SDKs introduce an abstraction layer on top of the serialization in form of an object model in computer memory. This allows for better operations on the data as well as more comprehensive enforcement of the AAS meta-model constraints compared to the data schemas.

It is important that SDKs are correct. AAS are widely used in I4.0 and any changes in SDKs are thus high-leverage. For example, errors in SDKs propagate to many downstream clients, and *vice versa*. In contrast, further improved correctness has a strong positive impact on a large number of users. As they are the core component of I4.0 for providing interoperability, a malfunctioning SDK for AAS models directly translates to broken value chains and consequent negative effects in the physical world.

This makes the assessment of an SDK for AAS crucial when developing complex I4.0 systems. Currently, existing SDKs follow different design philosophies, and some do sacrifice interoperability and correctness for developer's ergonomics, speed of prototyping and custom-tailored non-standard fea-

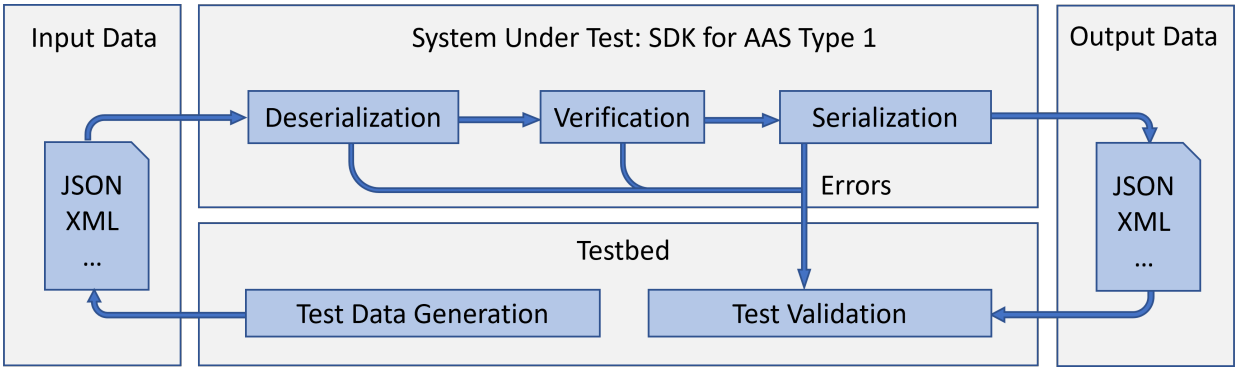

Fig. 1: Our Test Setting (from [6])

tures. In face of such trade-offs, the clients relying on an SDK should have a clear picture of its maturity and its feature scope. To produce systems of desired quality, the clients need means to evaluate an SDK and see if it fits their use cases before setting off the development.

Among different techniques for quality assurance in software engineering, *testing* is an established and widely-used approach [7]. Tests are customarily written by hand and executed automatically at every development iteration.

However, writing test cases manually requires a lot of effort, and this effort further grows with the complexity of the software. Frequent changes in the AAS meta-model coupled with the highly complex structures and large number of constraints make this associate this task with a lot of effort for an AAS SDK.

Automatic and semi-automatic testing is hence paramount, as already observed by Miny *et al.* in [6]. They realize that only through automatic generation of test cases a thorough testing of an SDK can be achieved. Though their work provides valuable insights, no concrete implementation has been attempted. Therefore, we present an implementation as shown in Fig. 1, which was used on the different available and applicable SDKs, frameworks and tools.

Our work clearly dwells on the ideas of Miny *et. al.*, but we take them a step further to realize an encompassing evaluation of the existing and future SDKs for managing AAS models. We contribute:

1) **A concrete implementation** of the proposition in [6],
2) **Exhaustive survey of the existing SDKs**, up to the extent of our knowledge,
3) **Evaluation and analysis** of the **maturity** and **feature scope** of the individual SDKs, and
4) **An extensive battery of generated test cases** that can be readily applied for bug discovery and interoperability testing of existing and novel SDKs.

As far as we are aware, no systematic attempt has been made in the literature to assess the SDKs of AAS, nor have any systematic approaches to their testing been proposed. However, it cannot be ruled out that at the time of publication other SDKs, frameworks and tools will be updated, which could allow for validation.

II. Related Work

Testing is a pillar of software engineering [7], and especially so for complex I4.0 systems consisting of many inter-operating components. The numerous test methods can be roughly split in two categories:

a) Black-box testing: where one ignores the implementation details of the *system under test* (SUT), and

b) White-box testing: where testing is based on the intricate knowledge of the SUT, *e.g.*, source code of a program.

In this work, we follow and refer to a high-level analysis of the testing methods in specific context of I4.0 digital twins presented in [6]. The variety of programming languages and environments which need to be considered in the context of I4.0 does not practically permit white-box testing. We take consequently the black-box approach. Furthermore, purely random black-box methods such as random and search-based testing [8], [9] are not applicable to our setting. The completely random test data violates the constraints of many classes in AAS meta-model most of the time, making for a wasteful and limited test approach. Instead, and again following [6], we generate the test data using a combination of:

c) Property-based Testing [10]: by scripting generation of the test data procedurally at random according to the constraints of the meta-model, where possible, and

d) Combinatorial Testing [11]: by using deterministic basic building blocks for situations where constraints are so complex that we could not procedurally fulfill them.

We point the reader to [6] for more details on testing in context of I4.0, and for a broader summary of general automatic testing methods for software to [7], respectively.

To the best of our knowledge, there is a very limited stakeholder-driven overview of the evaluation of existing SDKs for managing AAS models. Furthermore, individual projects were only sporadically tested by the respective development teams, but there was no overall quality control across different AAS SDKs, frameworks or tools. Considering scientific publications, the authors of [12] used the Basyx Python SDK to evaluate their test case generation approach for the AAS. However, they relied on the older version V3.0RC1 so that their results are difficult to compare to ours.

III. Approach

A. Test Procedure

When comparing the SDKs to evaluate, we found, that all of them offer at least these functions:

- **Deserialize** Parse a file or string containing a serialized AAS. This function outputs the parsed AAS in some form of internal representation like objects. The deserialization function accepts XML and JSON formatted inputs.
- **Validate** Check if the internal AAS representation is valid, *i.e.*, all constraints as defined by the specification are satisfied.
- **Serialize** Write the internal AAS representation to file or string. This function is the inverse of the deserialization, hence it supports writing to JSON and XML, respectively.

We exclude the processing of AASX files, as this would finaly only process XML and JSON files in a ZIP container. OPC and AML representations are also excluded as there are no official publication for these.

Our test procedure aims at invoking these functions in series to check if they are implemented correctly leading to the test setting shown in Figure 1.

For this, a set of valid test files is generated by the testbed. These valid files contain valid AAS. Therefore, they must pass the complete deserializtion-validation-serialization-chain without any errors. If so, it can be deduced that the SDK is implemented correctly. Furthermore, the testbed generates invalid test files containing invalid AASs. These files must be rejected by the deserialization or validation function of the SDK if the SDK is implemented correctly.

To execute the tests, we implemented a tiny test adapter for each SDK which takes a generated test file as input and invokes the deserialization, verification and serialization functions of the SDK. If any error is returned by these functions, the test adapter reports it accordingly. Consequently, we identified the following cases which can be reported by the test adapter:

- **INPUT_ERROR** An error has occurred on the I/O side when reading the test file. This status code is less relevant for the SUT, but is necessary in the broadest sense as a quality control for the test framework against sporadic errors, such as access violations or corrupt files.
- **DESERIALISATION_ERROR** This status code signals to the SUT that an error has occurred during deserialisation. Positive test cases should generally not reach this status code. For negative test-cases it is a valid action to notice these test-cases. This is usually the case when there is no explicit validation after deserialisation.
- **VALIDATION_ERROR** Validating the loaded model is a core part of every SDK and framework. However, not all software tools provide this step explicitly. In most cases, this step is implicitly performed during modification, deserialisation or serialisation. It can also be used to hold inconsistent models in memory for model transformations that need to be checked later.
- **SERIALISATION_ERROR** In the last step, the serialisation is performed by the SUT. If an error occurs during this step, a serialisation error is reported in the same way as during deserialisation.
- **OUTPUT_ERROR** In the last step, the serialisation is performed by the SUT. If an error occurs during this step, a serialisation error is reported in the same way as during deserialisation.
- **NO_ERROR** No errors occurred during the entire process of reading the test data, deserialising, validating, serialising and writing the result data. This is a pass case for positive test cases. The serialized output is written to disk for later comparison.
- **SERIALIZED_DO_NOT_MATCH** This report is created when the input and output of the test cases do not match. See Section III-C for more details.

To ensure reproducible test results, we installed SDKs from the appropriate package manager, if any. Otherwise we fall back to cloning a specific commit of the corresponding git repository. Please refer to Table I for details. Additionally, to avoid dependencies on the operating system, we installed the SDKs in dedicated Docker containers and executed the tests in them[1].

B. Test Data

The AAS meta-model V3.0RC02 [3] is fairly complex. It defines the data structures not only in terms of data types, but includes also many constraints with complex programming logic.

Aas-core-meta [4], [13], the formalization of the meta-model in Python language, lists 55 classes and 89 invariants implementing the meta-model constraints. A class has 8 ± 5 properties, with up to maximum 19 properties. For illustration, Listing 1 shows one formalized invariant of the class `Reference` and its involving constraint in Python language.

Listing 1: One fairly complex constraint of the AAS meta-model class `Reference` formalized as invariant in Python language in [4]

```
@invariant(
  lambda self:
  not (
    (
      self.type
      == Reference_types.Model_reference
    )
    and len(self.keys) > 1
    and (
      self.keys[-1].type
      == Key_types.Fragment_reference
    )
  ) or (
    self.keys[-2].type == Key_types.File
    or self.keys[-2].type == Key_types.Blob
  ),
  "Constraint AASd-127: For model references "
  "with more than one key, a key with type "
  "Fragment reference shall be preceded by "
  "a key with type File or Blob."
)
```

[1]All tests were last executed on 04/20/2023

Writing test data at such scale by hand is obviously tedious and error-prone. On the other hand, generating the test data fully automatically is equally daunting due to the complex logic of the constraints. Preliminary experiments with Z3 [14], a popular solver, revealed many difficult obstacles arising from array access and regular expressions, whose details we leave for future work. This leads us to semi-automatic approach as outlined in [6], where we combine some automatic generation with manually programmed procedures.

We outline here the generation procedure in abstract. We refer to our open-source code [15] for the specific details.

Positive and negative examples. We distinguish between *positive* and *negative* examples, following [6]. Positive examples (`ACCEPT` cases in [6]) are **valid** instances of an AAS model. These examples are expected to successfully deserialize, validate with an SDK, and serialize back to the original form. Afterwards, we mutate positive examples to obtain the negative ones. One part of the negative examples are **unserializable** (`REJECT-SCHEMA` in [6]), so that an SDK is not expected to deserialize them from a file into memory. The other part is **invalid**. The invalid examples can be deserialized, but violate one or more constraints and are expected to be rejected by SDK at the verification stage (`REJECT-META` in [6]).

Generation of positive examples. For every class of the meta-model, we write two different generation procedures. One procedure generates an instance with *all* the properties set, analogue to `ACCEPT-MAX` case proposed in [6]. We define another procedure to generate a *minimal* instance, with only the required properties set.

The procedures start by sampling values for each property, according to "Property-based Testing" as described in [6], [10]. For deterministic but random generation, we hash the paths to the property values relative to the root environment instance, and use the hashes as randomization seeds. Some classes lack constraints, so random values are already satisfactory, and the procedure finishes. However, for the classes with constraints, we have to *fix* the random data. We do that by manually scripting the recursive fixes on a class-by-class basis. The fixing procedure starts with the random data and then computes or unsets certain properties. We arbitrarily designed the fixing procedures until the resulting instances were easy to inspect and did not violate any constraints. For example, the class `Entity` mandates that either `specificAssetId` or `globalAssetId` are set if the entity type is set to self-manage. Therefore, whenever we fix a self-managed `Entity`, we arbitrarily unset the property `specificAssetId` to satisfy its invariants. In total, fixing procedures for 13 classes were necessary.

For classes with very complex constraints `Reference` and `SubmodelElementList` the fixing code itself turned out to be fairly difficult to develop. We implemented generation procedures for these two special cases entirely by hand. This is also in line with "Combinatorial Testing" paradigm as suggested in [6].

The details are available in our open-source code [15].

Recursion. When a property needs to be generated which is typed by a meta-model class, we simply sample a minimal instance of that class. In case of lists, we sample a list with a single minimal item. This allows us to generate "deep", but readable, instances recursively.

Edge cases. The (semi-)random generation of test data misses a lot of important edge cases by its mere stochastic nature. We select a set of relevant edge cases based on our experience with the data observed "in the wild". We then follow "Combinatorial Testing" devised in [6] and manually write procedures to generate the corresponding instances. Specifically, the following cases are covered.

- All possible `valueType`'s over classes `Extension`, `Property`, `Qualifier` and `Range`, as we observe that a mismatch between values and value types is a common cause of invalid models,
- Different compositions of keys over different kinds of `Reference`'s (namely, global and model references),
- Various settings of `SubmodelElementList` over different values for properties `semanticId-ListElement`, `typeValueListElement` and `valueTypeListElement` and the corresponding elements of the submodel element list.

Environment. According to the meta-model, the model data is expected to live nested in the root instance of class `Environment`. Hence, all the examples need to be nested in an environment and the corresponding enclosing instances. For example, an `AdministrativeInformation` lives nested in a `Submodel`, which is then nested in the property `submodels` of the `Environment`.

To generate the desired environment, we first represent the composability relations between classes in a directed graph. Each node represents a class, where an edge is drawn between a source and a target class if the target class appears as a type of a source class' property (or an item type of a list). For compact environments, we compute Dijkstra's shortest path [16] from `Environment` to each class, and generate the minimal nesting instances along the path. The generated example is nested at the final destination.

Unserializable examples (`REJECT-SCHEMA`). To generate examples not serializable by an SDK, we pick a positive example of a class and mutate its properties systematically in different scenarios:

- If a property is required, it is removed to test for requirement checks.
- If the property is of type string, it is set to an empty list. Otherwise, it is set to a string. This tests the type checking.
- If the property is an enumeration, we set it to an invalid random value not corresponding to any enumeration literals to test the handling of invalid enumerations.

Additionally, we insert an unexpected property to test the rejection of additional properties.

Invalid examples (`REJECT-META`). The complexity of the constraints does not allow for automatic generation of the test

data so we can not explore the space of invalid examples extensively. Instead, we manually write procedures to test for some of the prominent settings. Similar to unserializable examples, we mutate a large example of a class. The following scenarios are systematically covered:

- Properties constrained by a string pattern are replaced by a string violating the pattern. The invalid strings are obtained both by Rejection Sampling [17] and manual design. There were in total only 5 patterns in the meta-model.
- For properties whose length is constrained, we randomly generate a value that is either shorter than the minimum value or longer than the maximum length. The meta-model only uses strings, byte arrays and lists of instances. In the first two cases we randomly generate values of the desired length. In case of lists, we generate lists of the desired length and sample the corresponding number of minimal item instances.
- Properties representing a timestamp are set to an invalid date (February 29th, 2022, where 2022 is not a leap year).
- Properties constrained to a set of predefined values are set to an invalid value generated by Rejection Sampling.

We additionally script the following edge cases:

- Invalid `value` property not corresponding to `valueType` in instances of `Extension`, `Property`, `Qualifier` and `Range`.
- The value of `min` being greater than `max` in instances of `Range` over possible `valueType`.

Additionally, given the prominence of classes `Reference` and `SubmodelElementList` "in the wild", we manually write test cases for them since their complex constraints elude automation.

Figure 2 illustrates the overall generation approach.

C. JSON and XML Comparison

In case of positive test cases, the Software Development Kit (SDK) should be able to deserialize, validate and serialize the input successfully. In this case, an output file is generated. However, there may be a flaw in the SDK tested that causes the output file to be different from the input file. To detect such bugs, we need to compare the output file to the corresponding input file.

For this, however, a byte-wise comparison is not sufficient, because the output formats (JSON, XML) allow for some ambiguities:

- Both formats accept an arbitrary number of separating whitespaces between language elements.
- In JSON, the ordering of keys in objects is arbitrary.
- In XML, the ordering of attributes is arbitrary.
- In JSON, a number can be formatted in various ways.

Consequently, we implement a comparison algorithm for these input formats. It normalizes the input and output files such that the above ambiguities do not exist anymore. This is done by sorting keys and attributes alphabetically, stripping duplicate whitespaces, etc.

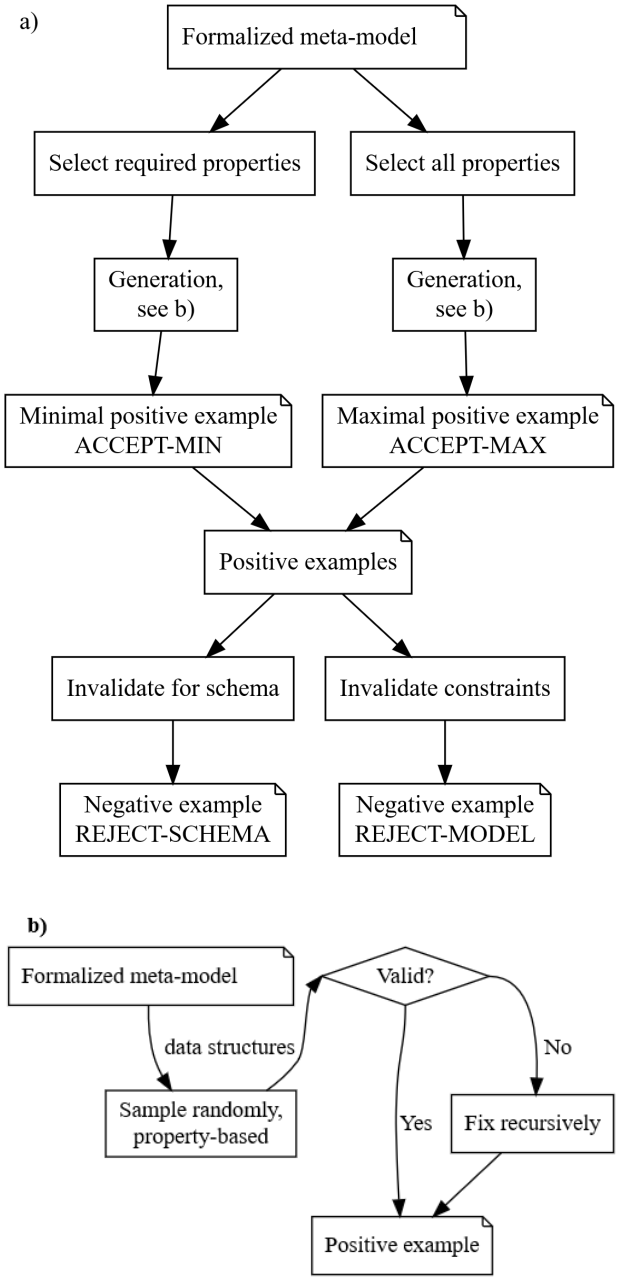

Fig. 2: **a)** Overview of our approach to generate test data based on AAS meta-model [4], and **b)** Generation of positive examples in more detail

IV. Evaluation

In our evaluation, we consider all relevant SDKs, frameworks, and software tools that are freely available. Relevant in this context means that the product has reached a certain level of widespread use, functional scope, and technological maturity. We do not consider products that can only map version 2 of AAS, as it is now obsolete, and only evaluate software for the current version 3.0RC02. Table I lists the evaluated SDKs.

TABLE I: Evaluated and tested SDKs.

SDK	Language	Version/Revison	Tested	Repository
AAS4J	Java	1.0.0-milestone-03	✓	github.com/eclipse-aas4j/aas4j
BaSyx Java SDK	Java	1.4.0	✓	github.com/eclipse-basyx/basyx-java-sdk
BaSyx Python SDK	Python	0.2.2	✓	github.com/eclipse-basyx/basyx-python-sdk
BaSyx DotNet SDK	C#	1.0.0		github.com/eclipse-basyx/basyx-dotnet-sdk
BaSyx Rust SDK	Rust	0.1.0		github.com/eclipse-basyx/basyx-rust-sdk
BaSyx C++ SDK	C++	3cddc1a		github.com/eclipse-basyx/basyx-cpp-sdk
aas-core3.0rc02-csharp	C#	2.0.0-rc2	✓	github.com/aas-core-works/aas-core3.0rc02-csharp
aas-core3.0rc02-golang	GoLang	07948f7		github.com/aas-core-works/aas-core3.0rc02-golang
aas-core3.0rc02-typescript	TypeScript	1.1.1	✓	github.com/aas-core-works/aas-core3.0rc02-typescript
aas-core3.0rc02-python	Python	2022.6.21	✓	github.com/aas-core-works/aas-core3.0rc02-python
AASX Package Explorer	C#	9a97978	✓	github.com/admin-shell-io/aasx-package-explorer
AASX Server	C#	9573e2d	✓	github.com/admin-shell-io/aasx-server
NOVAAS	JavaScript	658e09b3		gitlab.com/novaas/catalog/nova-school-of-science-and-technology/novaas

A. AAS Frameworks considered and tested

The following section gives an overview of all the AAS tools that we could find to the best of our knowledge. We also give the reasons why they were included or excluded from our evaluation.

a) aas4j: is a Java library that implements the AAS specification and includes serialization and deserialization modules, validators, and transformation libraries. It includes all classes and properties as defined in [3]. The library is available from the Maven Central Repository. We tested the provided Maven artifact which corresponds to the main branch of the repository. However, there are several branches listed on the project's GitHub page reflecting different stages of development. Accordingly, we rely on the `dataformat-json` and `dataformat-xml` artifacts, and in particular the deserialization and serialization methods. At the time of writing, the validation methods were not yet implemented, as the project is still in transition to version 3 of the AAS specification.

b) basyx-java-sdk: Eclipse *BaSyx* is the open source middleware for Industry 4.0 developers. BaSyx emerged from the research project Basys for I4.0 and has been continuously developed. BaSyx implements all necessary standards and infrastructure components for I4.0 production environments and provides the basis for software platforms. We have not tested BaSyx as a whole, as this is far beyond the scope of our current work. It is important to note that BaSyx is more than just a collection of SDKs for different languages. It provides a complete ecosystem in itself, including in particular its own bus system. At the time of writing, the various SDKs included in BaSyx have not been generated from a single source, but have been reimplemented for all supported languages. This is certainly a reason for the wide range of maturity in the different BaSyx SDKs.

BaSyx provides a software development kit written in Java within a wider BaSyx framework. It aims to support key Industry 4.0 concepts such as AAS, OPC-UA and REST APIs, enabling developers to quickly build and deploy I4.0 applications on a variety of devices and platforms. Basyx-java-sdk is planned to leverage aas4j in the next major release, which is expected to support V3.0 of the AAS [18]. We therefore ignore it from the evaluation, and test only aas4j.

c) basyx-dotnet-sdk: Basyx also has a DotNET framework based on C#. However, this is being developed separately and independently of the Java SDK development team. It does not yet support version 3.0 of the AAS, and we thus ignore it in the evaluation.

d) basyx-c++-sdk: Furthermore, an attempt was made to provide a C++ SDK within Eclipse Basyxs. However, only a wrapper for the basic functions of the Java SDK are provided, so we do not separately test this SDK.

e) basyx-rust-sdk: The Eclipse Basyxs Rust SDK is currently still under development and therefore cannot be evaluated yet.

f) basyx-python-sdk: The Eclipse Basyxs Python SDK has been continuously developed over many years, starting as a separate project from Basyx and eventually merging. Unlike all other Basyxs SDKs, it provides no virtual bus for communication due to its independent development.

g) aas-core-csharp: AAS Core Works is an informal group that provides a set of frameworks and SDKs for working with AAS in I4.0 systems. The software includes APIs and tools for creating, editing and managing AAS files and for integrating AAS-related functions into other software systems. AAS Core Works supports important I4.0 concepts such as semantic data modelling, information exchange and interoperability. The goal of the group is to accelerate development cycles, improve developers' workflows and create robust, scalable and interoperable I4.0 systems. The libraries are flexible and modular, so developers can easily integrate it into their existing software infrastructure and adapt it to their specific requirements. Special emphasis is placed on software correctness and development speed, achieved through sophisticated software engineering such as code generation by transpilation. The aas-core-csharp is the first SDK in this series.

h) aas-core-python: Shortly after the release of the C# SDK, a Python SDK was released with minimal effort, as the project leveraged the bulk of work invested in the transpiler. The main benefits of such an approach came to fruition, namely the feature parity and correctness of the C# SDK, while the code followed the style amenable to Python developers.

i) aas-core-typescript: In order to include web developers in the world of AAS, a Type Script SDK was provided by AAS Core Works. This also forms the basis for the AAS React Editor made by the same group. At the time of evaluation, development was not yet complete.

j) aas-core-golang: The Go SDK is currently work-in-progress, so it could not be considered at the time of the evaluation.

k) aasx-package-explorer: The AAS Package Explorer is a software tool that allows end users to create, edit and view AAS files. It provides a single user interface for working with AAS files. The tool also supports the import and export of AAS files in various formats, facilitating sharing and collaboration on AAS-related projects. The AAS Package Explorer was one of the first reference implementations of the AAS. As such, it is one of the most widely used implementations that has been manually implemented over several years and versions of the AAS.

l) aasx-server: The AAS server is an essential component of I4.0, which enables the provision and management of AAS in I4.0 systems. The AAS Server provides a platform for hosting and managing AAS instances, enabling users to discover, browse and interact with AAS objects in a standardized way. It supports the deployment of AAS instances on various hardware and software platforms, including cloud servers, edge devices and data centres. The original version of the AAS Server used the same meta-model implementation as the AAS Package Explorer. Due to the different aims of the two projects (user interface *versus* backend software), aasx-server is integrating aas-core-csharp as the underlying SDK by copying and modifying the code. Therefore, we evaluate it separately from aas-core-csharp, although the output is expected to be the same.

m) NOVA Asset Administration Shell: The NOVA Asset Administration Shell (NOVAAS) is an open source reference implementation and runtime environment for the AAS concept developed by the NOVA School of Science and Technology. The solution has been designed and developed using best practices from IT engineering. The core is implemented in Javascript and uses node-red as the development environment and node.js as the runtime environment. The exchange data is formatted as JSON and packed into AASX packages. For this reason, NOVAAS was not tested, as the test data are only XML and JSON files.

B. Test procedure and assessment measure

The evaluation was carried out as described in Chapter III. As described in Section III-A a, the frameworks, SDKs and software tools described in Section IV-A were embedded in the test setup, see Figure 1.

1) Quality and Quantity of Input Files of Testcases: Our generator produced 9097 input files for the test cases. It generated 4718 JSON files and 4379 XML files. In the case of JSON 2556 files for expected test cases as positive examples and 2540 in the case of XML files. These cases shall be processed with status code **NO_ERROR** in the test framework. In addition, 2162 JSON and 1839 XML files for negative examples were generated as unexpected test cases. These test cases shall not exit with status code **NO_ERROR**. Therefore, these test runs are expected to terminate with status code **DESERIALISATION_ERROR** or **VALIDATION_ERROR**.

When ever an output file is created, the input and output are compared semantically as described in section III-C. If the output file and input file are not equal in this comparison the status code **SERIALIZED_DO_NOT_MATCH** will be set for the test case.

2) Applied metrics: A simple metric is used to assess the evaluation. For this purpose, the compliance factor of the respective SUT is determined. For each positive test case, one point is awarded if the test case is completed in the test system with a status code of **NO_ERROR** and if the input and output files are semantically identical. For negative test cases, one point is awarded for status codes **DESERIALISATION_ERROR** and **VALIDATION_ERROR**. The scores of the positive and negative test cases are then added together. The compliance factor is determined by dividing the sum of the points by the number of positive and negative test cases. The following formula represents the compliance factor Fc:

$$Fc = \frac{\text{Points pos. test cases} + \text{Points neg. test cases}}{\text{pos. test cases} + \text{neg. test cases}}$$

C. Results

The evaluation of the SDKs, frameworks and tools presented in A, using the assessment performed in section III and the metrics defined in section IV-B1 and IV-B2, led to the evaluation results shown in Table II for JSON and in Table III for XML.

TABLE II: Compliance factors Fc of the SDKs with JSON testfiles

SDK	Points. pos. test cases	Points. neg. test cases	Fc
AAS4J	2348	509	0.61
BaSyx Java SDK	2532	250	0.59
BaSyx Python SDK	2102	2162	0.90
aas-core3.0rc02-csharp	2556	2162	1.00
aas-core3.0rc02-typescript	2556	2130	0.99
aas-core3.0rc02-python	2556	2162	1.00
AASX Package Explorer	263	2162	0,51
AASX Server	2556	2162	1.00

TABLE III: Compliance factors Fc of the SDKs with XML testfiles

SDK	Points. pos. test cases	Points. neg. test cases	Fc
AAS4J	2275	574	0.65
BaSyx Java SDK	2532	250	0.64
BaSyx Python SDK	0	1839	0.42
aas-core3.0rc02-csharp	2540	1839	1.00
aas-core3.0rc02-typescript[2]	0	0	0.00
aas-core3.0rc02-python	2540	1839	1.00
AASX Package Explorer	17	1839	0.42
AASX Server	2428	1839	0.97

[2]No XML support implemented

D. Discussion

The results were very different for the different SDKs, frameworks and tools.

When looking at JSON and XML test cases, it is noticeable that the implementation of JSON has a significantly higher conformance factor than that of XML. One reason for this could be that JSON is preferred by software developers or that the JSON specification is also used for the AAS REST API [19].

Considering SDKs, frameworks and tools with a low compliance factor, a closer look reveals that one reason for the results could be the current state of development. This concerns both the software and the specification side. Due to the current development of the release candidate of the specification, not all requirements and constraints could be implemented or priority was given to the implementation of the final version 3.0.

The number of failures for negative test cases is over all very low. Looking at the results of the SUT, which received very few points for negative test cases, it can be seen that these are mostly errors in checking **VALIDATION_ERROR** or this step is not performed propperly. We suspect that since many SDKs only document the meta-model constraints rather than implementing them into actual code, the reason for the SUTs failing these test cases is the lack of the validation in the serialisation or deserialsation phase.

If $Fc = 1$ is achieved for the negative test cases, but hardly any points are achieved for the positive test cases, no positive overall assessment can be made for the SUT. This is because the missing functionality of the positive test cases also calls into question the functionality of the negative test cases.

V. Conclusion

In this paper we have proposed how to evaluate and test different SDKs, frameworks and tools for managing interoperable digital twins in I4.0 in form of AAS. Our tests have shown that verification of conformity with the AAS meta-model is necessary given the generally low compliance factors. Furthermore, we have observed that there is still room for improvement.

We see one possible solution to mitigating these failures in model-driven development of SDKs, such that the code can be adjusted at each update of the meta-model automatically.

In the future, we plan to re-evaluate the SDKs once the major version 3.0 of the AAS meta-model has been released, and the SDKs, frameworks and tools adopted it. Furthermore, we want to extend our evaluation to integration tests with a wider family of I4.0 products.

At the point of adapting the test cases for the next versions of the meta-model, we note the main limitations of our approach: 1) the manual effort involved in the fixing procedures and 2) scripting generation of edge cases. Due to restricted available time and focus, we cover only the obvious cases (2), while many other remain uncovered, resulting in sub-optimal test coverage of the meta-model.

To address 1) and 2) in the future, we plan to explore alternative solutions to rely less on programmer's work. In particular, we want to use solvers such as Z3 [14] to fully automatize test data generation. We hope that Z3 could help us reach much better coverage (2), while substantially reducing the manual labor (1).

Additionally, we want to develop a reactive visual application for generating test data to aid the development of the future meta-models.

Finally, we would like to test how our approach generalizes to other non-AAS meta-models within the field of automation.

References

[1] A. Braune, C. Diedrich, S. Grüner, G. Huettemann, , and Others, "Usage view of asset administration shell," tech. rep., 03 2019.

[2] "Industrial Digital Twin Association (IDTA)." https://industrialdigitaltwin.org. [Accessed 2023-4-20].

[3] S. Bader, E. Barnstedt, H. Bedenbender, and Others, *Details of the Asset Administration Shell. Part 1 The exchange of information between partners in the value chain of Industrie 4.0 (Version 3.0RC02)*. 05 2022.

[4] "Source code of aas-core-meta." https://zenodo.org/record/7807680. [Accessed 2023-4-20].

[5] "Schemas of the asset administration shell." https://github.com/admin-shell-io/aas-specs. [Accessed 2023-4-20].

[6] T. Miny, S. Heppner, I. Garmaev, T. Kleinert, M. Ristin, H. W. Van De Venn, B. Otto, K. Meinecke, C. Diedrich, N. Braunisch, *et al.*, "Semi-automatic testing of data-focused software development kits for industrie 4.0," in *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*, pp. 269–274, IEEE, 2022.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, 2013.

[8] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communication of the ACM*, vol. 33, no. 12, 1990.

[9] T. Y. Chen, "Adaptive random testing," in *The Eighth International Conference on Quality Software*, 2008.

[10] D. R. MacIver, "In praise of property-based testing." https://increment.com/testing/in-praise-of-property-based-testing/, 2019. [Online; accessed 25-March-2022].

[11] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[12] B. Otto and T. Kleinert, "A flow graph based approach for controlled generation of aas digital twin instances for the verification of compliance check tools," in *IECON 2022–48th Annual Conference of the IEEE Industrial Electronics Society*, pp. 1–6, IEEE, 2022.

[13] N. Braunisch, M. Ristin-Kaufmann, R. Lehmann, and H. W. van de Venn, "Generative and model-driven sdk development for the industrie 4.0 digital twin," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–4, 2021.

[14] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), p. 337–340, Springer-Verlag, 2008.

[15] "Source code of aas-core3.0rc02-testgen." https://zenodo.org/record/7828720. [Accessed 2023-4-20].

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] L. Devroye, *Non-Uniform Random Variate Generation*. New York, USA: Springer, 1984.

[18] "GitHub issue commenting the relation between basyx-java-sdk and aas4j." https://github.com/eclipse-basyx/basyx-java-sdk/issues/153. [Accessed 2023-4-20].

[19] S. Bader, B. Berres, B. Boss, A. Gatterburg, , and Others, "Details of the asset administration shell. part 2 - interoperability at runtime - exchanging information via application programming interfaces (version 1.0rc02)," tech. rep., 11 2021.