Sicherstellung von Konformität mithilfe von formalen Spezifikationen am Beispiel der Verwaltungsschale

Von der Fakultät für Georessourcen und Materialtechnik der
Rheinisch-Westfälischen Technischen Hochschule Aachen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von

Björn Otto, M.Sc.

Berichtende:	Univ.-Prof. Dr.-Ing. Tobias Kleinert
	Univ.-Prof. Dr.-Ing. Hartmut Zadek

Tag der mündlichen Prüfung: 26.11.2025

Vorwort

Ich danke allen, die zur Entstehung und Umsetzung dieser Arbeit beigetragen haben.

Mein größter Dank gilt Herrn Prof. Dr.-Ing. Tobias Kleinert, Inhaber des Lehrstuhls für Informations- und Automatisierungssysteme für die Prozess- und Werkstofftechnik der RWTH Aachen, für die vielen angenehmen Gespräche und fachlichen Diskussionen, die den Fokus meiner Arbeit immer wieder geschärft und mich für die besonderen Anforderungen der Automatisierungstechnik sensibilisiert haben.

Besonders herzlich möchte ich mich bei meinem Zweitbetreuer Herrn Prof. Dr.-Ing. Hartmut Zadek, Leiter des Lehrstuhls für Logistik, Institut für Engineering von Produkten und Systemen an der Otto-von-Guericke-Universität, bedanken, der mich bei der Durchführung dieser Arbeit sowohl persönlich als auch organisatorisch sehr unterstützt hat.

Außerdem möchte ich mich bei den Mitgliedern des AAS Core Projektes bedanken, insbesondere bei Herrn Dr. Marko Ristin und Herrn Dipl.-Inf. Nico Braunisch, die mit vielen Gesprächen meine Arbeit auf inhaltlicher wie auch auf technischer Ebene vorangebracht haben.

Ferner gilt mein Dank allen Mitarbeitern des Lehrstuhls für Informations- und Automatisierungssysteme für die Prozess- und Werkstofftechnik der RWTH Aachen für die äußerst hilfreiche Umsetzung des Forschungsseminars. Insbesondere danke ich den Mitgliedern des Fachbereichs semantische Beschreibungsmittel (in alphabetischer Reihenfolge) Herrn Igor Garmaev, Herrn Sebastian Heppner sowie Herrn Dr.-Ing. Torben Miny. Die vielen wertvollen Diskussionen halfen meinem Verständnis rund um die Verwaltungsschale, förderten immer wieder den kritischen Austausch mit meiner Arbeit und waren auch auf persönlicher Ebene wertvoll.

Auch möchte ich mich bei der Industrial Digital Twin Association bedanken, insbesondere bei Herrn Dr. Christian Mosch und bei Frau Juilee Tikeka, zum einen für die Möglichkeit, die *AAS Test Engines* in dieser Arbeit zu verwenden, zum anderen für die vielen wertvollen Kontakte ins Ökosystem der Verwaltungsschale. Des Weiteren danke ich Herrn Prof. Dr.-Ing. Christian Diedrich, Inhaber des Lehrstuhls für Integrierte Automation an der OVGU Magdeburg, für die organisatorische Unterstützung dieser Arbeit. Weiterhin danke ich dem Freundeskreis des Lehrstuhls für Informations- und Automatisierungssysteme für die Prozess- und Werkstofftechnik für die finanzielle Unterstützung während der Entstehung dieser Promotion.

Abschließend möchte ich mich besonders bei meiner Ehefrau Anna Isabell für ihren kritischen Blick und bei meinem Sohn Frederick Tim für seine Motivation bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungen

AAS	Asset Administration Shell (Verwaltungsschale)
AASX	Paketformat für die Verwaltungsschale
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CRUD	Create-Read-Update-Delete
CD	Concept Description
CSV	Comma-Separated Values
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDTA	Industrial Digital Twin Association
IoT	Internet of Things
IRDI	International Registration Data Identifier
JSON	JavaScript Object Notation
OCL	Object Constraint Language
OPC	Open Platform Communications
	Open Packaging Conventions
OWL	Web Ontology Language
REST	Representational State Transfer
RDF	Resource Description Framework
SDK	Software Development Kit
SUT	System under Test
TRL	Technology Readiness Level
UML	Unified Modeling Language
URL	Uniform Resource Locator
WG	Working Group
VWS	Verwaltungsschale
XML	Extensible Markup Language
XSD	XML Schema Definition

Kurzfassung

Interoperabilität ist eines von drei Kernzielen für Industrie 4.0 [16]. Eine wesentliche Voraussetzung für Interoperabilität ist Konformität. Um Konformität sicherzustellen, haben sich Konformitätstests etabliert, d.h. das zu überprüfende System wird mit einem Satz an vordefinierten Testfällen untersucht. Wenn all diese Testfälle positiv ausfallen, wird das System als konform angesehen. Im Rahmen von Industrie 4.0 wurde die Verwaltungsschale etabliert, ein Standard für digitale Zwillinge. Allerdings fehlen für die Verwaltungsschale bisher solche Konformitätstests [3, 4, 8, 17]. Im Rahmen dieser Arbeit wird daher ein Testsystem vorgestellt, das in der Lage ist, Konformitätstests für Implementierungen der Verwaltungsschale durchzuführen. Parallel zur Definition der eigentlichen Konformitätstests wird die Frage untersucht, inwieweit formale Spezifikationen das Erstellen von Konformitätstests unterstützen und beschleunigen können. Wesentlich hierfür ist das Konzept, dass die Konformitätstests nicht *manuell* definiert werden, sondern *automatisiert* aus den formalen Spezifikationen generiert werden. Als formale Spezifikationen werden dabei JSON-Schema [18] (zur Beschreibung des Metamodells) und OpenAPI [19] (zur Beschreibung von reaktiven Verwaltungsschalen) gewählt. Diese werden als Teil der menschenlesbaren Spezifikation der Verwaltungsschale veröffentlicht.

Zunächst wird das Ökosystem rund um die Verwaltungsschale analysiert. Dadurch werden die zentralen Softwarekomponenten identifiziert, die wesentlich verantwortlich sind für die Interoperabilität im Ökosystem der Verwaltungsschale. Basierend auf dieser Analyse werden drei dieser Komponenten ausgewählt, deren Konformität im Rahmen dieser Arbeit durch Konformitätstests sichergestellt werden soll. Dabei handelt es sich um passive Verwaltungsschalen, Verwaltungsschalen-SDKs sowie reaktive Verwaltungsschalen. Dazu werden Verfahren vorgestellt, um die Konformität dieser Komponenten sicherzustellen. Durch die Verwendung von formalen Spezifikationen bei allen drei Verfahren kann der manuelle Aufwand um etwa 40% gesenkt werden. Darüber hinaus stellt die automatische Generierung von Testfällen sicher, dass die Testfälle vollständig sind, in dem Sinne, dass alle Aspekte des Metamodells bzw. der API abgeprüft werden. Es zeigen sich allerdings auch Nachteile wie die teils erschwerte Lesbarkeit der Testergebnisse sowie nötige Anpassungen um z.B. die Constraints des Metamodells abbilden zu können.

Die beschriebenen Verfahren wurden in drei Softwaretools (TRL 6-7) umgesetzt. Mithilfe dieser Tools werden im Rahmen dieser Arbeit 3 SDKs und 4 Serverimplementierungen auf Konformität überprüft. Bei den SDKs ist die Konformität bereits recht hoch, bei den Serverimplementierungen bestehen allerdings teilweise noch zahlreiche Defizite bezüglich der Konformität. Die im Rahmen dieser Arbeit vorgestellten Konformitätstests werden beim Entwickeln von Software für das Ökosystem der Verwaltungsschale einen hilfreichen Leitfaden bieten, um die Konformität in Zukunft weiter zu erhöhen und schlussendlich die geforderte Interoperabilität zu erreichen [9].

Abstract

Interoperability is one of the three core goals of Industry 4.0 [16]. A key prerequisite for interoperability is conformance. To ensure conformance, conformance tests have become established, i.e., the system under test is evaluated using a set of predefined test cases. If all these test cases pass, the system is considered conformant. As part of Industry 4.0, the Asset Administration Shell (AAS) was established, a standard for digital twins. However, such conformance tests are still missing for the Asset Administration Shell [3, 4, 8, 17]. Therefore, this work presents a test system capable of conducting conformance tests for implementations of the AAS. In parallel with defining the actual conformance tests, this work also explores the extent to which formal specifications can support and accelerate the creation of conformance tests. A key concept here is that the test cases are not defined *manually*, but instead *automatically* generated from formal specifications. The formal specifications used are JSON Schema [18] (to describe the metamodel) and OpenAPI [19] (to describe reactive Asset Administration Shells). These are published as part of the human-readable AAS specification.

First, the ecosystem surrounding the Asset Administration Shell is analyzed. This analysis identifies the central software components that are crucial for interoperability within the AAS ecosystem. Based on this analysis, three components are selected whose conformance has to be ensured in this work through conformance testing: These are passive AASs, AAS SDKs and reactive AASs. Different methods are presented to verify the conformance of these components. By using formal specifications in all three methods, manual effort can be reduced by approximately 40%. Moreover, the automatic generation of test cases ensures their completeness, meaning that all aspects of the metamodel or API are tested. However, some disadvantages are also observed, such as reduced readability of test results and necessary adjustments to, for example, map constraints of the metamodel.

The described methods have been implemented in three software tools (TRL 6–7). Using these tools, three SDKs and four server implementations are tested for conformance in this work. While the SDKs already show a high level of conformance, the server implementations still exhibit significant deficits in this regard. The conformance tests introduced in this work will provide a helpful guideline for developing software within the AAS ecosystem, in order to further improve conformance and ultimately achieve the required interoperability [9].

1 Einleitung

In ihrem „Leitbild 2030 für Industrie 4.0“ [16] definiert die Plattform Industrie 4.0 drei Kernziele, um zukünftig die Wettbewerbsfähigkeit der deutschen Industrie gewährleisten zu können. Diese Kernziele sind Souveränität, Nachhaltigkeit und Interoperabilität. Sie bilden damit die drei Pfeiler von Industrie 4.0. Letzteres Ziel, die Interoperabilität, soll die „flexible Vernetzung unterschiedlicher Akteure zu agilen Wertschöpfungsnetzen“ [16, Seite 5] ermöglichen. Die Idee ist, dass Wertschöpfungsketten nicht mehr isoliert betrachtet werden. Vielmehr sollen hoch-dynamische Prozesse Wertschöpfungsketten ermöglichen, die übergreifend zwischen verschiedenen Herstellern und Branchen agieren. Dies ermöglicht die Schaffung neuartiger Produkte und Dienstleistungen.

Wesentliche Herausforderung zum Erreichen von Interoperabilität ist das Schaffen und Etablieren gemeinsamer Standards zum herstellerübergreifenden Austausch von Daten und Prozessen. Mit der Verwaltungsschale [3, 8] wurde ein solcher Standard geschaffen. Die Verwaltungsschale (VWS, *engl.* Asset Administration Shell, AAS) dient dazu, Assets zu beschreiben. Ein Asset ist eine werthaltige Entität (z.B. Produktspezifikation, Herstellungsprozess, Produkt, Produktionsressource etc.), die in einer gewissen Weise Teil einer Wertschöpfungskette sein kann. Die Verwaltungsschale wird dazu genutzt, um Informationen von Assets abzubilden und über vereinheitlichte Schnittstellen auf diese Informationen zuzugreifen. Dies gilt nicht nur für Informationen, die während der eigentlichen Herstellung anfallen. Vielmehr ermöglicht die Verwaltungsschale das Abbilden aller Phasen eines Lebenszyklus [20] eines Assets.

Seit der Veröffentlichung des Verwaltungsschalenstandards [3, 4, 8, 17] wurde ein Vielzahl von Software für die Verwaltungsschale implementiert. Dies umfasst verschiedene Aspekte des Standards, diverse Plattformen und Programmiersprachen sowie verschiedene Hersteller. Die entstandene Heterogenität führt dazu, dass die verfügbaren Implementierungen der Verwaltungsschale den Standard in Details verschieden umsetzen. Dies hat verschiedene Ursachen: Teilweise ist die Spezifikation der Verwaltungsschale zu ungenau oder sehr komplex, was zu Fehlimplementierungen führen kann. Außerdem haben verschiedene Hersteller unterschiedliche Anforderungen, was zu gewollten und ungewollten Abweichungen von der Spezifikation führt.

Die zahlreichen Abweichungen vom Standard führen schlussendlich dazu, dass die Implementierungen nicht mehr vollumfänglich interoperabel sind. Da dies eine zentrale Anforderung von Industrie 4.0 im Allgemeinen bzw. der Verwaltungsschale im Speziellen ist, sind somit Maßnahmen zu etablieren, um die Interoperabilität der Implementierungen zu erhöhen. Eine wesentliche Voraussetzung für Interoperabilität ist Konformität zur Spezifikation der Verwaltungsschale. Zur Sicherstellung der Konformität haben sich in der Softwareentwicklung Konformitätstests [21] etabliert. Dies bezeichnet einen Satz von Testfällen, die jeweils einzelne Aspekte der Zielspezifikation exemplarisch überprüfen. Besteht ein System alle Testfälle, dann kann es als konform zur Spezifikation angesehen werden.

Ein Testsystem zur Überprüfung der Konformität muss einige Anforderungen erfüllen [9]:

- **Adaptierbarkeit:** Neue Versionen der Spezifikation der Verwaltungsschale werden häufig und regelmäßig veröffentlicht [3]. Das Testsystem muss ebenfalls mit dieser Geschwindigkeit entwickelt werden können.

- **Nutzerfreundlichkeit:** Nutzer des Testsystems benötigen einen aussagekräftigen und zugänglichen Testbericht. Dieser muss es ermöglichen, fehlgeschlagene Tests zu analysieren und Entwickler befähigen, gefundene Fehler effizient zu beseitigen.

- **Qualität:** Das Testsystem selbst sollte möglichst fehlerfrei sein, d.h. die enthaltenen Testfälle möglichst korrekt sein.

Es stellt sich die Frage, wie sich diese Anforderungen mit möglichst geringem Aufwand erfüllen lassen. Insbesondere das manuelle Definieren und Implementieren der Testfälle erfordert dabei einen großen Aufwand. In anderen Bereichen hat sich gezeigt, dass das *automatische* Erzeugen von Testfällen ein Mittel ist, um den Aufwand zu senken [22]. Kern dieser Methoden sind meist formale Beschreibungen der Zielspezifikation. Aus diesen formalen Spezifikationen leitet das Testsystem automatisiert Testfälle ab.

Im Rahmen dieser Arbeit soll daher ein Testsystem entwickelt werden, das die Konformität von Implementierungen der Verwaltungsschale überprüft. Dabei soll das Testsystem die Testfälle mithilfe von formalen Spezifikationen *automatisiert* erstellen. Dafür werden im Rahmen dieser Arbeit Methoden entwickelt, die das automatisierte Erstellen von Testfällen ermöglichen. Ferner soll im Rahmen dieser Arbeit untersucht werden, inwieweit formale Spezifikationen die Erfüllung der obigen Anforderungen ermöglichen. Die Anwendung der Methoden liefert somit eine Beurteilung des Reifegrades bestehender Software für die Verwaltungsschale.

1.1 Zielsetzung und Forschungsfragen

Im Rahmen dieser Arbeit sollen zwei wesentliche Aspekte untersucht werden. Zum einen soll ein Testsystem entwickelt werden, um das Softwareökosystem der Verwaltungsschale auf Konformität zu den Spezifikationen der Verwaltungsschale [3, 4, 8, 17] zu überprüfen. Zum anderen soll untersucht werden, inwieweit formale Spezifikationen dabei von Vorteil sind, indem Konformitätstests automatisiert abgeleitet werden. Daraus ergeben sich die folgenden fünf Forschungsfragen:

FF1: Wie ist das Softwareökosystem rund um die Verwaltungsschale aufgebaut? Welche Abhängigkeiten gibt es und welche grundlegenden Komponenten ergeben sich daraus, deren Konformität sicherzustellen ist?

Ziel dieser Forschungsfrage ist zu analysieren, *welche* Softwarekomponenten auf Konformität zu überprüfen sind.

FF2: Welche Besonderheiten zeichnen die Verwaltungsschale aus softwaretechnischer Sicht aus? Welche speziellen Anforderungen an ein System zum Überprüfen der Konformität der bzgl. Interoperabilität relevanten Teile des Verwaltungsschalen-Ökosystems ergeben sich daraus?

Anhand dieser Frage soll untersucht werden, welche Aspekte durch die zu entwickelnden Tests abgedeckt werden müssen. Implizit ergibt die Antwort auf diese Forschungsfrage, weshalb Konformitätstests für die Verwaltungsschale komplex sind und Standardmethoden und -tools nur eingeschränkt verwendet werden können.

FF3: Wie kann ein Testsystem für Softwarekomponenten der Verwaltungsschale umgesetzt werden?

Im Rahmen der Beantwortung dieser Frage werden zwei wesentliche Aspekte beleuchtet: Zum einen, wie Testaufbauten für die Konformitätsprüfung gestaltet werden, zum anderen, wie aus formalen Spezifikationen automatisiert Testfälle abgeleitet werden können.

FF4: Welche Fehler können mit dem Testsystem in realen Softwarekomponenten für die Verwaltungsschale aufgedeckt werden? Welche Grenzen gibt es?

Diese Forschungsfrage dient der Evaluation und Bewertung der vorgestellten Methoden. Indirekt liefert die Beantwortung dieser Frage auch eine Bewertung der Konformität der existierenden Softwarekomponenten im Ökosystem der Verwaltungsschale.

FF5: Welche Vor- und Nachteile hat die automatische Ableitung von Konformitätstests aus formalen Spezifikationen?

Diese Forschungsfrage bewertet die Eignung von formalen Spezifikationen für Konformitätstests. Wesentlich ist hierbei die Bewertung, inwieweit formale Spezifikationen bei der Erfüllung der obigen Anforderungen Adaptierbarkeit, Nutzerfreundlichkeit und Qualität dienlich sind.

Nach [1] werden vier Ebenen der Interoperabilität unterschieden, die in Abbildung 1.1 dargestellt sind. In dieser Arbeit wird vorrangig *syntaktische* Interoperabilität betrachtet. Semantische Interoperabilität wird am Rande in Form von Teilmodellen betrachtet. Das bedeutet, dass die vorgestellten Methoden davon ausgehen, dass die zu testenden Systeme bereits eine grundlegende Konnektivität hergestellt haben und Bits austauschen können. Explizit wird *nicht* organisatorische Interoperabilität betrachtet.

Abbildung 1.1: Ebenen der Interoperabilität (nach [1])

1.2 Eigene Veröffentlichungen und Software

1.2.1 Veröffentlichungen

A Flow Graph based Approach for Controlled Generation of AAS Digital Twin Instances for the Verification of Compliance Check Tools

IECON, 2022

Der Beitrag [23] beschreibt ein Konzept, um Verwaltungsschalen-SDKs auf Konformität zu überprüfen. Dafür wird das Metamodell mithilfe von JSON-Schema formalisiert, um daraus automatisiert Beispielinstanzen von Verwaltungsschalen zu erzeugen. Hierfür wird aus dem JSON-Schema ein Flussgraph abgeleitet. Darauf aufbauend wird ein Algorithmus vorgestellt, um diesen Flussgraphen systematisch abzudecken, um systematisch Beispielinstanzen zu erzeugen. Das Konzept und der vorgestellte Algorithmus bilden die Grundlage für diese Arbeit, insbesondere für das Testen von SDKs für Verwaltungsschalen.

Fences: Systematic Sample Generation for JSON-Schemas using Boolean Algebra and Flow Graphs

AST, 2024

Dieser Beitrag [13] greift die Konzepte aus [23] auf und entwickelt sie konsequent weiter. Insbesondere wird die Normalisierung von JSON-Schema vorgestellt, um die Constraints der Verwaltungsschale bei der Generierung von Beispielinstanzen berücksichtigen zu können. Ergebnis ist ein Verfahren, das nicht nur für das JSON-Schema der Verwaltungsschale, sondern für *beliebige* JSON-Schemata Beispielinstanzen erzeugt. Das Verfahren wird in dieser Arbeit beim Test von Verwaltungsschalen-SDKs und Servern eingesetzt.

Model-Based Test Case Generation for Compliance Checking of Reactive Asset Administration Shells

KOMMA, 2022

Der Beitrag [24] ist einer der ersten, der ein Verfahren zu Sicherstellung der Konformität von Verwaltungsschalenservern vorstellt. Grundlage bildet die OpenAPI-Beschreibung der Verwaltungsschale. Daraus werden automatisiert Testfälle abgeleitet. Eine wichtige Beobachtung ist die Tatsache, dass die Verknüpfungen zwischen den einzelnen Operationen

wichtige Testfälle liefern. Diese Arbeit basiert auf der vorgestellten Idee, die OpenAPI-Beschreibung der Verwaltungsschale zu verwenden. Allerdings wird zwischen negativen und positiven Tests unterschieden. Zur Erzeugung von negativen Tests wird das Verfahren aus [13] verwendet. Positive Testfälle werden manuell definiert.

Test-Framework zur Qualitätssicherung von Verwaltungsschalen-SDKs

Automation, 2023

Der Beitrag [5] verwendet die Erkenntnisse aus [23] und [12], um ein Test-Framework zur Qualitätssicherung von Verwaltungsschalen-SDKs zu entwickeln. Während [23] ein Verfahren zur Erzeugung von JSON Dateien liefert, werden in diesem Beitrag auch XML und AASX Dateien automatisiert erzeugt. Diese werden genutzt, um verschiedene Verwaltungsschalen-SDKs auf Konformität zu überprüfen. Abschließend werden verschieden Metriken diskutiert. Das vorgestellte Erzeugen von XML und AASX Dateien wird in dieser Arbeit aufgegriffen, ebenso die vorgestellten Metriken.

Automatisierte Generierung von Beispielinstanzen für Verwaltungsschalen Submodel Templates

Automation, 2024

Der Beitrag [15] beleuchtet die Rolle von Teilmodellen im Rahmen von Konformitätstests. Dafür wird ein Verfahren vorgestellt, um Teilmodelle mithilfe von JSON-Schema zu formalisieren. Diese Formalisierungen dienen im ersten Schritt dem Überprüfen von Instanzen auf Konformität zu den Teilmodellen. Im nächsten Schritt werden die Formalisierungen genutzt, um automatisch Testfälle zu generieren. Diese können beispielsweise zum Testen oder Dokumentieren der Teilmodelle genutzt werden. Das Verfahren wird exemplarisch anhand der Teilmodelle *Contact Information* [6] und *Digital Nameplate for Industrial Equipment* [7] evaluiert.

Test Engines for the Asset Administration Shell

at-Automatisierungstechnik, 2025

Der Journal-Beitrag [9] stellt die AAS Test Engines vor, ein Tool für die Überprüfung der Konformität von Verwaltungsschalenimplementierungen. Hierfür wird zunächst das Ökosystem der Verwaltungsschale analysiert. Basierend darauf werden zu testende Komponenten identifiziert. Anschließend werden Testaufbauten vorgestellt und anhand des AASX Servers evaluiert.

1.2.2 Software

Begleitend zu dieser Arbeit wurden drei Softwaretools entwickelt, die im Nachfolgenden beschrieben werden. Neben der Beschreibung wird auch eine Einschätzung des TRL nach [25] gegeben:

- **TRL 1**: Grundprinzipien beobachtet und beschrieben

- **TRL 2**: Technologiekonzept und/oder Anwendung formalisiert
- **TRL 3**: Analytische und experimentelle, kritische Funktion und/or Charakteristik
- **TRL 4**: Funktionale Verifikation in Laborumgebung
- **TRL 5**: Funktionale Verifikation in einer relevanten Umgebung
- **TRL 6**: Modell demonstriert die kritische Funktionalität in einer relevanten Umgebung
- **TRL 7**: Modell demonstriert die Funktionalität in der tatsächlichen Einsatzumgebung
- **TRL 8**: Tatsächliches System abgeschlossen und qualifiziert für den Einsatz
- **TRL 9**: Tatsächliches System erfolgreich im Einsatz

JSON-Schema Tool

`https://github.com/ifak/json_schema_tool`
TRL 7, MIT Lizenz

Das JSON-Schema Tool kann für ein gegebenes JSON-Schema überprüfen, ob JSON-Dokumente das Schema erfüllen. Das JSON-Schema Tool wird in Kapitel 4 verwendet, um die Konformität von Typ 1 Verwaltungsschalen zu überprüfen. Wesentliche Anforderungen bei der Entwicklung waren daher Geschwindigkeit und Flexibilität. Das Tool wird mit einem TRL 7 bewertet: Es ist funktional voll ausgereift und unterstützt den JSON-Schema Standard [18]. Die Funktionalität wurde mithilfe der offiziellen JSON-Schema Testsuite an realen Anwendungsfällen nachgewiesen.

Fences

`https://github.com/ifak/fences`
TRL 6, MIT Lizenz

Fences ist ein Tool, um für ein JSON-Schema systematisch Beispielinstanzen zu erzeugen. Fences implementiert das in [13] vorgestellte Verfahren. In dieser Arbeit wird es in Kapitel 5 verwendet, um Testfälle für den Test von Verwaltungsschalen-SDKs zu erzeugen. Außerdem wird es in Kapitel 6 verwendet, um invalide Parameterwerte für negative Testfälle zu erzeugen. Fences wird mit einem TRL 6 bewertet: Die angestrebte Funktionalität wurde implementiert, allerdings bisher nur an der Verwaltungsschale evaluiert und nicht in generischen Anwendungsfällen.

AAS Test Engines

`https://github.com/admin-shell-io/aas-test-engines`
TRL 6, Apache 2.0 Lizenz

Bei den AAS Test Engines handelt es sich um das von der IDTA genutzte Tool für Konformitätstests für die Verwaltungsschale. Dieses Tool wurde begleitend zu dieser Arbeit

entwickelt. Es nutzt Erkenntnisse und Verfahren aus dieser Arbeit. Daher wurde es in Kapitel 4, Kapitel 5 und Kapitel 6 für die Durchführung der Evaluierungen verwendet. Das Tool wird mit einem TRL von 6 bewertet: Die grundlegende Funktionalität wurde nachgewiesen. Außerdem ist ein Großteil der Funktionalität, d.h. des Metamodells und der Profile der Verwaltungsschale, implementiert. Ferner werden die AAS Test Engines bereits in der Praxis von Implementierungen der Verwaltungsschale zur Sicherstellung der Konformität verwendet.

1.3 Gliederung

Grundlage dieser Arbeit bilden formale Spezifikationen. Deren Konzepte werden in Kapitel 2 vorgestellt. Insbesondere werden die formalen Spezifikationen JSON-Schema und OpenAPI erläutert, die für diese Arbeit verwendet werden. Des weiteren wird in Kapitel 2 beschrieben, wie Konformitätstests durchgeführt werden. In diesem Zusammenhang wird die Rolle von formalen Spezifikationen verdeutlicht, die es ermöglichen, Konformitätstests weitgehend zu automatisieren.

In Kapitel 3 wird die Verwaltungsschale vorgestellt, die den Anwendungsfall für diese Arbeit darstellt. Dafür werden zwei der drei Erscheinungsformen der Verwaltungsschale erläutert. Implementierungen dieser Erscheinungsformen werden in dieser Arbeit auf Konformität überprüft. Den Abschluss von Kapitel 3 bildet eine systematische Betrachtung des Softwareökosystems rund um die Verwaltungsschale. Diese Betrachtung liefert damit die Antwort auf **Forschungsfrage 1**.

In Kapitel 4 wird ein Verfahren vorgestellt, mit dem passive Verwaltungsschalen auf Konformität überprüft werden. Dafür wird das Metamodell der Verwaltungsschale mithilfe von JSON-Schema formalisiert. Eine abschließende Evaluation zeigt Vor- und Nachteile des Verfahrens auf. Damit werden die **Forschungsfragen 2-5** im Kontext von passiven Verwaltungsschalen beantwortet.

Im Anschluss wird in Kapitel 5 ein Verfahren vorgestellt, mit dem Verwaltungsschalen-SDKs auf Konformität überprüft werden. Dieses nutzt die im vorherigen Kapitel entwickelte Formalisierung zum automatisierten Erzeugen von Testfällen. Abschließend werden Verwaltungsschalen aus dem Ökosystem auf Konformität geprüft. Dies beantwortet **Forschungsfragen 2-5** im Kontext von Verwaltungsschalen-SDKs.

In Kapitel 6 wird ein Verfahren vorgestellt, mit dem die Konformität von reaktiven Verwaltungsschalen überprüft wird. Dafür wird die API der Verwaltungsschale mithilfe von OpenAPI formalisiert. Daraus werden im Anschluss semi-automatisch Konformitätstests abgeleitet. Das vorgestellte Verfahren wird anschließend auf API-Implementierungen aus dem Ökosystem angewendet, um deren Konformität zu überprüfen. Dies beantwortet die **Forschungsfragen 2-5** im Kontext von reaktiven Verwaltungsschalen.

In Kapitel 7 wird die Verwendung von formalen Spezifikationen in Lebenszyklen von Software der Verwaltungsschale betrachtet. Dafür werden Phasen des Softwarelebenszyklus betrachtet, um die **Forschungsfrage 5** zu beantworten.

Abschließend gibt Kapitel 8 eine Zusammenfassung und einen Ausblick. Den Bezug der Forschungsfragen, Veröffentlichungen und Software zeigt Tabelle 1.1.

Tabelle 1.1: Forschungsfragen, Veröffentlichungen und Software

Kapitel		3	4	5	6	7
Forschungsfrage 1: Softwareökosystem VWS		x				
Forschungsfrage 2: Anforderungen			x	x	x	
Forschungsfrage 3: Umsetzung Testsystem			x	x	x	
Forschungsfrage 4: Bewertung Testsystem			x	x	x	
Forschungsfrage 5: Formale Spezifikationen			x	x	x	x
A Flow Graph based Approach for Controlled ...	[23]			x		
Fences: Systematic Sample Generation for ...	[13]			x		
Model-Based Test Case Generation for ...	[24]				x	
Test-Framework zur Qualitätssicherung von ...	[5]			x		
Automatisierte Generierung von Beispiel ...	[15]	x				
Test Engines for the Asset Administration Shell	[9]	x	x	x	x	x
JSON-Schema Tool			x		x	
Fences				x	x	
AAS Test Engines			x	x	x	x

2 Formale Spezifikationen und Konformität

Formale Spezifikationen [26, 27] beschreiben Systeme wie z.B. Softwaresysteme mithilfe einer geeigneten formalen Abstraktion, sodass sie maschinell lesbar und verarbeitbar sind. Damit stehen formale Spezifikationen im direkten Gegensatz zu menschenlesbaren Spezifikationen, die in der Regel in Form eines Fließtextes veröffentlicht werden [3]. Dadurch sind menschenlesbare Spezifikationen in der Regel schwer durch Maschinen interpretierbar.

Je nach zu beschreibendem System bieten sich unterschiedliche Formen der formalen Spezifikation an:

- **Zustandsbasierte Spezifikationen** beschreiben die gültigen Systemzustände. Dafür werden Invarianten festgelegt, die die gültigen Systemzustände eindeutig definieren. Ein Beispiel hierfür ist die formale Sprache Z [28], die Systeme mithilfe von Prädikatenlogik beschreibt.
- **Verhaltensorientierte Spezifikationen** beschreiben das gültige Verhalten eines Systems. Beispiele hierfür sind (endliche) Automaten und Petrinetze [29].
- **Datenorientierte Spezifikationen** definieren gültige Ein- und Ausgabedaten des Systems. Beispiele hierfür sind Schemasprachen wie XSD [30].

Die Auswahl der zu verwendenen Spezifikation richtet sich nach dem Einsatzzweck und weiteren Randbedingungen wie z.B. Vorkenntnisse der Beteiligten oder vorhandene Tools. In dieser Arbeit wird mit JSON-Schema eine datenorientierte und mit OpenAPI eine verhaltensorientierte Spezifikation verwendet.

2.1 JSON-Schema

JSON (JavaScript Object Notation) [31] ist ein allgemein einsetzbares Serialisierungsformat (ähnlich wie XML). Darauf aufbauend wurde mit JSON-Schema [32] eine Syntax definiert, um den Inhalt eines JSON-Dokumentes zu beschreiben. JSON-Schema wird eine wesentliche Rolle in dieser Arbeit einnehmen, insbesondere bei der Formalisierung des Metamodells der Verwaltungsschale. Daher gibt dieser Abschnitt eine Übersicht über JSON-Schema.

2.1.1 JSON und YAML

Ein JSON-Dokument ist hierarchisch aufgebaut und enthält in der Wurzel genau einen Wert. Dieser Wert kann einen von sechs verschiedenen Datentypen instanziieren. Im einfachsten Fall handelt es sich dabei um einen der vier Primitivtypen:

- Zahlen, wie z.B. `12` oder `-3.14`
- Zeichenketten, wie z.B. `""` oder `"text"`
- Bool-Werte `true` und `false`
- Der Null-Wert `null`

Darüber hinaus gibt es zwei komplexe Datentypen:

- Mit Arrays werden Listen von Werten dargestellt. Die Elemente der Liste können Werte aller sechs Datentypen sein, einschließlich des Arrays selbst. Beispiele für Arrays sind `[12, 22]`, `[]` oder `[true, [100, 200]]`.
- Mit Objekten werden Key-Value-Beziehungen abgebildet. Während die Keys ausschließlich Zeichenketten sein können, sind als Werte alle Datentypen erlaubt. Beispiele für Objekte sind `{}` oder `{"name":"Peter","age":44}`.

Durch die beiden komplexen Datentypen können JSON-Dokumente beliebig tief entfaltet sein. Solch tief entfaltete JSON-Dokumente sind für Menschen schwer lesbar, insbesondere weil JSON Einrückungen und Zeilenumbrüchen keine besondere Bedeutung zuweist. Um die Lesbarkeit zu verbessern, hat sich daher neben JSON auch YAML (YAML Ain't Markup Language) als Serialisierungsformat etabliert [33]. Technisch gesehen ist YAML sogar eine Obermenge von JSON, d.h. jedes JSON-Dokument ist auch ein gültiges YAML-Dokument. Im Gegensatz zu JSON ist YAML deutlich besser lesbar, insbesondere weil es Eindrückungen anstelle der Paare von geschweiften Klammern erlaubt. Einen Vergleich zwischen JSON und YAML zeigt Abbildung 2.1. Zur besseren Lesbarkeit werden alle JSON-Beispiele in dieser Arbeit als YAML formatiert.

```json
{ "name": "Peter",
  "age": 44,
  "children": [
    {  "name": "Tina",
       "age": 10
    },
    {  "name": "Dirk",
       "age": 15
    }
  ]
}
```

(a) JSON

```yaml
name: Peter
age: 44
children:
- name: Tina
  age: 10

- name: Dirk
  age: 15
```

(b) YAML

Abbildung 2.1: Identische Daten formatiert als JSON und YAML

2.1.2 JSON-Schema Validierung

Ein JSON-Dokument kann zunächst eine beliebige Struktur annehmen, die in der Regel vom Entwickler definiert wird. Um Interoperabilität mit anderen Anwendungen zu

ermöglichen, kann die Struktur über ein JSON-Schema formalisiert werden. Das Schema beschreibt dann z.B., welche Attribute vorhanden sein müssen und welche Typen diese haben.

Ein JSON-Schema ist zunächst selbst ein JSON-Dokument, in der Regel ein JSON Objekt. Um zu überprüfen, ob ein JSON-Dokument ein JSON-Schema erfüllt, werden alle Keys in dem JSON Objekt durchgegangen. Jeder Key definiert dabei eine Bedingung, die das JSON-Dokument erfüllen muss. Wenn das JSON-Dokument *alle* vorgegebenen Bedingungen erfüllt, so erfüllt es das ganze JSON-Schema. Das einfachste Schema ist somit das leere Schema `{}`. Das leere Schema wird von jedem JSON Wert erfüllt, da es keine Bedingungen definiert.

Typunabbhängige Bedingungen

Typunabbhängige Bedingungen können direkt auf den zu prüfenden JSON-Wert angewandt werden. So erlaubt die `const` Bedingung einen zu akzeptierenden Wert direkt anzugeben, wie z.B.

```
const: 3.14
```

Diese Schema wird von genau einem JSON-Dokument erfüllt, nämlich `3.14`. Alternativ kann mit `enum` auch eine Liste von zu akzeptierenden Werten angegeben werden:

```
enum:
- 3.14
- Hello World
```

Dieses Schema wird dann zusätzlich von dem JSON-Dokument `"Hello World"` erfüllt. Die wichtigste typunabhängige Bedingung ist die `type` Bedingung. Sie gibt an, welche Typen von Werten erlaubt sind. Z.B. akzeptiert folgendes Schema nur Zahlen und Zeichenketten:

```
type:
- number
- string
```

Die Vorgabe von erlaubten Typen findet sich in nahezu allen JSON Schemata, weil andernfalls implizit alle Typen erlaubt sind.

Bedingungen für Zahlenwerte

Bei Zahlenwerten kann ein Wertebereich definiert werden. Folgendes Schema akzeptiert nur Zahlenwerte $12 \leq x \leq 100$:

```
type: number
minium: 12
maximum: 100
```

Entsprechend akzeptiert folgendes Schema Zahlenwerte $12 < x < 100$:

```
type: number
exclusiveMinimum: 12
exclusiveMaximum: 100
```

Ferner kann mithilfe von `multipleOf` angegeben werden, dass nur Vielfache eines Wertes akzeptiert werden. Z.B. akzeptiert folgendes Schema nur gerade Zahlen:

```
type: number
multipleOf: 2
```

Bedingungen für Zeichenketten

Für Zeichenketten lassen sich Einschränkungen zur Länge angeben. Z.B. akzeptiert folgendes Schema nur Zeichenketten, die zwischen 10 und 20 Zeichen haben:

```
type: string
minLength: 10
maxLength: 20
```

Zusätzlich kann mithilfe von `pattern` spezifiziert werden, welchem regulären Ausdruck die Zeichenkette genügen muss. Folgendes Schema akzeptiert z.B. nur Zeichenketten mit Großbuchstaben:

```
type: string
pattern: "^[A-Z]*$"
```

Bedingungen für Arrays

Bei Arrays kann die Anzahl der erlaubten Elemente begrenzt werden. Z.B. akzeptiert folgendes Schema nur Arrays mit 10 bis 20 Elementen:

```
type: array
minItems: 10
maxItems: 20
```

Darüber hinaus können Bedingungen für die Elemente im Array definiert werden. Dafür kann mithilfe von `items` ein Subschema angegeben werden, das auf alle Elemente des Arrays angewandt wird. So akzeptiert folgendes Schema nur Arrays mit Zahlenwerten als Elementen:

```
type: array
items:
  type: number
```

Das Konzept des **Subschemas** ist sehr mächtig und wird an diversen Stellen bei JSON-Schema verwendet. So kann das Subschema beliebig komplex werden und sogar weitere Subschemata enthalten. Zur Illustration sei folgendes Schema gegeben:

```
type: array
items:
  type: array
  items:
    type: number
```

Es akzeptiert nur Arrays deren Elemente Arrays von Zahlenwerten sind, wie z.B:

```
[
  [0, 1, 2],
  [3, 4],
  []
]
```

Bedingungen für Objekte

Bei Objekten können Einschränkungen für die Anzahl der Properties gemacht werden. Folgendes Schema akzeptiert nur Objekte mit 10 bis 20 Properties:

```
type: object
minProperties: 10
maxProperties: 20
```

Darüber hinaus können bestimmte Properties als erforderlich markiert werden (alle nicht genannten Properties sind optional). Folgendes Schema akzeptiert nur Objekte, die mindestens die Properties `name` und `age` enthalten:

```
type: object
required:
- name
- age
```

Schließlich können die Werte der einzelnen Properties beschränkt werden. Hierfür wird für jede Property ein Subschema definiert. Folgendes Schema akzeptiert nur Objekte mit der Property `name`, die wiederum ein String sein muss:

```
type: object
required:
- name
properties:
  name:
    type: string
```

Subschemata können, wie bei Arrays, beliebig komplex und verschachtelt werden.

Logische Applikatoren

JSON-Schema bietet mit dem Konzept der Logischen Applikatoren die Möglichkeit, einen Satz von Subschemata mithilfe logischer Operationen zu verknüpfen. Beispielsweise akzeptiert folgendes Schema nur Zahlen zwischen 10 und 20 oder 70 und 80:

```
type: number
anyOf:
- minimum: 10
  maximum: 20
- minimum: 70
  maximum: 80
```

Der logische Applikator ist hierbei `anyOf`. Dieses Schlüsselwort erwartet eine Liste von Subschemata, deren Ergebnisse dann mit einem logischen ODER verknüpft werden. Neben `anyOf` bietet JSON-Schema insgesamt 5 logische Applikatoren, die in Tabelle 2.1 gelistet sind. Von besonderem Interesse ist hierbei das `not` Schlüsselwort [34]. Dieses invertiert das verwendete Subschema. So akzeptiert folgendes Schema alle Werte, die *kein* Objekt sind:

```
not:
  type: object
```

Dieses Schema akzeptiert somit Dokumente wie `12`, `[1,2,3]` oder `"Hello World"`.

Tabelle 2.1: Logische Applikatoren (nach [13])

Schlüsselwort	Beschreibung	Logische Operation
allOf	Alle Subschemata müssen akzeptieren	$\wedge$
anyOf	Mindestens ein Subschemata muss akzeptieren	$\vee$
oneOf	Genau ein Subschema muss akzeptieren	$\oplus$
not	Das Subschema darf nicht akzeptieren	$\neg$
if-then-else	Das Subschema wird abhängig von einem anderen Subschema überprüft	$\Longrightarrow$

Referenzen

Referenzen werden durch das Schlüsselwort `$ref` markiert. Der zugeordnete Wert wird als JSON Pointer [35] interpretiert, der auf ein anderes Schema im Gesamtschema verweist. Somit akzeptiert die `$ref` Bedingung einen JSON Wert genau dann, wenn das referenzierte Schema den JSON Wert akzeptiert. Referenzen werden insbesondere für zwei Anwendungsfälle verwendet:

Subschemata wiederverwenden: Oft werden Subschemata dupliziert verwendet, wie in folgendem Beispiel:

```
type: object
properties:
  first_name:
    type: string
    minLength: 10
  last_name:
    type: string
    minLength: 10
```

Das duplizierte Subschema sollte stattdessen extrahiert werden und durch eine Referenz ersetzt werden:

```
type: object
properties:
  first_name:
    $ref: '#/$defs/Name'
  last_name:
    $ref: '#/$defs/Name'
$defs:
  Name:
    minLength: 10
    type: string
```

Hier wird das spezielle `$defs` Schlüsselwort verwendet, unter dem man wiederzuverwendende Subschemata ablegen kann. Bei Änderungen am Subschema muss dieses somit nur noch an einer Stelle angepasst werden. Dies erhöht die Wartbarkeit und die Lesbarkeit des Schemas.

Rekursive Schemata: Viele reale Daten enthalten Rekursionen wie beispielsweise der Stammbaum des britischen Königshauses (aus [36]):

```yaml
name: Elizabeth
children:
- name: Charles
  children:
  - name: William
    children:
    - name: George
    - name: Charlotte
  - name: Harry
```

Dieser kann durch folgendes Schema beschrieben werden:

```yaml
type: object
properties:
  name:
    type: string
  children:
    type: array
    items:
      $ref: "#"
```

Jede Person enthält mit `children` ein Attribut, das wiederum eine Liste von Personen enthält. Diese Rekursion wird durch die Referenz `"#"` ausgedrückt. Sie verweist auf die Wurzel des Schemas, das dadurch rekursiv wird.

2.2 OpenAPI

REST APIs [37] sind eine weit verbreitete Möglichkeit, um strukturiert auf Webservices zuzugreifen. Um die Funktionalitäten einer solchen REST API zu beschreiben, wurde 2011 der Swagger Standard publiziert [38]. Dieser wurde 2016 als OpenAPI Standard [19] weiterentwickelt. Der nachfolgende Abschnitt gibt eine Übersicht über REST APIs sowie das OpenAPI Format.

2.2.1 HTTP und REST

Das Hypertext Transfer Protocol (HTTP) [39–41] wurde 1997 in der noch heute verwendeten Version 1.1 veröffentlicht. Es bildet die Grundlage der Datenübertragung im World Wide Web. Als zustandsloses Protokoll definiert HTTP einzelne, voneinander unabhängige Transaktionen zwischen einem Client und einem Server (siehe Abbildung 2.2).

Der Client initiiert eine Transaktion mit einem Request. Dieser besteht aus folgenden Teilen:

- Die angeforderte **URL** wie z.B. `/index.html`
- Die **Methode**, die bestimmt, welche Aktion ausgeführt werden soll. Z.B. signalisiert die Methode `GET`, dass Inhalte gelesen werden sollen oder die Methode `POST`, dass Inhalte geschrieben werden sollen.
- **Header**, die Metainformationen übermitteln, wie z.B. die Sprache des Clients oder das gewünschte Ausgabeformat.

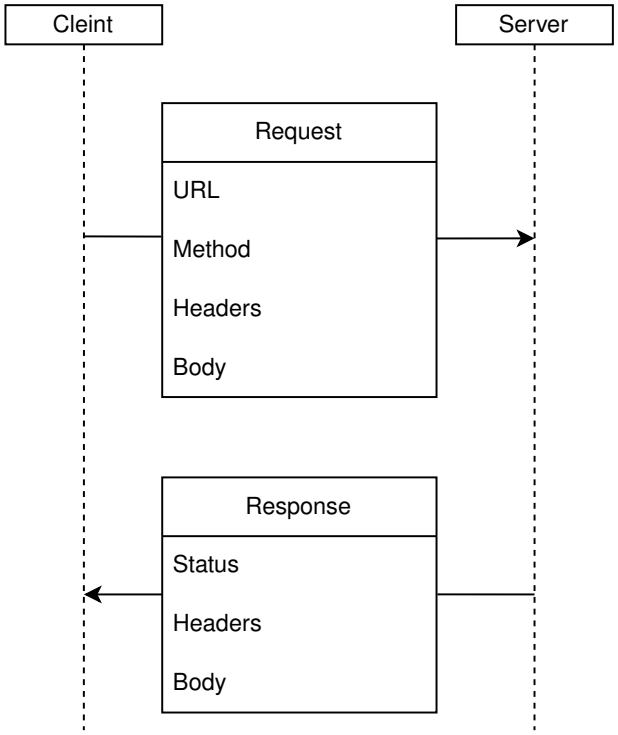


Abbildung 2.2: HTTP-Anfrage und Antwort

- Ein optionaler **Body**, in dem Daten übergeben werden. Z.B. können hier bei `POST` die zu schreibenden Daten übergeben werden.

Der Server antwortet daraufhin mit einem Response, womit die Transaktion beendet ist. Der Response besteht aus folgenden Teilen:

- Ein numerischer **Statuscode**, der angibt, ob der Request erfolgreich war. Statuscodes 200-299 zeigen an, dass die Aktion erfolgreich war. Die Werte 300-399 leiten den Client um, d.h. der Request soll bei einer anderen URL erneut probiert werden. Der Fehlerfall wird von Statuscodes 400-599 indiziert.
- Ähnlich wie beim Request geben **Header** zusätzliche Metainformationen zum Response.
- Im optionalen **Body** werden Daten übertragen, z.B. beim `GET`-Request die angefragten Daten.

Basierend auf HTTP definiert Representational State Transfer [37] (REST) die Struktur der URLs sowie die Semantik der einzelnen Methoden. Im Zentrum von REST stehen *Ressourcen*. Ein Ressource ist dabei eine Entität, deren Bedeutung sich aus der Anwendung ergibt. Beispielsweise bieten viele Webservices Zugriff auf eine Ressource `User` an, die die Benutzer im System repräsentiert. Zu jeder Ressource gibt es dann eine URL, um auf die Ressource zuzugreifen, z.B. `/users`. Je nachdem, welche Methode beim Zugriff auf diese URL angegeben wird, werden dann unterschiedliche Aktionen ausgelöst. Diese orientieren sich am klassischen CRUD (Create, Read, Update, Delete) Lebenszyklus:

- **POST** dient dem Erstellen der Ressource

- **GET** dient dem Auslesen der Ressource
- **PATCH** dient dem Update einer Ressource
- **DELETE** dient dem Löschen einer Ressource

Eine detaillierte Beschreibung dieser Methoden findet sich beispielsweise in [37].

2.2.2 Metamodell des OpenAPI-Formats

Das OpenAPI Format [19] dient der Beschreibung von HTTP Webdiensten. Daher finden sich viele Termini aus HTTP direkt im OpenAPI-Format wieder. Der nachfolgende Abschnitt gibt einen Einstieg in das OpenAPI-Format, in dem die wichtigsten Konzepte und Elemente erläutert werden. Als Anwendungsbeispiel soll dabei ein hypothetischer Webdienst dienen, der das Lesen und Schreiben von Nachrichten ermöglicht. Die Nachrichten sollen als JSON Objekte an die API übergeben werden und können wie folgt aussehen:

```
author: Petra
text: Hallo!
```

Zum Verarbeiten von Nachrichten bietet der Webdienst zwei Operationen an, die unter URL `/messages` erreichbar sind:

- Mit der Methode POST kann eine Nachricht geschrieben werden. Im Request wird eine Nachricht übergeben.
- Mit der Methode GET werden alle Nachrichten abgerufen. In der Antwort findet sich eine Liste von Nachrichten.

Im Folgenden wird dieser Webdienst mithilfe von OpenAPI formal spezifiziert. Eine Übersicht über die Elemente des OpenAPI Metamodells zeigt Abbildung 2.3.

Den Einstiegspunkt bildet das `OpenAPI` Objekt. Dieses hat als erforderliches Attribut `openapi`, das die Version des OpenAPI Metamodells enthält. Ferner gibt es das `info` Attribut, das Metainformationen über den Webdienst enthält. Eine erste minimale Beschreibung des Nachrichten-Webdienstes sieht somit folgendermaßen aus (OpenAPI Beschreibungen werden in der Regel als YAML Datei serialisiert):

```
openapi: 3.1
info:
  title: Nachrichten-Webdienst
  version: 1.0
```

Zu beachten ist, dass `3.1` die Version des OpenAPI Metamodells bezeichnet, `1.0` die Version des Webdienstes. Als nächstes wird der Endpunkt `/messages` formalisiert. Hierfür dient das `paths` Attribut, das ein Mapping von URL auf Endpunktbeschreibung enthält:

```
openapi: 3.1
info: ...
paths:
  /messages:
    description: Endpunkt fuer Nachrichten
```

An dieser Stelle kann das `description` Attribut genutzt werden, um eine menschenlesbare Beschreibung einzufügen.

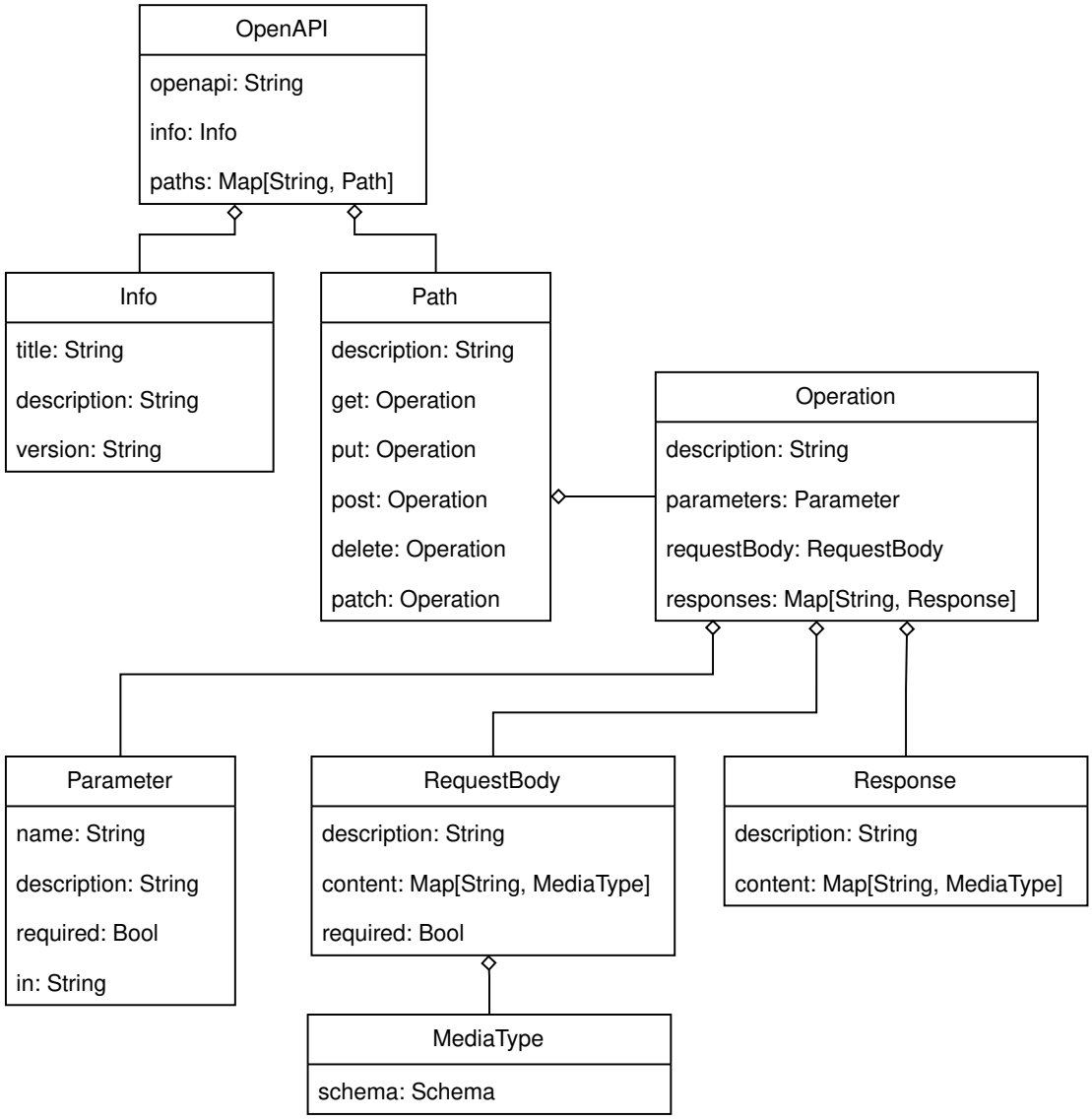


Abbildung 2.3: Metamodell von OpenAPI (vereinfacht)

Die `Path`-Objekte enthalten Attribute für die verschiedenen HTTP-Methoden. Im Falle des Beispiel-Webdienstes sind dies `get` und `post`:

```
openapi: 3.1
info: ...
paths:
  /messages:
    description: Endpunkt fuer Nachrichten
    post:
    get:
```

An dieser Stelle ist es sinnvoll, das Metamodell der Nachrichten zu beschreiben. Dies geschieht mithilfe von JSON-Schema (siehe Abschnitt 2.1):

```
openapi: 3.1
info: ...
paths:
  /messages:
    description: Endpunkt fuer Nachrichten
    post:
    get:
components:
  schemas:
    Message:
     type: object:
     properties:
       author:
         type: string
       text:
         type: string
```

Hiermit wird festgelegt, dass Nachrichten Objekte sind, die die Attribute `author` und `text` haben. Das Schema ist unter `components` abgelegt, das speziell für wiederverwendbare Schemata gedacht ist. Dies erhöht zum einen die Lesbarkeit, zum anderen aber auch die Wartbarkeit der OpenAPI-Beschreibung. Für die POST Methode muss nun beschrieben werden, dass die zu schreibende Nachricht im Body mit zu übergeben ist:

```
openapi: 3.1
info: ...
paths:
  /messages:
    description: Endpunkt fuer Nachrichten
      post:
        requestBody:
          content:
           'application/json':
             schema:
               $ref: '#/components/schemas/Message'
      get:
components: ...
```

In diesem Fall sind die Daten mit JSON serialisiert. Mit OpenAPI können aber auch andere Serialisierungsformate wie XML angegeben werden. Abschließend wird analog dazu

im Falle der GET Methode eine Liste aller Nachrichten im Body zurückgegeben:

```
openapi: 3.1
info: ...
paths:
  /messages:
    description: Endpunkt fuer Nachrichten
      post: ...
      get:
        responses:
          '200':
            content:
              'application/json':
                schema:
                  type: array
                  items:
                    $ref: '#/components/schemas/Message'
components: ...
```

Die Zahl '200' bezieht sich hierbei auf den Statuscode, den der Webdienst zurückgibt, hier also den Gutfall. Für reale Webdienste werden in der Regel noch weitere Statuscodes für verschiedene Fehlerfälle beschrieben.

Setzt man alle diese Bestandteile zusammen, ergibt sich somit die finale OpenAPI-Beschreibung, wie in Listing 2.1 dargestellt.

Listing 2.1: OpenAPI-Beschreibung eines Webdienstes für Nachrichten

```
openapi: 3.1
info:
  title: Nachrichten-Webdienst
  version: 1.0
paths:
  /messages:
    description: Endpunkt fuer Nachrichten
      post:
        requestBody:
          content:
           'application/json':
             schema:
               $ref: '#/components/schemas/Message'
      get:
        responses:
          '200':
            content:
              'application/json':
                schema:
                  type: array
                  items:
                    $ref: '#/components/schemas/Message'
components:
  schemas:
    Message:
```

```
type: object:
properties:
  author:
    type: string
 text:
   type: string
```

Durch diese Beschreibung sind alle Eigenschaften des Webdienstes erfasst.

2.3 Sicherstellung von Konformität mithilfe von formalen Spezifikationen

Zur Sicherstellung von Konformität hat sich im Rahmen von Softwaresystemen das Testen etabliert [21, 22]. Das Kernartefakt beim Testen sind die Testfälle. Ein Testfall ist ein Paar aus Eingabe und der Ausgabe, die von dem Softwaresystem auf die Eingabe erwartet wird. Die Art der Eingabe und Ausgabe hängt dabei sehr stark vom Testgegenstand (die zu testende Komponente) und der Teststufe (Unittest, Abnahmetest etc.) ab. Hierzu einige Beispiele:

- Auf der Ebene von Unittests werden in der Regel einzelne Softwarefunktionen getestet. Eingabe und Ausgabe sind hierbei Objekte und Datentypen der gewählten Programmiersprache.
- Im Rahmen von Netzwerktests werden Netzwerkpakete als Ein- und Ausgabe der Testfälle definiert.
- Für den Test von Nutzeroberflächen wird eine Reihe von Maus- und Tastatureingaben vorgegeben, auf die Nutzeroberfläche mit einem vorgegebenen Aussehen antworten soll.
- Für den Test von Dateien wird überprüft, ob die Dateien konform zu einem vorgegebenen Format oder Metamodell sind.

Man unterscheidet zwischen Positiv- und Negativtests. Bei Positivtests wird eine valide Eingabe an die Software übergeben, die sich entsprechend valide verhalten muss. Bei Negativtests hingegen wird eine invalide Eingabe an die Software übergeben. Invalide bedeutet hierbei z.B., dass sich die Eingabe außerhalb des gültigen Wertebereiches befindet.

Unter der Teststufe versteht man die Granularität, in der getestet wird. Bei Unittests werden einzelne Softwarefunktionen getestet, bei Integrationstests das Zusammenspielen einzelner Komponenten und bei Systemtests das komplette Softwaresystem. Die Ausführung der Tests in niedrigeren Teststufen dauert in der Regel kürzer. Daher werden hier oft sehr viele, kleinschrittige Testfälle definiert. Mit steigender Teststufe dauert auch die Ausführung länger, d.h. hier werden normalerweise deutlich weniger, aber sehr umfangreiche Tests wie Systemtests definiert. Die verschiedenen Teststufen werden im Rahmen von Entwicklungsparadigmen unterschiedlich abgebildet. Beispielsweise finden sich im neuen V-Modell [2] die Teststufen linear abgebildet zu den einzelnen Stufen des Entwicklungsprozesses (Abbildung 2.4).

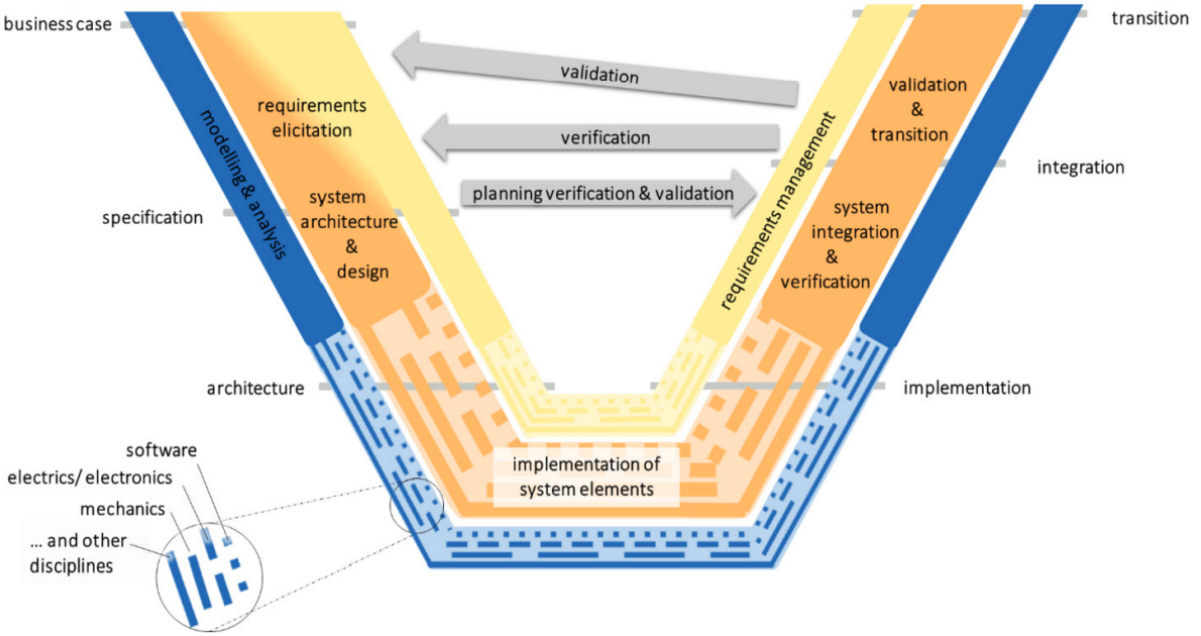


Abbildung 2.4: Das neue V-Modell (aus [2])

Neben der Teststufe wird zwischen Black- und Whiteboxtest unterschieden. Beim Blackboxtest wird die Software ohne Vorwissen über selbige getestet. Beim Whiteboxtest hingegen fließt Wissen über die Software in das Design der Testfälle mit ein. So sind Unittests per Definition Whiteboxtests, während Systemtests in der Regel als Blackboxtests durchgeführt werden.

In dieser Arbeit werden **Konformitätstests** betrachtet. Diese haben die Besonderheit, dass die Testfälle unabhängig von dem später zu bewertenden Softwaresystem entwickelt werden müssen. Folglich handelt es sich hierbei um Blackboxtests. Um Vergleichbarkeit zu gewährleisten, müssen alle Konformitätstests mit demselben Satz an Testfällen ausgeführt werden. Diese Testfälle werden hierzu in einer Testdatenbank zusammengefasst.

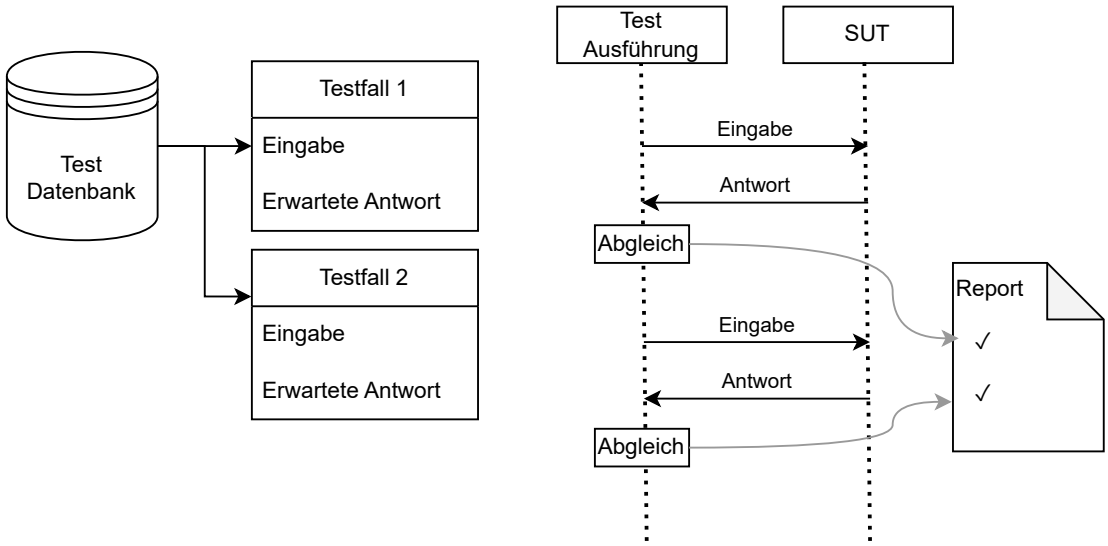


Abbildung 2.5: Durchführung eines Konformitätstests

Die Testdurchführung ist in Abbildung 2.5 dargestellt. Zunächst wird der erste Testfall aus der Testdatenbank entnommen und die Eingabe an das zu testende System übergeben. Dieses liefert eine Ausgabe, die dann mit der erwarteten Ausgabe verglichen wird. Stimmen die erwartete Ausgabe und die tatsächliche Ausgabe überein, hat das System den Testfall bestanden. Dieses Ergebnis wird im Testreport vermerkt. Auf diese Weise werden alle Testfälle der Testdatenbank abgearbeitet. Hat das System alle Testfälle bestanden, so wird es als konform angesehen.

2.3.1 Erzeugung der Testdatenbank

Die Testdatenbank ist das Herzstück des Konformitätstests. Das Definieren und Entwickeln der Testfälle ist somit ein sehr wichtiger Prozess. Die Testfälle müssen diversen Ansprüchen genügen:

- **Korrektheit** Die Testfälle müssen das spezifizierte Verhalten korrekt abbilden.
- **Vollständigkeit** Die Testfälle müssen alle Aspekte der Spezifikation abdecken.
- **Effizienz** Es sollte nicht zu viele Testfälle geben, da sich sonst die Aufwand bei Erstellung und Testausführung unnötigerweise erhöht.
- **Aussagekraft** Wenn Testfälle fehlschlagen, sollte leicht nachvollziehbar sein, warum sie fehlgeschlagen sind und welcher Defekt in der Software vorliegt.

Bei kleineren Softwaresystemen kann die Testdatenbank manuell erstellt werden. Durch den kleineren Umfang können die obigen Kriterien sichergestellt werden. Bei größeren Softwaresystem wie der Verwaltungsschale ist dies allerdings nicht mehr so leicht möglich. Insbesondere müssen die Testfälle bei jeder neuen Version der Spezifikation neu erzeugt werden. Außerdem ist das manuelle Erstellen sehr fehleranfällig und schwer parametrierbar z.B. in Bezug auf die gewünschte Abdeckung der Spezifikation.

An dieser Stelle zeigt sich ein weiterer großer Vorteil von formalen Spezifikationen: Sie ermöglichen das automatische Generieren der Testdatenbank. Dies ist in Abbildung 2.6 dargestellt.

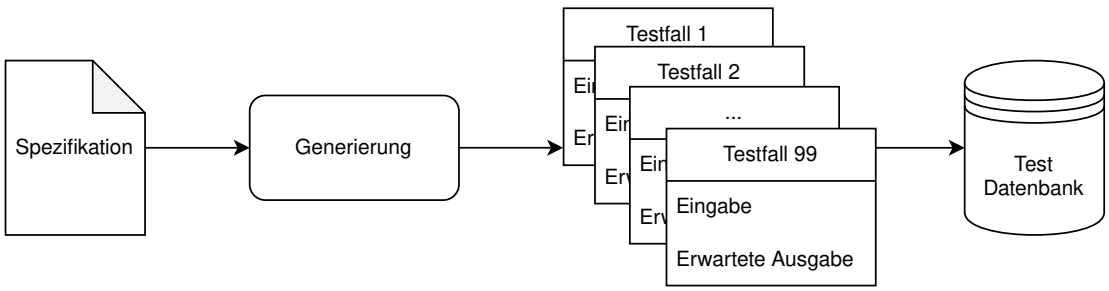


Abbildung 2.6: Automatische Generierung der Testdatenbank

Die automatische Erzeugung von Testdaten aus diversen Quellen wie formalen Spezifikationen ist seit vielen Jahren Gegenstand der Forschung [22]. Im Nachfolgenden werden einige grundlegende Methoden aufgeführt. Im Allgemeinen gilt bei der Wahl des Verfahrens: Je mehr manueller Aufwand bei der initialen Konfiguration des Verfahrens nötig ist, desto besser ist die Qualität (bzgl. der oben genannten Anforderungen) der automatisch

erzeugten Testdaten. Das liegt daran, dass bei der initialen Konfiguration Wissen über das zu testende System abgelegt wird. Je mehr von diesem Wissen dem Testsystem zur Verfügung steht, desto besser sind die erzeugten Testfälle. Abbildung 2.7 zeigt einen schematischen Vergleich der verschiedenen Verfahren. Moderne Tools kombinieren oft mehrere dieser Methoden, um bessere Testergebnisse zu erzielen.

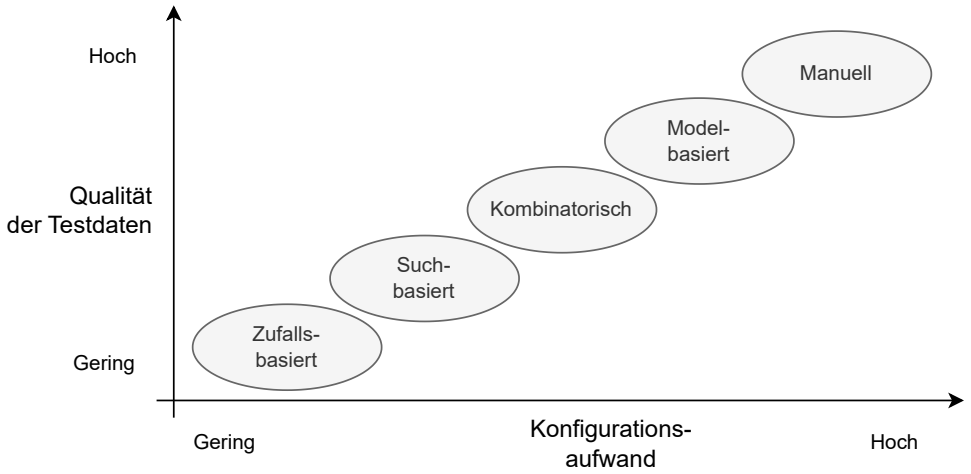


Abbildung 2.7: Vergleich von Verfahren zur Erzeugung von Testfällen

Zufallsbasierte Verfahren

Zufallsbasierte Verfahren erzeugen Testfälle zufällig [42]. Ein großer Vorteil dieser Verfahren ist, dass kaum Konfiguration der Generierung und wenig Wissen über das SUT nötig sind. Da komplett zufällige Testfällen nur eine geringe Qualität aufweisen, wurden diverse Methoden entwickelt, um die Qualität der Testfälle zu erhöhen. Frühe Arbeiten wie [43] verbessern zufällig erzeugte Testfälle, indem diese einer symbolischen Ausführung des SUTs unterzogen werden. Bei der symbolischen Ausführung wird ein Programm mit Symbolen anstelle von konkreten Werten ausgeführt, um Abhängigkeiten zwischen Ein- und Ausgabewerten abzuleiten. Dabei auftretende Nebenbedingungen werden durch neue Zufallsdaten abgedeckt. Moderne Methoden wie Fuzzing [44] beobachten das SUT während der Ausführung der zufälligen Testfälle, um Rückschlüsse auf sinnvolle neue Testfälle zu erhalten.

Suchbasierte Verfahren

Suchbasierte Verfahren [45, 46] erzeugen Testfälle iterativ. Bei jeder Iteration werden bestehenden Testfälle mutiert oder neue erzeugt, wobei eine gegebene Zielfunktion maximiert werden soll. Ein Tester muss hierbei lediglich die Zielfunktion und einen Suchalgorithmus vorgeben, um automatisiert Testfälle zu erzeugen. Allerdings sind die Ergebnisse sehr abhängig davon, dass eine sinnvolle Zielfunktion gewählt wird. Außerdem kann die Erzeugung von guten Testfällen eine lange Zeit dauern.

Kombinatorisches Testen

Beim kombinatorischen Testen [47, 48] werden vorgegebene Testfälle oder abstrakte Klassen von Testfällen in der Art kombiniert, dass sie möglichst viele Fehlerfälle abdecken. Die Autoren von [49] haben kombinatorisches Testen in industriellen Anwendungsfällen erfolgreich angewendet.

Modellbasierte Verfahren

Beim modellbasierten Testen [50, 51] werden Testfälle anhand eines Systemmodells abgeleitet. Modellbasierte Verfahren erfordern einen hohen manuellen Aufwand, weil zunächst das Systemmodell angelegt werden muss. Dafür können sehr viele und qualitativ hochwertige Testfälle automatisch aus dem Modell erzeugt werden. Eine Unterform des modellbasierten Testen ist Property-based-Testing [52], bei dem Testfälle anhand eines Datenmodells abgeleitet werden. Ein verbreitetes Tool ist Hypothesis [53], das Property-based-Testing für die Programmiersprache Python implementiert. Hypothesis wird im Rahmen dieser Arbeit als Vergleichstool zur Evaluation verwendet.

2.3.2 Testreport und Metriken

Die Ausführung eines Konformitätstests liefert, wie in Abbildung 2.5 dargestellt, einen Report. Der Report enthält eine Auflistung aller Testfälle und ob die Testfälle von dem SUT bestanden wurden oder nicht. Neben dieser binären Einteilung führen moderne Testsysteme oft auch weitere Zwischenstufen ein, wie z.B. Warnungen oder fatale Fehler [9]. Diese Art von Report ist für Entwickler hilfreich, um Defekte in der Software zu beheben. Dafür werden die fehlgeschlagenen Testfälle nacheinander abgearbeitet und behoben.

Während detaillierte Reports für Entwickler hilfreich sind, können für Entscheider Metriken aus dem Report generiert werden. Dadurch können z.B. Softwarekomponenten zu verschiedenen Entwicklungszeitpunkten oder von verschiedenen Herstellern leicht verglichen werden. Dafür teilt man die Testfälle in negative und positive Testfälle bzw. akzeptierte und abgewiesene Testfälle ein. Anschließend ordnet man die Anzahlen entsprechend der vier möglichen Kombinationen in einer Konfusionsmatrix [54]:

	Positive Testfälle	Negative Testfälle
Akzeptiert	True Positives TP	False Positives FP
Abgewiesen	False Negatives FN	True Negatives TN

Eine häufig eingesetzte Metrik ist die Genauigkeit A_U. Sie beschreibt das Verhältnis der bestandenen Testfälle zur Anzahl aller Testfälle:

$$A_u = \frac{TP + TN}{TP + FP + TN + FN}$$

In der Regel definiert ein Testsystem deutlich mehr negative als positive Testfälle. Die (unbalancierte) Genauigkeit hat daher den Nachteil, dass die negativen Testfälle deutlich stärker gewichtet werden aufgrund ihrer hohen Anzahl. Dieses Problem löst die balancierte Genauigkeit A_b, indem der Mittelwert beider Raten gebildet wird:

$$A_b = \frac{\frac{TP}{TP+FN} + \frac{TN}{TN+FP}}{2}$$

Die vorgestellten Metriken können als Teil des Testreports ausgegeben werden. Sie werden im Rahmen dieser Arbeit zur Evaluation verschiedener Verfahren und Komponenten verwendet.

3 Verwaltungsschale

Im Rahmen der vierten Industrielle Revolution (Industrie 4.0) soll eine vollständige Digitalisierung von Wertschöpfungsketten in der Industrie erfolgen [16]. Grundlage hierfür bilden Digitale Zwillinge, also digitale Repräsentationen von Gütern und Prozessen in der realen Welt. Die Verwaltungsschale (*engl.* Asset Administration Shell, AAS) ist ein Standard für Digitale Zwillinge [55]. Die Spezifikation und Entwicklung der Verwaltungsschale wird von der Industrial Digital Twin Association [56] (IDTA) vorangetrieben.

Bei der Verwaltungsschale werden drei Erscheinungsformen unterschieden. Abbildung 3.1 gibt eine Übersicht. Bei der ersten Erscheinungsform handelt es sich um einen Austausch von Verwaltungsschalen in Form von Dateien. Dies wird auch als **passive Verwaltungsschale** bezeichnet. Bei der zweiten Erscheinungsform, der **reaktiven Verwaltungsschale**, handelt es sich um eine laufende Instanz einer Verwaltungsschale, auf die mithilfe eines Application Programming Interface (API) zugegriffen werden kann. Die dritte Erscheinungsform wird als **proaktive Verwaltungsschale** bezeichnet. Hierbei agieren verschiedene Verwaltungsschalen autonom miteinander mithilfe der I4.0 Sprache [57]. Im Rahmen dieser Arbeit werden nur die ersten beiden Erscheinungsformen betrachtet, da es für proaktive Verwaltungsschalen bisher keine Spezifikation gibt.

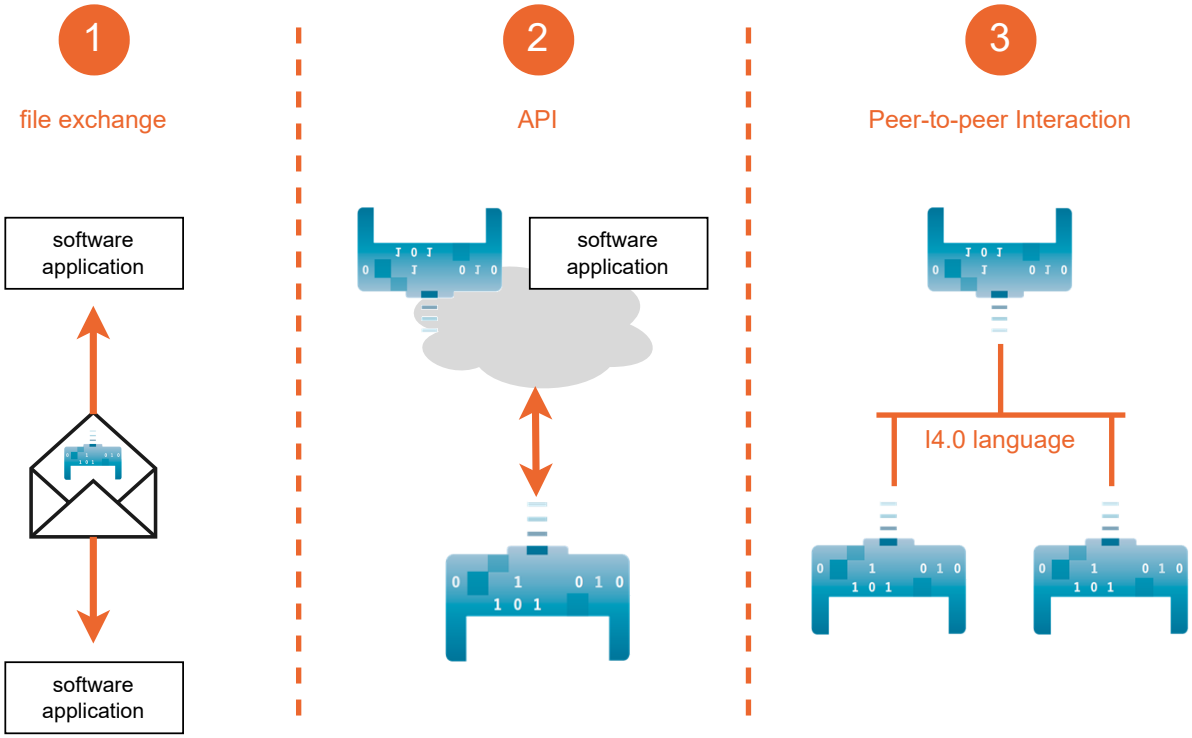


Abbildung 3.1: Erscheinungsformen der Verwaltungsschale (aus [3])

Die Spezifikation der Verwaltungsschale ist in folgende Teile aufgeteilt:

- **Part 1: Metamodell** [3] beschreibt das grundlegende Metamodell der Verwaltungsschale. Das Metamodell der Verwaltungsschale bildet die Basis für alle anderen Parts.
- **Part 2: Application Programming Interfaces** [8] beschreibt Schnittstellen, um auf laufende Instanzen von Verwaltungsschalen zuzugreifen. Dabei werden die Schnittstellen zunächst technologieneutral beschrieben und im Anschluss auf HTTP REST abgebildet.
- **Part 3a: Data Specification – IEC 61360** [17] beschreibt, wie basierend auf IEC 61360, Semantiken von Daten in der Verwaltungsschale beschrieben werden können.
- **Part 5: Package File Format (AASX)** [4] spezifiziert das AASX Dateiformat, das als Containerformat eine Verwaltungsschale samt allen ergänzenden Dateien bündelt.

Part 1 ist die Grundlage für die passive Verwaltungsschale, Part 2 die Grundlage für die reaktive Verwaltungsschale. Part 3a und 5 sind Ergänzungen zu Part 1, die zur besseren Verständlichkeit in eigene Parts ausgelagert wurden. Für die proaktive Verwaltungsschale wurde bisher keine Spezifikation veröffentlicht. Im folgenden wird ein Überblick über die Inhalte der Spezifikation der Verwaltungsschale gegeben. Außerdem beantwortet dieses Kapitel die 1. Forschungsfrage.

3.1 Metamodell

Grundlage aller Erscheinungsformen ist das Metamodell der Verwaltungsschale, das in Part 1 der Spezifikation [3] beschrieben wird. Der Einstiegspunkt im Metamodell ist die `Environment`, die in Abbildung 3.2 dargestellt ist. Die `Environment` listet alle Verwaltungsschalen, alle Submodelle sowie alle Konzeptbeschreibungen auf.

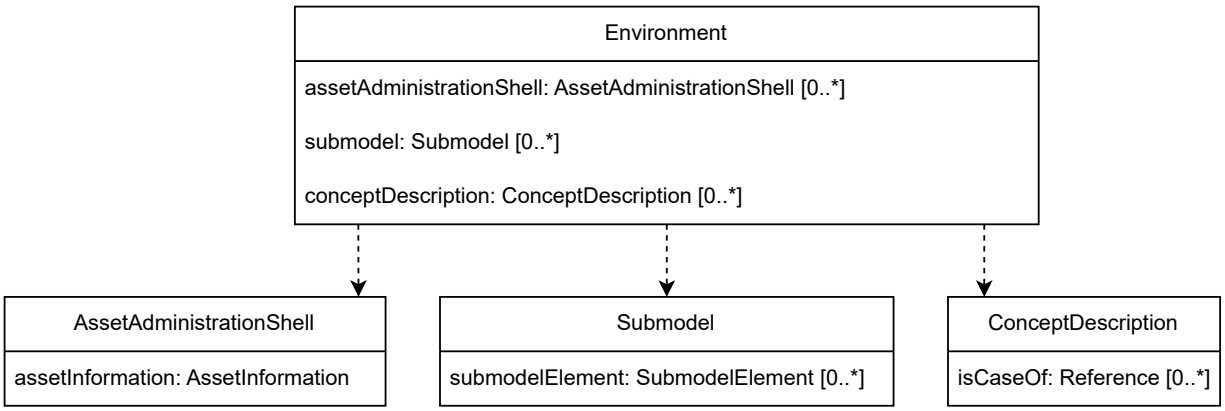


Abbildung 3.2: Metamodell von `Environment` (nach [3])

3.1.1 Metamodell von AssetAdministrationShell

Das Metamodell einer Verwaltungsschale (`AssetAdministrationShell`) zeigt Abbildung 3.3. Das Modell `AssetAdministrationShell` hat drei Attribute: `derivedFrom` ist

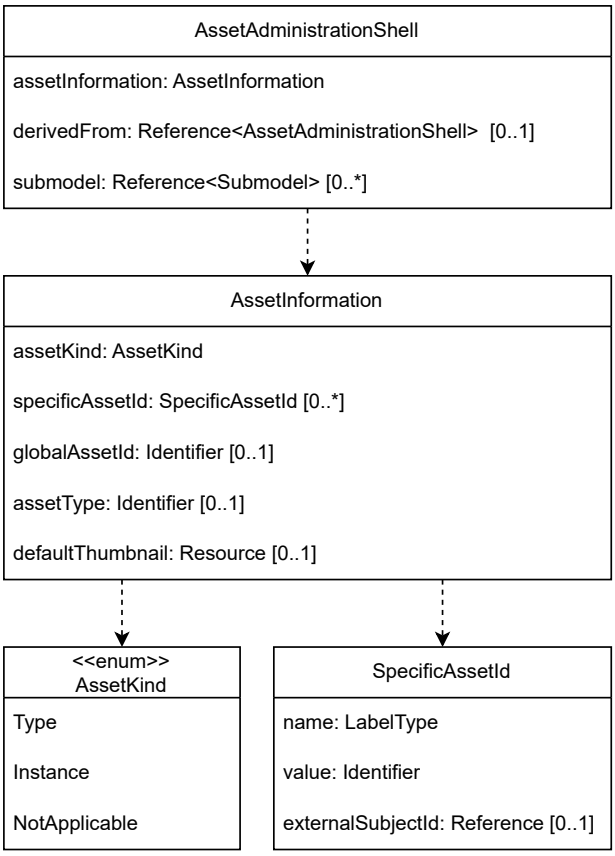


Abbildung 3.3: Metamodell von `AssetAdministrationShell` (nach [3])

eine Referenz auf eine andere Verwaltungsschale, von der diese Verwaltungsschale seine Eigenschaften geerbt hat. Im Attribut `submodel` finden sich Referenzen auf alle Submodelle, die in der Verwaltungsschale enthalten sind. Dies können externe Referenzen sein oder Verweise auf Submodelle aus der `Environment`. Im Attribut `assetInformation` werden Metainformationen über die Verwaltungsschale abgelegt. Hierbei legt `assetKind` fest, ob es sich um einen Typ oder eine Instanz handelt. Darüber hinaus enthält die `globalAssetId` eine globale ID zur Identifikation der Verwaltungsschale. Diese kann zum Beispiel im `derivedFrom` Attribut genutzt werden, um festzuhalten, wenn eine Instanz von einem Typ abgeleitet wurde. Neben der globalen ID können noch interne IDs im Attribut `specificAssetId` hinterlegt werden. Beispiele für spezifische Asset IDs sind Seriennummern und herstellerspezifische IDs.

3.1.2 Metamodell von Submodel

Submodelle enthalten die eigentlichen Nutzdaten der Verwaltungsschale. Deren Art und Struktur hängt von der Einsatzdomäne ab. Abbildung 3.4 zeigt das Metamodell eines `Submodels`.

Ein Submodell enthält Submodellelemente im Attribut `submodelElement`. Hierbei handelt es sich um eine abstrakte Klasse, die diverse konkrete Kindklassen aufweist.

Die eigentlichen Daten sind in Submodellelementen der Klasse `DataElement` enthalten, die diverse Kindklassen hat (siehe Abbildung 3.5). In Instanzen vom Typ `Property` können einzelne Werte abgelegt werden, wie z.B. Zeichenketten oder Zahlen. Für Werte, die sprach-

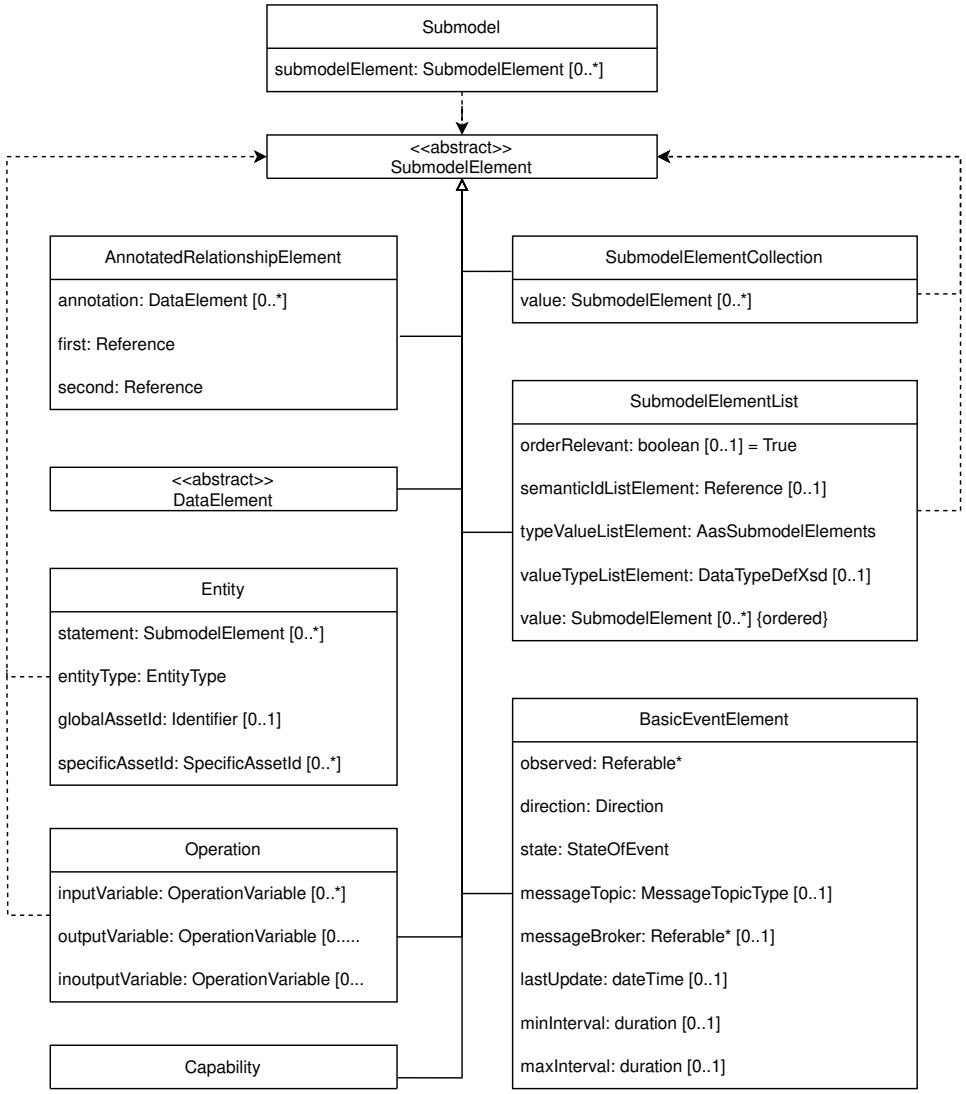


Abbildung 3.4: Metamodell von `Submodel` (nach [3])

bzw. ortsabhängig sind, wird der Typ `MultiLanguageProperty` genutzt. In einem `Range` können Wertebereiche abgelegt werden. Um Dateien in einer Verwaltungsschale abzulegen, gibt es die Klassen `File` bzw. `Blob`.

Von besonderem Interesse für diese Arbeit sind die Klassen `SubmodelElement-Collection` sowie `SubmodelElementList`. Diese enthalten eine Liste von Submodellelementen, also ggf. auch Instanzen von sich selbst. Dadurch wird das Metamodell der Verwaltungsschale rekursiv.

3.1.3 Metamodell von ConceptDescription

Konzeptbeschreibungen beschreiben die Semantik von Submodellelementen. Das Metamodell von `ConceptDescription` zeigt Abbildung 3.6. Das Attribut `isCaseOf` ist eine Referenz auf eine externe semantische Beschreibung, der diese Konzeptbeschreibung folgt.

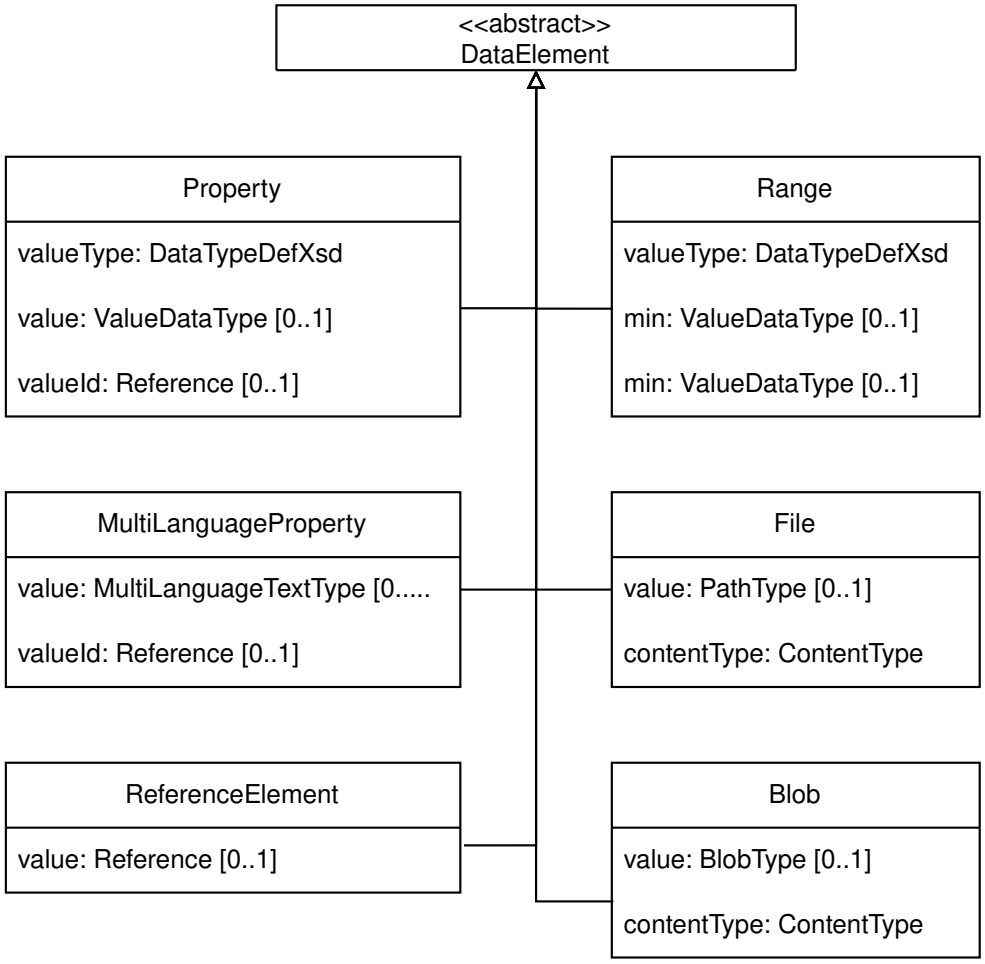


Abbildung 3.5: Metamodell von `DataElement` (nach [3])

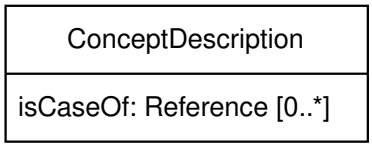


Abbildung 3.6: Metamodell von `ConceptDescription` (nach [3])

3.2 Constraints

Die Spezifikation der Verwaltungsschale enthält neben dem Metamodell auch *Constraints*. Diese beschreiben zusätzliche Bedingungen, die die Attribute einer Verwaltungsschale erfüllen müssen. Eine Auflistung aller Constraints findet sich in Tabelle 3.1. Diese werden in der Spezifikation der Verwaltungsschale nicht weiter kategorisiert. Im Rahmen dieser Arbeit wurden die Constraints analysiert, um sie in vier Kategorien einzuteilen. Dies vereinfacht die spätere Formalisierung der Constraints.

Wertebezogene Constraints beschreiben Anforderungen an den Wert *eines* Attributes, also unabhängig von anderen Attributen. Als Beispiel sei hier Constraint AASd-002 [3, Seite 57] genannt:

> **Constraint AASd-002**: idShort of Referables shall only feature letters, digits, underscore ("_"); starting mandatory with a letter, i.e. [a-zA-Z][a-zA-Z0-9_]*.

AASd-002 bezieht sich lediglich auf das `idShort` Attribut und auf keine weiteren Attribute. Somit kann dieser Constraint geprüft werden ohne Kenntnis der anderen Attribute.

Bei **bedingten Constraints** müssen manche Attribute existieren oder bestimmte Werte belegen in Abhängigkeit von der Existenz oder Belegung anderer Attribute. Ein Beispiel hierfür ist AASd-005 [3, Seite 47]:

> **Constraint AASd-005:** If AdministrativeInformation/version is not specified, AdministrativeInformation/revision shall also be unspecified. This means that a revision requires a version. If there is no version, there is no revision. Revision is optional.

Zur Überprüfung dieses Constraints müssen mehrere Attribute in Beziehung gesetzt werden, nämlich die Attribute `version` und `revision`. Zu beachten ist, dass bei bedingten Constraints nur die Existenz der Attribute oder die Belegung mit bestimmten Konstanten zu überprüfen ist.

Nicht prüfbare Constraints lassen sich nicht deterministisch oder algorithmisch eindeutig überprüfen. Ein Beispiel hierfür ist Constraint AASd-012 [3, Seite 72]:

> **Constraint AASd-012:** if both the MultiLanguageProperty/value and the MultiLanguageProperty/valueId are present, the meaning must be the same for each string in a specific language, as specified in MultiLanguageProperty/valueId.

Die Bedeutung einer Zeichenkette lässt sich nicht deterministisch in mehreren Sprachen überprüfen. Zur Überprüfung ließe sich ein externer Übersetzungsservice nutzen. Allerdings ist dieser insbesondere bei längeren Zeichenketten nicht deterministisch. Außerdem ist es für Nutzer schwierig, die Instanz einer Verwaltungsschale zu korrigieren, die aufgrund dieses Constraints zurückgewiesen wurde.

Bei allen anderen, Constraints handelt es sich um **komplexe Constraints**. Ein Beispiel hierfür ist AASd-128 [3, Seite 93]:

> **Constraint AASd-128:** For model references, i.e. References with Reference/type = ModelReference, the Key/value of a Key preceded by a Key with Key/type=SubmodelElementList is an integer number denoting the position in the array of the submodel element list.

Ein komplexer Constraint lässt sich deterministisch prüfen. Dafür müssen aber mehrere Attributwerte in Beziehung zueinander gesetzt werden, teilweise sogar mit zusätzlichen arithmetischen Operationen.

Tabelle 3.1: Constraints für passive Verwaltungsschalen

Bezug	Constraint	Art
AdministrativeInformation	AASd-005	bedingt
HasSemantics	AASd-118	bedingt
Qualifier	AASd-006	komplex
	AASd-020	wertbezogen
Referable	AASd-002	wertbezogen
AssetInformation	AASd-131	bedingt
SpecificAssetId	AASd-133	wertbezogen
DataElement	AASd-090	wertbezogen
Entity	AASd-014	bedingt
MultiLanguageProperty	AASd-012	nicht prüfbar
Operation	AASd-134	komplex
Property	AASd-007	nicht prüfbar
	AASd-107	komplex
	AASd-114	komplex
	AASd-115	komplex
	AASd-108	komplex
	AASd-109	komplex
References	AASd-121	wertbezogen
	AASd-122	wertbezogen
	AASd-123	wertbezogen
	AASd-124	komplex
	AASd-125	bedingt
	AASd-126	komplex
	AASd-127	komplex
	AASd-128	komplex
Referable, Identifiable	AASd-117	bedingt
	AASd-120	wertbezogen
	AASd-022	komplex
Qualifier	AASd-021	komplex
	AASd-119	wertbezogen
	AASd-129	wertbezogen
Extension	AASd-077	komplex
AssetInformation	AASd-116	wertbezogen
Types	AASd-130	wertbezogen

3.3 Passive Verwaltungsschalen

Bei passiven Verwaltungsschalen [3] handelt es sich un Instanzen von Digitalen Zwillingen, die als Datei serialisiert wurden. Der Vorteil dieser Erscheinungsform liegt in ihrer Einfachheit: eine Datei kann über verschiedenste Wege verteilt werden (Mail, physischer Datenträger, ...) und benötigt keine besondere Infrastruktur. Dafür kann eine Datei keine dynamischen Daten des Digitalen Zwillings enthalten, sondern lediglich statische Daten vom Zeitpunkt der Erstellung der Datei.

Für die Verwaltungsschale sind Serialisierungen in folgenden Formaten spezifiziert:

- **XML und JSON**: eXtensible Markup Language (XML) [30] und JavaScript Object Notation (JSON) [31] sind generisches Formate zum Austausch von Daten.
- **RDF**: Resource Description Framework (RDF) [58] ist eine graphbasierte Repräsentation zum Ableiten von semantischen Beziehungen.
- **AutomationML**: AutomationML [59] ist ein Format für den Austausch von Planungsdaten während des Engineerings.
- **OPC UA**: OPC Unified Architecture (OPC UA) [60] ist ein Standard zum Austausch von Maschinendaten aus der Produktion.
- **AASX**: Ist ein Containerformat, das Verwaltungsschalen serialisiert als XML / JSON sowie ergänzende Dateien enthält.

Für passive Verwaltungsschalen werden Serialisierungen in XML, JSON und AASX verwendet. Daher werden im Rahmen dieser Arbeit lediglich diese Formate betrachtet. Eine besondere Rolle nimmt dabei die JSON Serialisierung ein, da sie für die Nutzdaten von reaktiven Verwaltungsschalen verwendet wird.

3.3.1 Mappings

Das Metamodell der Verwaltungsschale ist technologieneutral. Für die Serialisierung legt die Spezifikation Mappings fest, die zusammen mit entsprechenden Schemabeschreibungen veröffentlicht werden [61]. Zur Illustration der Mappings wird im Folgenden die Beispielinstanz aus Abbildung 3.7 verwendet. Diese genügt dem Metamodell wie in Abschnitt 3.1 beschrieben. Die Beispielinstanz besteht aus einer Verwaltungsschale mit der ID `www.example.com/ids/1` und einem Submodel mit der ID `www.example.com/ids/3`. Das Submodel enthält als einziges Submodelelement eine Property mit dem Wert `example_value`.

Im Allgemeinen verwenden alle Serialisierungen die `Environment` (siehe Abschnitt 3.1) als Einstiegspunkt für die Serialisierung. Alle identifizierbaren Elemente werden darin abgelegt. An allen anderen Stellen werden entsprechende Referenzen verwendet.

Listing 3.1 zeigt, wie die Beispielinstanz als JSON serialisiert wird. Dabei gelten folgende Mappings: Alle Klassen werden als JSON Objekte serialisiert und die Attribute werden zu Properties im JSON Objekt. Dabei werden alle Attribute mit einer Kardinalität > 1 als Arrays abgebildet und ihr Name in den Plural gesetzt (z.B. `submodels`). Schließlich wird noch das spezielle Attribut `modelType` eingeführt. Dieses Attribut dient als Diskriminator mit folgendem Zweck: An den Stellen, an denen im Metamodell lediglich eine abstrakte

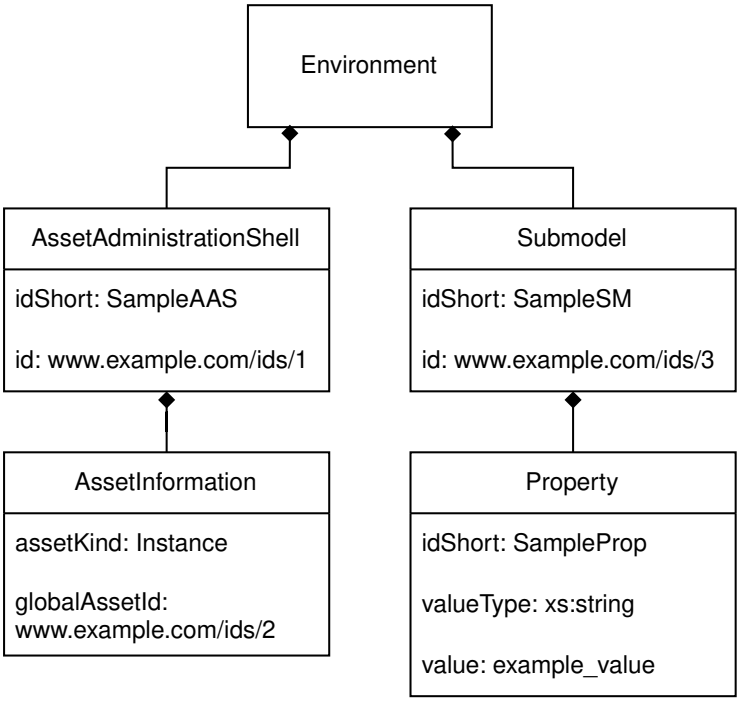


Abbildung 3.7: Beispielinstanz einer Verwaltungsschale

Klasse als Typ angegeben ist (z.B. `submodelElements: SubmodelElement`) wird bei der Serialisierung eine konkrete Instanz einer Kindklasse abgelegt. Der Typ der Kindklasse wird dann beim Deserialisieren durch den Diskriminator festgelegt.

Ähnliche Mappings gibt es auch für die XML Serialisierung. Listing 3.2 zeigt, wie die Beispielinstanz als XML serialisiert wurde. Dabei gelten folgende Mappings: Alle Klassen werden als Inhalt eines gleichnamigen Tags abgelegt, z.B. `Submodel` als `<submodel>...</submodel>`. Für jedes Attribut der Klasse wird dann ein passender Tag eingefügt. Im Gegensatz zur JSON Serialisierung ergibt sich der Diskriminator für abstrakte Klasse somit implizit aus dem Namen des umschließenden Tags.

3.3.2 AASX Format

Das AASX Format ist in Part 5 der AAS Spezifikation [4] beschrieben. Es dient dazu, mehrere Verwaltungsschalen samt ergänzenden Dateien (wie Handbücher) in einer Datei zu bündeln und somit die Handhabung zu vereinfachen. Das AASX Format basiert auf den Open Packaging Conventions (OPC[1]) [62].

Die OPC sind ein Containerformat, d.h. sie bündeln beliebige Dateien in einem zip-Archiv [63]. Um bestimmte Dateien in diesem Container auffinden zu können, definieren die OPC das Konzept von Beziehungen (*engl.*, Relationships). Jede Datei im OPC Container steht dabei in Beziehungen zu anderen Dateien im Container. Folglich wird dadurch ein Beziehungsgraph aufgebaut, bei dem die Knoten den Dateien und die Kanten den Beziehungen entsprechen.

Abbildung 3.8 zeigt den Beziehungsgraphen für AASX-Pakete. Wesentlich ist die Beziehung `<rel_aas>/aasx-origin`, die auf den Oberknoten für alle Verwaltungsschalen im

[1] trotz derselben Abkürzung nicht zu verwechseln mit Open Platform Communications

Listing 3.1: Beispiel für eine serialisierte Verwaltungsschale (JSON, formatiert als YML)

```
assetAdministrationShells:
- modelType: AssetAdministrationShell
  idShort: SampleAAS
  id: www.example.com/ids/1
  assetInformation:
    assetKind: Instance
    globalAssetId: www.example.com/ids/2
submodels:
- modelType: Submodel
  idShort: SampleSM
  id: www.example.com/ids/3
  submodelElements:
  - modelType: Property
    idShort: SampleProp
    valueType: xs:string
    value: example_value
```

Listing 3.2: Beispiel für eine serialisierte Verwaltungsschale (XML)

```
<environment xmlns="https://admin-shell.io/aas/3/0">
  <assetAdministrationShells>
    <assetAdministrationShell>
      <idShort>SampleAAS</idShort>
      <id>www.example.com/ids/1</id>
      <assetInformation>
        <assetKind>Instance</assetKind>
        <globalAssetId>www.example.com/ids/2</globalAssetId>
      </assetInformation>
    </assetAdministrationShell>
  </assetAdministrationShells>
  <submodels>
    <submodel>
      <idShort>SampleSM</idShort>
      <id>www.example.com/ids/3</id>
      <submodelElements>
        <property>
          <idShort>SampleProp</idShort>
          <value>example_value</value>
          <valueType>xs:string</valueType>
        </property>
      </submodelElements>
    </submodel>
  </submodels>
</environment>
```

Listing 3.3: Beispiel für eine `.rels` Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<Relationships xmlns="http://schemas.openxmlformats.org/packages/...">
  <Relationship
    Type="http://www.admin-shell.io/aasx/relationships/aasx-origin"
    Target="/aasx/aasx-origin" />
</Relationships>
```

Paket zeigt. Dieser Oberknoten wiederum hat Beziehungen `<rel_aas>/aas-spec` zu den eigentlichen Verwaltungsschalen. Diese sind XML- oder JSON-formatiert, wie im vorherigen Abschnitt beschrieben. Zu jeder Verwaltungsschale kann es auch noch ergänzende Dateien geben, die mit der Beziehung `<rel_aas>/aas-suppl` markiert werden.

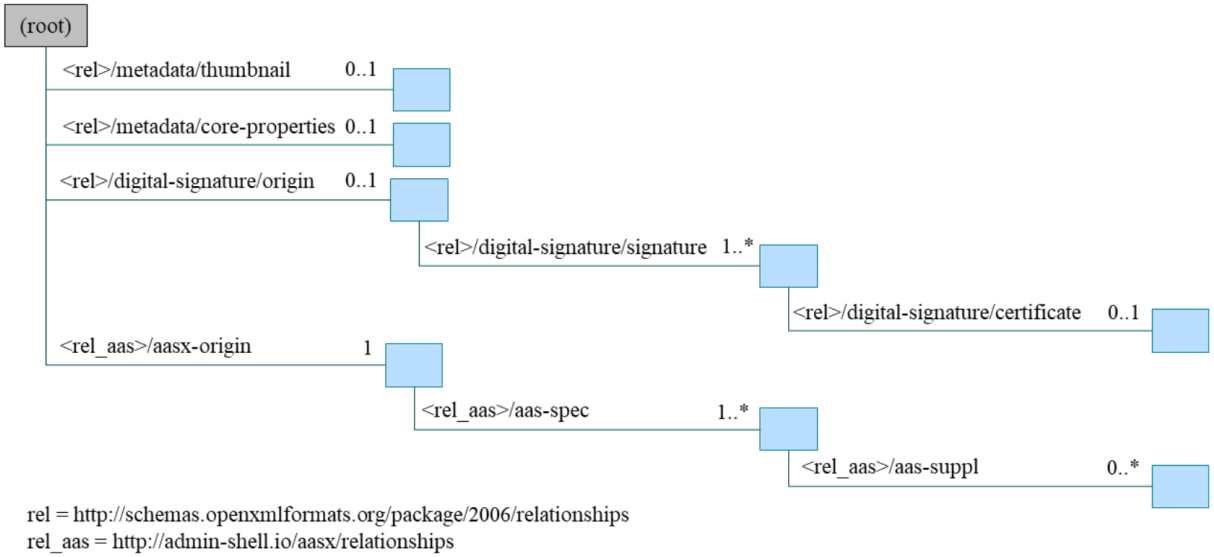


Abbildung 3.8: Beziehungsgraph für AASX Dateien (aus [4, Seite 17])

Der Beziehungsgraph wird im Standard der OPC als *logisches* Modell bezeichnet, das zunächst unabhängig von der eigentlichen Serialisierung existiert. Im darauf aufbauenden *physischen* Modell beschreibt der OPC-Standard die Abbildung des Beziehungsgraphen in einem zip-Archiv. Dies funktioniert wie folgt: Zu jedem Knoten im Graph mit der Datei `<Datei>` gibt es im selben Verzeichnis eine Datei namens `.rels/<Datei>.rels`. Die Datei ist XML formatiert. Ein Beispiel zeigt Listing 3.3. `Type` gibt die Art der Beziehung an und `Target` die Zieldatei, mit der die Beziehung besteht.

Abbildung 3.9 zeigt beispielhaft den Inhalt eines AASX-Paketes. Den Einstiegspunkt bildet dabei die Datei `_rels/.rels`, die auf die Datei `/aasx/assx-origin` verweist. Diese wiederum ist leer und dient lediglich als Oberknoten für die einzelnen Verwaltungsschalen, die in `/aasx/_rels/aasx-origin.rels` aufgelistet werden. Im Beispiel findet sich eine Verwaltungsschale unter `/aasx/files/the_aasx.xml`.

Neben den eigentlichen Verwaltungsschalen unterstützt das OPC-Format auch ein Thumbnail für das Paket sowie Signaturen und Verschlüsselung der Inhalte. Konformitätsprüfungen von Signaturen und Verschlüsselungen sind nicht Teil dieser Arbeit. Nähere Erläuterungen finden sich in den entsprechenden Standards [4, 62].

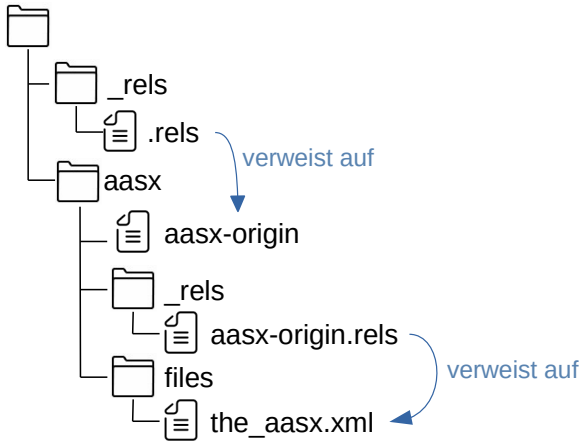


Abbildung 3.9: Inhalt eines AASX Containers (aus [5])

3.3.3 Teilmodelle

Teilmodelle (*engl.* Submodel Templates) [14, 64] bilden die Grundlage für semantische Interoperabilität der Verwaltungsschale. Dafür legen sie Zusammenstellung, Struktur und Bedeutung von Submodellelementen fest. Tabelle 3.2 zeigt eine Auflistung einiger Teilmodelle. Wenn ein Submodel in einer Instanz einer Verwaltungsschale ein Teilmodell implementiert, dann müssen alle Submodel-Elemente vorhanden sein, die das Teilmodell vorgibt. Ein Submodel implementiert höchstens ein Teilmodell. Innerhalb einer Verwaltungsschale können aber durchaus mehrere Teilmodelle implementiert werden. Um das Teilmodell zu identifizieren, das ein Submodel implementiert, wird die Semantic ID herangezogen. Im Rahmen dieser Arbeit werden exemplarisch die Teilmodelle *Contact Information* und *Digital Nameplate* betrachtet.

Tabelle 3.2: Einige Teilmodelle der Verwaltungsschale [14]

Teilmodell	Beschreibung
Bills of Material	Materiallisten
Carbon Footprint	Austausch von Daten über CO2 Emissionen
Contact Information	Ablage von Kontaktdaten
Digital Nameplate	Digitales Typenschild mit Informationen über das Asset
Handover Documentation	Übergabedokumente, z.B. beim Kauf von Komponenten
Technical Data	Daten von technischem Equipment

Zur einfachereren Beschreibung der Teilmodelle werden einige Konventionen in den Spezifikationen verwendet [6, 7, 65]. So werden die einzelnen Merkmale Submodellelementen zugeordnet. Sie werden über vordefinierte Semantic IDs identifiziert. Daneben finden sich in den UML-Klassendiagrammen zu den Merkmalen die `idShort`s. Ferner sind die Typen der Submodellelemente in Kurzschreibweise angegeben. So bezeichnet `File` beispielsweise ein Submodellelement vom Typ `File`, `LangString` bezeichnet ein Submodellelement vom Typ `MultiLanguageProperty`. Für Submodellelemente vom Typ `Property` gibt es zusätzlich dedizierte Kurzschreibweisen: So bezeichnet `String` ein Submodellelement der Klasse `Property`, wobei der `valueType` auf `xs:string` gesetzt ist, bei `Date` ist `valueType`

auf `xs:date` gesetzt. Teilweise sind idShorts mit dem Suffix {`00`} angegeben. Dies bedeutet, dass dieses Merkmal mehrfach vorkommen darf. Die idShort wird dann mit einer zweistelligen Zahl beendet, damit sie eindeutig ist. Dadurch können Listen von Submodel-Elementen gebildet werden.

Contact Information

Das Teilmodell *Contact Information* dient dem Ablegen von Kontaktinformationen zu einem Asset [6]. Dadurch kann sichergestellt werden, dass bei Bedarf (z.B. bei einer Störung) passende Verantwortliche direkt kontaktiert werden können. Es hat die Semantic ID `https://admin-shell.io/zvei/nameplate/1/0/ContactInformations`. Das UML-Diagramm des Teilmodells zeigt Abbildung 3.10. Es besteht aus einer Liste von Kontaktadressen, wobei das Merkmal `RoleOfContactInformation` die Rolle des Kontaktes angibt. Hierfür definiert die Spezifikation eine vorgegebene Liste von gültigen IRDIs:

- Administrativer Kontakt (0173-1#07-AAS927#001)
- Geschäftlich (0173-1#07-AAS928#001)
- Anderer Kontakt (0173-1#07-AAS929#001)
- Kontakt für Gefahrgüter (0173-1#07-AAS930#001)
- Technischer Kontakt (0173-1#07-AAS931#¸001)

Für jeden Kontakteintrag gibt es eine Reihe von Merkmalen, die in der Verwaltungsschale hinterlegt werden können. In den Merkmalen `FirstName`, `MiddleName` und `Title` kann der Name des Kontaktes hinterlegt werden. Die Adresse des Kontaktes wird in den Merkmalen `StateCounty`, `Street`, `POBox` sowie `Zipcode` hinterlegt. Abschließend werden die eigentlichen Kontaktdaten in `Phone`, `Fax` sowie `Email` hinterlegt. Einige Merkmale, wie z.B. `Phone` und `Email` enthalten wiederum weitere Untermerkmale.

Digital Nameplate

Das Teilmodell *Digital Nameplate for Industrial Equipment* dient dem Ablegen von digitalen Typenschildern [7]. Das Teilmodell hat die Semantic ID `https://admin-shell.io/zvei/nameplate/2/0/Nameplate`. Es ermöglicht damit die Umsetzung der EU Maschinen Direktive 2006/42/EC Das UML-Diagramm zeigt Abbildung 3.11. Das Teilmodell enthält diverse Felder, um den Hersteller und eine Maschine zu identifizieren [15]:

- `ManufacturerName`: Name des Herstellers
- `SerialNumber`: die Seriennummer der Maschine
- `YearOfConstruction` und `DateOfManufacture`: Daten der Herstellung der Maschine
- `HardwareVersion`, `FirmwareVersion` und `SoftwareVersion`: Versionsnummern von verwendeten Komponenten.

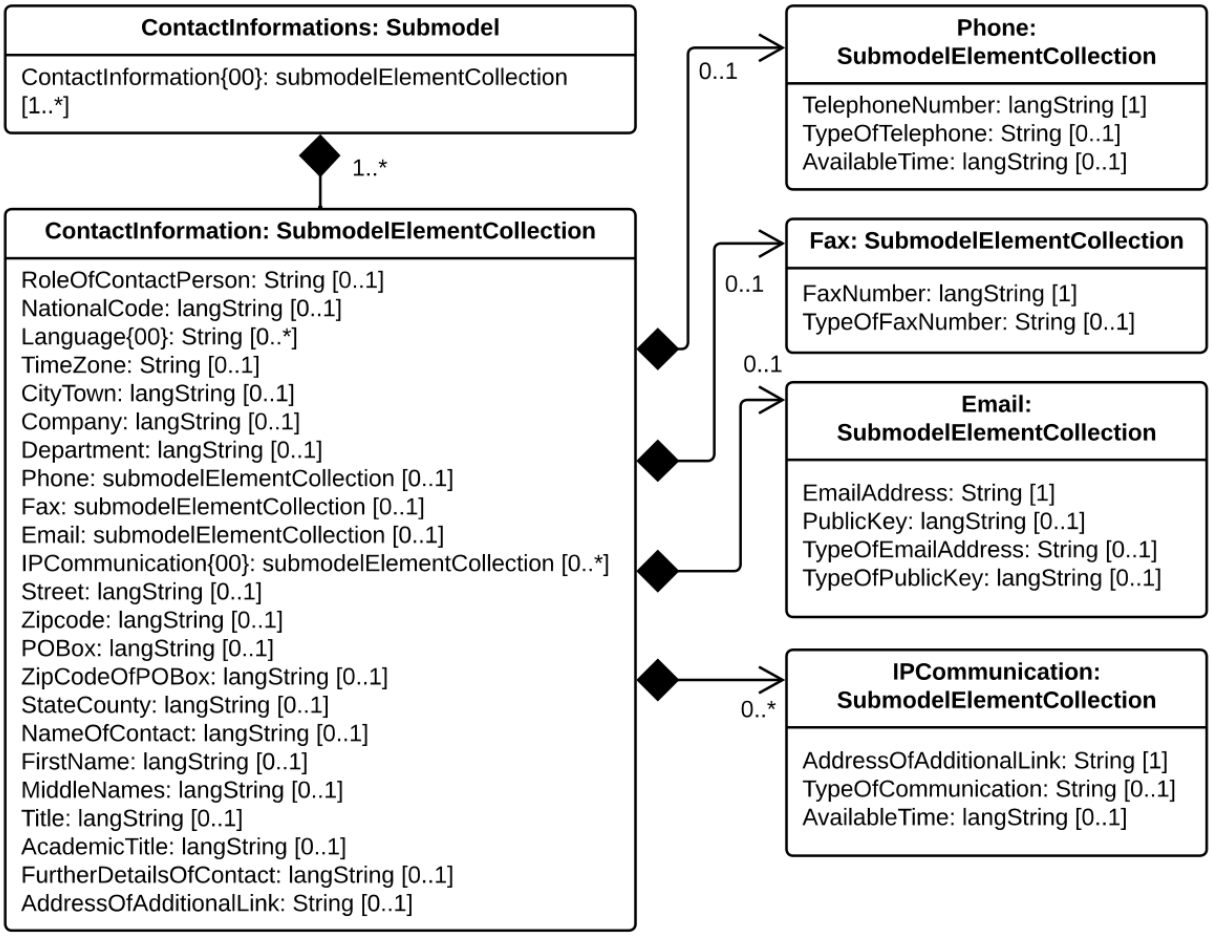


Abbildung 3.10: Teilmodell Contact Information (aus [6])

Das Merkmal `ContactInformation` enthält Kontaktinformationen zum Hersteller. Es verweist wiederum auf das Teilmodell Contact Information (beschrieben im vorherigen Abschnitt). Darüber hinaus spezifiziert das Teilmodell einige Zusatzbedingungen, z.B. dass von den beiden Merkmalen `ManufacturerProductFamily` und `ManufacturerProductType` mindestens eines angegeben werden muss.

3.4 Reaktive Verwaltungsschalen

Während passive Verwaltungsschalen lediglich statische Informationen zum Zeitpunkt ihrer Erstellung enthalten, erlauben reaktive Verwaltungsschalen den Zugriff auf dynamische Inhalte. In der Spezifikation [8] werden zunächst die logischen Operationen auf reaktiven Verwaltungsschalen technologieneutral beschrieben. Im Anschluss werden diese Operationen auf HTTP/REST gemappt (siehe auch Unterabschnitt 2.2.1).

3.4.1 Operationen

Operationen bilden die atomare Einheit, in der die Aktionen mit reaktiven Verwaltungsschalen spezifiziert sind. Eine Operation hat genau eine atomare Funktion. Dies kann beispielsweise das Abrufen aller Submodelle oder das Schreiben einer Verwaltungsschale

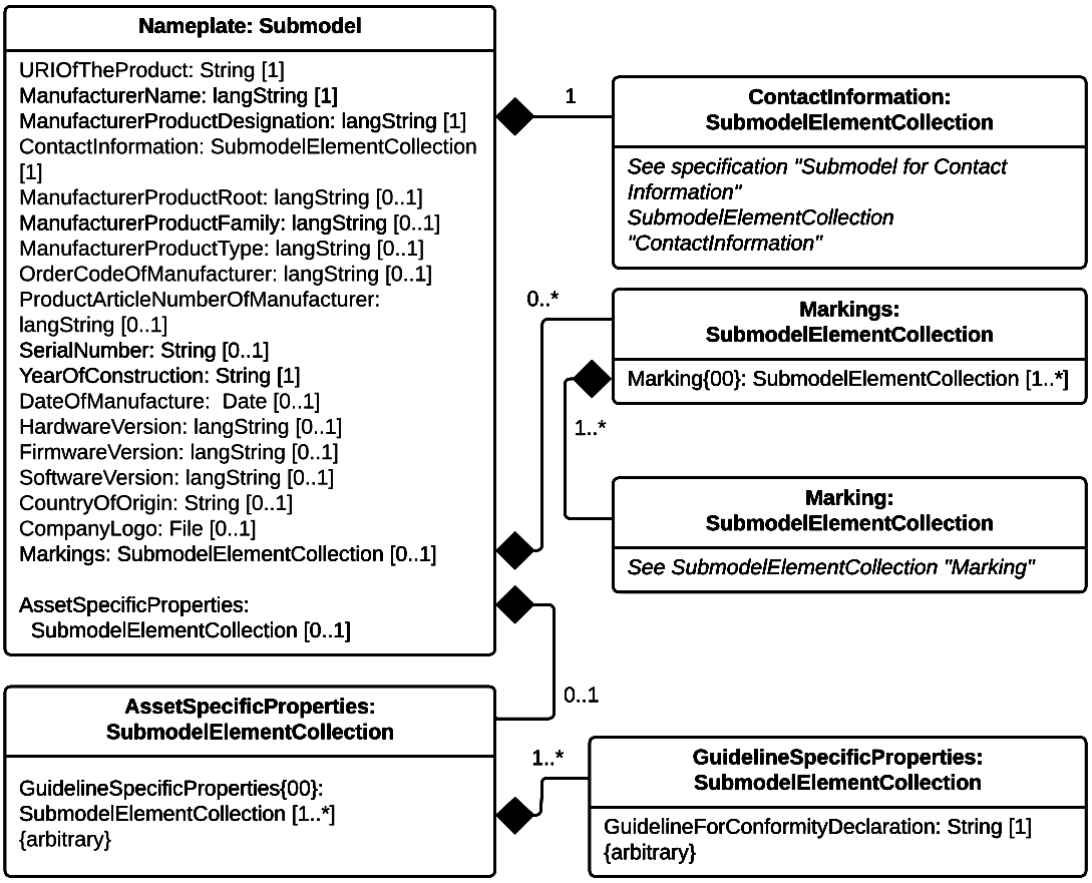


Abbildung 3.11: Teilmodell von Digital Nameplate (aus [7])

sein. Operationen haben einen (teilweise leeren) Satz an Eingabeparametern und liefern nach Ausführung einen Statuscode und ggf. Antwortdaten.

Neben den spezifischen Parametern gibt es auch einige generische Parameter, die bei vielen Operationen übergeben werden können. Für nahezu alle Operationen, die Listen von Entitäten (wie z.B. Submodelle oder Verwaltungsschalen) auslesen, können **Paging-Parameter** übergeben werden. Über Paging können Listen seitenweise ausgelesen werden. Dies ist insbesondere dann sinnvoll, wenn die Listen sehr groß werden. Für das Paging existieren zwei Parameter: `limit` und `cursor`. Mit dem Parameter `limit` wird angegeben, auf wieviele Einträge die Liste in der Antwort begrenzt werden soll. D.h., wird `limit=10` gesetzt, dann liefert der Server Listen mit maximal 10 Einträgen. Zusätzlich liefert der Server einen Wert für `cursor`, falls noch mehr Einträge in der Liste existieren. Ruft der Client die Operation erneut auf und übergibt dabei zusätzlich den `cursor`, dann werden die nächsten `limit` Einträge aus der Liste zurückgeben. Dies kann der Client solange wiederholen, bis der Server keinen `cursor` mehr zurückliefert. Damit signalisiert der Server, dass das Ende der Liste erreicht wurde. Somit fungiert der `cursor` wie ein Offset in die zu traversierende Liste. Wie der `cursor` genau implementiert ist, bleibt den Entwicklern überlassen. Ein Client hat lediglich die Aufgabe, den Wert bei jedem Aufruf exakt zu spiegeln.

Ferner können bei vielen Operationen **Serialisierungsmodifier** übergeben werden. Dabei handelt es sich um die Parameter `level`, `extent` und `content`. Mit `level` kann gesteu-

ert werden, wie tief verschachtelte Hierarchien von Submodellelementen traversiert werden. Dabei liefert `level=deep` die komplette Hierarchie, `level=core` nur ein Submodellelement mit etwaigen direkten Kindelementen. Mit dem Parameter `extent` können Submodellelemente vom Typ `Blob` gesteuert werden. Für `extent=WithoutBLOBValue` werden Blobs ohne Wert zurückgeliefert, für `extent=WithBLOBValue` mit Wert. Damit kann die Größe der Antwort reduziert werden, was z.B. für IoT-Geräte wichtig sein kann. Ferner kann mit dem Modifier `content` die Art der Antwort gesteuert werden. Es existieren fünf verschiedene Parameterwerte, die die Antwort grundlegend verändern:

- **Normal** gibt die Standardserialisierung als JSON zurück
- **Metadata** nur Metadaten werden zurückgegeben
- **Value** nur die Werte werden zurückgegeben (Value-Only Serialisierung [8])
- **Reference** Referenzen werden zurückgegeben
- **Path** gibt den idShort-Pfad von Submodellelementen zurück

Die Serialisierungsmodifier können unter bestimmten Einschränkungen miteinander kombiniert werden. Die genauen Bedingungen finden sich in der Spezifikation [8].

3.4.2 Interfaces

Die einzelnen Operationen, die für reaktive Verwaltungsschalen existieren, werden in Interfaces gruppiert. Folgende Interfaces sind spezifiziert:

- AAS Interfaces bieten Zugriff auf die eigentlichen Nutzdaten:
  - **Asset Administration Shell Interface** bietet Zugriff auf die Daten einer Verwaltungsschale wie insbesondere alle Submodellreferenzen.
  - **Submodel Interface** erlaubt den Zugriff auf die Elemente eines Submodells wie beispielsweise Dateien und Operationen.
  - **Serialization Interface** enthält nur eine Operation, die eine Serialisierung von Verwaltungsschalen und Submodellen liefert.
  - **AASX File Server Interface** greift auf Verwaltungsschalen in Form von AASX-Paketen zu (siehe Unterabschnitt 3.3.2).
- Registration Interfaces erlauben das Nachschlagen von Endpunkten:
  - **Asset Administration Shell Registry Interface** liefert den Endpunkt für eine bestimmten Verwaltungsschalen-Deskriptor.
  - **Submodel Registry Interface** liefert den Endpunkt für einen bestimmten Submodell-Deskriptor.
- Repository Interfaces erlauben den Zugriff auf einzelne Entitäten, wenn auf einem Server mehrere Entitäten gleichzeitig abgelegt werden:
  - **Asset Administration Shell Repository Interface** ermöglicht den Zugriff auf einzelne Verwaltungsschalen in einem Verwaltungsschalen-Repository.

  - **Submodel Repository Interface** ermöglicht den Zugriff auf einzelne Submodelle in einem Submodell-Repository.
  - **Concept Description Interface** ermöglicht den Zugriff auf einzelne Konzeptbeschreibungen in einem Konzeptbeschreibungs-Repository.

- **Asset Administration Shell Basic Discovery Interface** ermöglicht das Auffinden einer Verwaltungsschale für ein bestimmtes Asset.

- **Description Interface** hat nur eine Operation, die eine Liste von Features zurückliefert, die ein Server unterstützt.

Diese Operationen sind zunächst technologieunabhängig definiert. Im Standard [8] wird ein Mapping auf HTTP/REST definiert. Mappings auf weitere Technologien sollen in Zukunft folgen. Im Rahmen dieser Arbeit wird daher nur die HTTP/REST Schnittstelle betrachtet.

3.4.3 Zusammenhang der Interfaces

Der Zusammenhang zwischen den einzelnen Interfaces soll im Folgenden anhand eines Beispielanwendungsfalls illustriert werden. Dieser ist in Abbildung 3.12 dargestellt.

Zunächst fragt der Client bei der AAS Registry nach einer Verwaltungsschale mit einer bekannten Asset-ID. Die Asset-ID kann der Client z.B. durch Scannen eines QR-Codes an einer Maschine erhalten haben. Hierbei wird die Operation `GetAasDescriptorById` aus dem Asset Administration Shell Registry Interface genutzt. Als Ergebnis erhält der Client einen Verwaltungsschalen-Deskriptor. Dieser Deskriptor enthält die Adresse des Endpunktes, an dem die Verwaltungsschale abgelegt ist. Dort fragt der Client die Daten der Verwaltungsschale ab, mithilfe der Operation `GetAssetAdmninistrationShell` aus dem Asset Administration Shell Interface.

Danach möchte der Client auch einige Submodelle abrufen. Einige davon liegen auf dem Server der Verwaltungsschale selbst, sodass der Client sie direkt mithilfe der Operation `GetSubmodel` aus dem Submodel Interface abrufen kann. Andere Submodelle werden von einem anderen Server bereitgestellt. Dessen Adresse findet der Client mithilfe einer Submodellregistry, bei der er mit der Operation `GetSubmodelDescriptorById` aus dem Submodel Registry Interface nach dem Endpunkt des Submodellservers fragt. Auch hier kann der Client das Submodel mithilfe der Operation `GetSubmodel` die Daten des Submodells erfragen.

3.4.4 Servicespezifikationen und Profile

Wie beschrieben, sind jedem Interface eine (nicht-leere, disjunkte) Menge von Operationen zugeordnet. Diese Zuordnung allein deckt reale Anwendungsfälle allerdings nicht ab, weshalb Part 2 der Spezifikation zusätzlich Servicespezifikationen definiert. Eine Servicespezifikation kombiniert ein oder mehrere Interfaces in einem funktionalen Zusammenhang. Abbildung 3.14 zeigt diesen Zusammenhang. Einige Interfaces sind direkt in einer Servicespezifikation enthalten (markiert mit einem x in Abbildung 3.14). Andere sind über Superpfade erreichbar in der Servicespezifikation enthalten (markiert mit einem s in Abbildung 3.14). Ein Superpfad erlaubt den Zugriff auf APIs, indem eine andere API

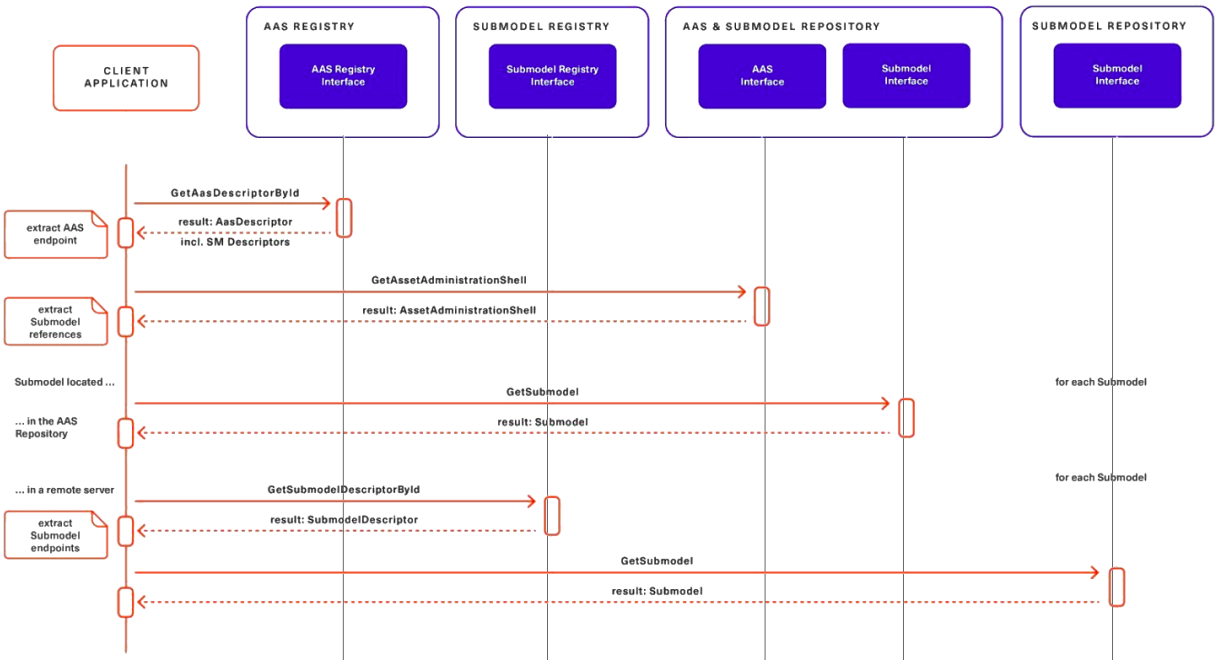


Abbildung 3.12: Zusammenhang von Interfaces (aus [8, Seite 154])

als Präfix verwendet wird. In folgendem Beispiel wird ein Submodel-Element abgerufen:

```
/shells/AAS-ID /submodels/SUBMOD-ID /submodel-elements/PATH
```

AAS Repository API | Submodel Repository API | Submodel API

Zunächst wird eine Verwaltungsschale aus einem Repository abgerufen, dann ein bestimmtes Submodell aus dieser Verwaltungsschale und zum Schluss ein bestimmtes Submodell-Element. Durch die Verwendung des Superpfades wird die Anzahl der nötigen Requests von drei auf einen reduziert.

Basierend auf den Servicespezifikationen werden in Part 2 Profile definiert, die die Servicespezifikationen einschränken. Z.B. gibt es für viele Servicespezifikationen entsprechende Profile, die die Operationen auf lesenden Zugriff einschränken. Profile sind für die Konformitätstests wichtig, da sie sich am ehesten an realen Anwendungsfällen orientieren. Ein Testsystem wird daher Testausführungen auf Basis von Profilen anbieten.

3.5 Ökosystem

Aufbauend auf den Standards hat sich über die Zeit ein breites Ökosystem an Software für die Verwaltungsschale entwickelt. Einen Überblick über die verschiedenen Arten von Komponenten zeigt Abbildung 3.15, einige Beispielkomponenten Tabelle 3.3.

3.5.1 Software Development Kits

Basis vieler Applikationen bilden Software Development Kits (SDKs). Ein SDK bietet diverse Grundfunktionen zum Arbeiten mit passiven Verwaltungsschalen [12]. Im Kern bilden SDKs das Metamodell der Verwaltungsschale ab (siehe Abschnitt 3.1). Sie stellen

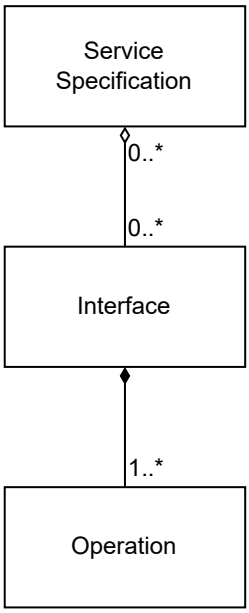


Abbildung 3.13: Operationen, Interfaces und Servicespezifikationen (aus [8])

Service Specifications: \ Contained APIs:	Asset Administration Shell API	Submodel API	AASX File Server API	Asset Administration Shell Registry API	Submodel Registry API	Asset Administration Shell Repository API	Submodel Repository API	Concept Description Repository API	Asset Administration Shell Basic Discovery API	Serialization API	Description API
Asset Administration Shell Service Specification	x	s								x	x
Submodel Service Specification		x								x	x
AASX File Server Service Specification			x								x
Asset Administration Shell Registry Serv. Spec				x	s						x
Submodel Registry Service Specification					x						x
Discovery Service Specification									x		x
Asset Administration Shell Repository Serv. Spec.	s	s				x	s			x	x
Submodel Repository Service Specification		s					x			x	x
ConceptDescription Repository Service Spec								x		x	x

Abbildung 3.14: Servicespezifikationen (aus [8, Seite 129])

dafür das Metamodell z.B. in Form von Klassen bereit. Darüber hinaus bieten SDKs Funktionen zum Serialisieren und Deserialisieren von Dateien mit Verwaltungsschalen. Je nach

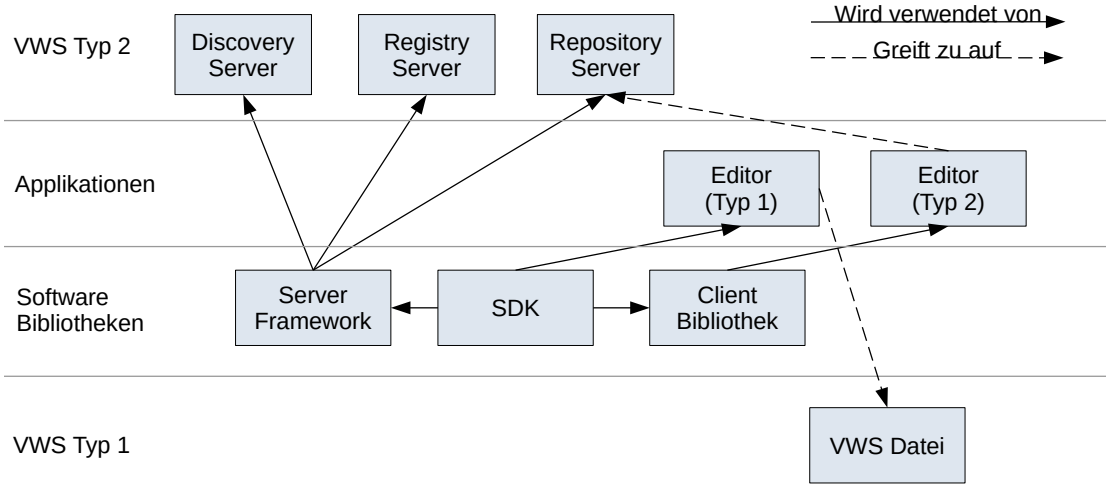


Abbildung 3.15: Komponenten des Verwaltungsschalen-Ökosystems (aus [9])

Tabelle 3.3: Beispiele für Komponenten des Verwaltungsschalen-Ökosystems

Kategorie	Name	Repository
SDK	Eclipse BaSyx	`github.com/eclipse-basyx`
	AAS4J	`github.com/eclipse-aas4j/aas4j`
	AAS Core Works	`github.com/aas-core-works`
Server	AASX Server	`github.com/eclipse-aaspe/server`
	Eclipse BaSyx Java V2 SDK	`github.com/eclipse-basyx/basyx-java-server-sdk`
	Eclipse BaSyx Python SDK	`github.com/rwth-iat/basyx-python-sdk`
	FA³ST	`github.com/FraunhoferIOSB/FAAAST-Service`
Applikation	AASX Package Explorer	`github.com/admin-shell-io/aasx-package-explorer`
	AAS Manager	`github.com/rwth-iat/aas_manager`
	AAS Portal	`github.com/FraunhoferIOSB/AASPortal`
	Frauenhofer AAS GUI	`github.com/eclipse-basyx/basyx-applications`
	AAS Core CLI Swiss Knife	`github.com/aas-core-works`

SDK werden weitere Funktionen implementiert, z.B. das Ablegen von Verwaltungsschalen in Datenbanken.

Über die Zeit haben sich diverse SDKs etabliert, die sich insbesondere in der verwendeten Programmiersprache unterscheiden [66]. Im Rahmen des Projektes **Eclipse BaSyx** wurden SDKs für diverse Programmiersprachen entwickelt wie z.B. BaSyx Python SDK oder BaSyx Java SDK. **AAS4J** ist eine weitere Java Implementierung des Metamodells. AAS4J unterstützt die Dateiformate AASX, JSON und XML. Im Rahmen von der informellen **AAS Core Works** Gruppe wurden diverse SDKs veröffentlicht, die aus einer Formalisierung des Metamodells generiert wurden. Zu den unterstützen Programmiersprachen zählen C#, Go, Python, JavaScript, Java und C++.

3.5.2 Server

Server implementieren einige oder alle Interfaces, die in Abschnitt 3.4 beschrieben sind. Die meisten Server sind weitreichend konfigurierbar, insbesondere welche Verwaltungsschalen

im Server abgelegt werden sollen.

Die offizielle Serverimplementierung ist der **AASX Server**. Im Rahmen des Projektes Eclipse BaSyx wurde das **Eclipse BaSyx Java V2 SDK**[2] entwickelt. Es besteht aus einigen vorgefertigten Serverkomponenten, die zusammen alle AAS Interfaces implementieren. Java V2 SDK verwendet intern das AAS4J SDK zur Implementierung des Metamodells. Außerdem kann es als Server Framework zur Implementierung eigener Server verwendet werden.

Ebenfalls im Rahmen des Projektes Eclipse BaSyx wurde das **Eclipse BaSyx Python SDK** entwickelt. Dieser Server implementiert den Asset Administration Shell Repository Service sowie den Submodel Repository Service. Die **Fraunhofer Advanced Asset Administration Shell Tools (FA³ST)** sind eine weitere Serverimplementierung. Es werden ähnliche Interfaces wie bei der Eclipse BaSyx Java Implementierung unterstützt.

Tabelle 3.4 zeigt eine Übersicht darüber, welche Server welche Servicespezifikationen unterstützen. Die Repository Servicespezifikationen (AAS, Submodell, Concept Description) werden von allen Servern unterstützt. Im AAS Repository ist der Service für einzelne Verwaltungsschalen (AAS Service) enthalten. Genauso ist im Submodel Repository der Service für einzelne Submodelle (Submodel Service) enthalten. Beide Einzelservices werden von den Servern nur indirekt über die entsprechenden Repository Services implementiert. Alle Server außer BaSyx Python implementieren darüber hinaus eine Registry für Verwaltungsschalen (AAS Registry) und Submodelle (Submodel Registry). Für letztere hat der AASX Server nur eine indirekte Unterstützung über die AAS Registry. Ferner bieten alle Server außer BaSyx Python einen Discovery Service an. Die AASX File Server Spezifikation wird nur vom AASX Server und von BaSyx Java implementiert.

Tabelle 3.4: Von Verwaltungsschalen-Servern unterstütze Profile

Servicespezifikation	AASX Server	BaSyx Java	BaSyx Python	FA³ST
AAS Repository	✓	✓	✓	✓
Submodel Repository	✓	✓	✓	✓
Concept Description Repo.	✓	✓	✓	✓
AAS Service	(✓)	(✓)	(✓)	(✓)
Submodel Service	(✓)	(✓)	(✓)	(✓)
AAS Registry	✓	✓	-	✓
Submodel Registry	(✓)	✓	-	✓
Discovery Service	✓	✓	-	✓
AASX File Server	✓	✓	-	-

✓ Implementiert (✓) Implementiert als Teil eines anderen Service - Nicht implementiert

3.5.3 Clientbibliotheken und Serverframeworks

Clientbibliotheken abstrahieren den Zugriff auf reaktive Verwaltungsschalen für Softwareanwendungen. Bisher gibt es wenige dedizierte Clientbibliotheken. Die meisten Clientanwendungen für reaktive Verwaltungsschalen implementieren den Zugriff direkt, ohne dafür auf eine externe Bibliothek zurückzugreifen. Serverframeworks bilden das Gegenstück: Sie

[2]Der Begriff *SDK* meint hier Serverframework

abstrahieren die HTTP/REST Schnittstelle von reaktiven Verwaltungsschalen zur Implementierung von Servern. Auch hier stellen bisher wenige Projekte ein dediziertes Serverframework bereit. Eine Ausnahme bildet das **BaSyx Java V2 Server SDK**: Es stellt sowohl eine Clientbibliothek als auch ein Serverframwork zur Verfügung. Beide sind für die Programmiersprache Java vorgesehen.

3.5.4 Applikationen

Applikationen ermöglichen dem Zugriff auf passive bzw. reaktive Verwaltungsschalen für Endnutzer. In der Regel zeichnen sie sich durch eine grafische Benutzeroberfläche aus. Für passive Verwaltungsschalen stellt die IDTA den **AASX Package Explorer** bereit. Abbildung 3.16 zeigt einen Screenshot. Dieser ermöglicht das Editieren von Verwaltungsschalendateien. Ähnliche Funktionalitäten bietet auch der **AAS Manager**. Dieser nutzt intern das BaSyx Python SDK. Abbildung 3.17 zeigt einen Screenshot.

Vergleichbare Editoren gibt es auch für reaktive Verwaltungsschalen. Beispielsweise ermöglicht das **AAS Portal** den Zugriff auf reaktive Verwaltungsschalen. Ähnliches bietet auch das **Frauenhofer AAS GUI**. Beide Editoren stehen als Webanwendung zur Verfügung. Das AAS Core Works Projekt stellt das **AAS Core CLI Swiss Knife** zur Verfügung. Dieses implementiert diverse Hilfsfunktionen zum Umgang mit Verwaltungsschalen in Form einer Konsolenanwendung.

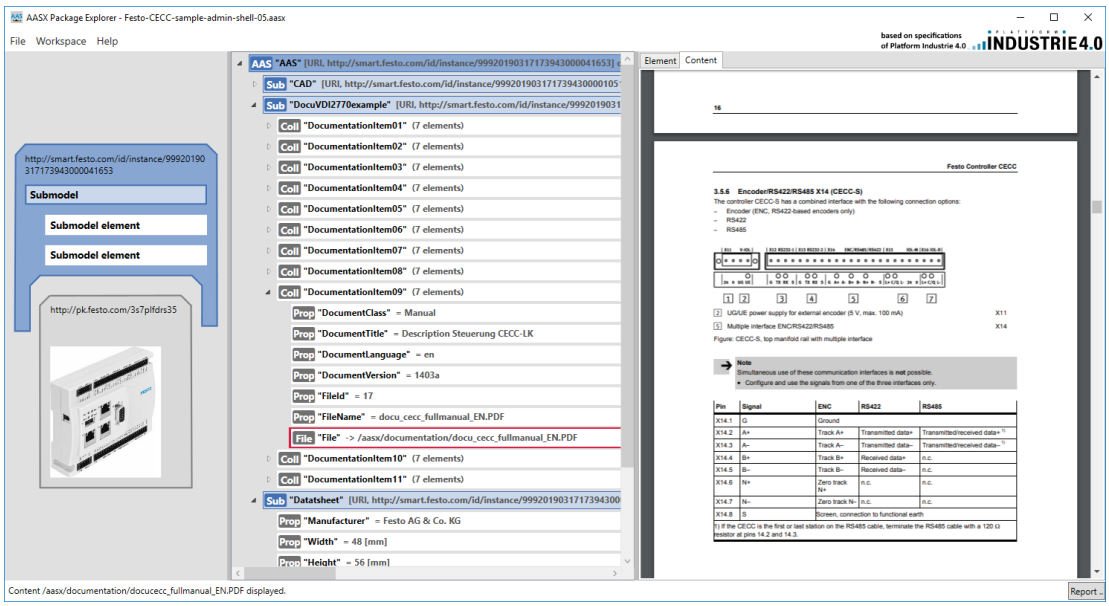


Abbildung 3.16: Screenshot des AASX Package Explorers [10]

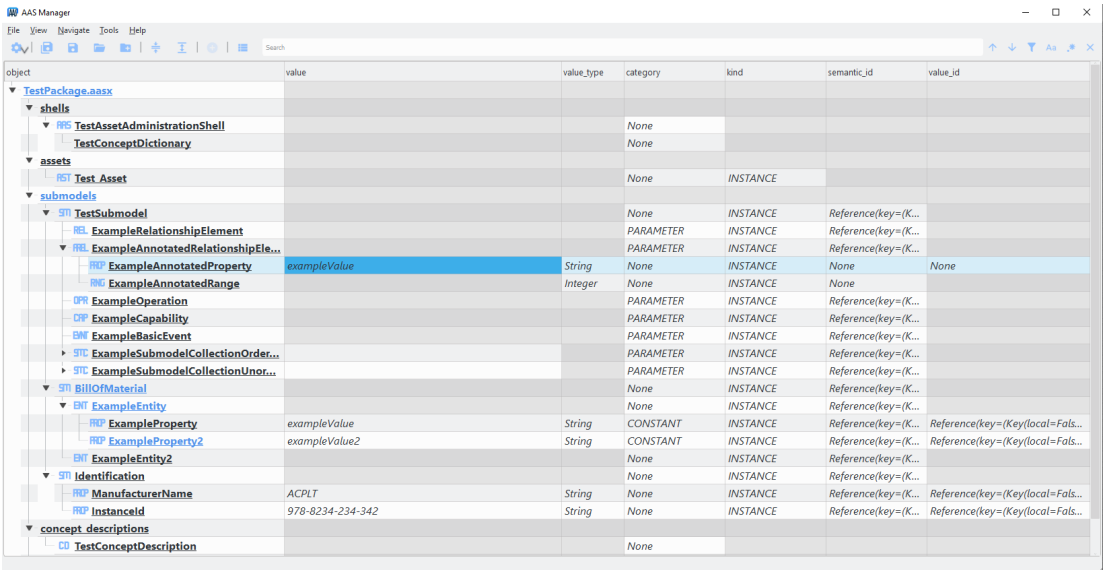


Abbildung 3.17: Screenshot des AAS Managers [11]

3.6 Fazit

Basierend auf der Analyse des Ökosystems lässt sich die erste Forschungsfrage beantworten:

> **FF1:** Wie ist das Softwareökosystem rund um die Verwaltungsschale aufgebaut? Welche Abhängigkeiten gibt es und welche grundlegenden Komponenten ergeben sich daraus, deren Konformität sicherzustellen ist?

Abbildung 3.15 zeigt die Komponenten des Verwaltungsschalenökosystems und die Abhängigkeiten zwischen diesen. Grundlage dieses Ökosystems bilden passive Verwaltungsschalen, die als Datei serialisiert wurden. Somit bilden diese die erste Komponente, deren Konformität sicherzustellen ist. Weiterhin zeigt sich, dass nahezu jede Softwarekomponente im Ökosystem auf ein SDK zurückgreift. Diese implementieren Grundfunktionen, insbesondere das Serialisieren und Deserialisieren von passiven Verwaltungsschalen. SDKs bilden somit die zweite zu überprüfende Komponente. Ferner stellen Server reaktive Verwaltungsschalen bereit. Diese bilden folglich die dritte Komponente im Ökosystem, deren Konformität sicherzustellen ist.

Bei den Servern ist es sinnvoll, einzelne Servicespezifikationen zu testen, da diese realen Anwendungsfällen entsprechen. Um möglichst viele Interfaces abzudecken, werden daher im Rahmen dieser Arbeit folgende Servicespezifikationen gewählt:

- Asset Administration Shell Repository Servicespezifikation
- Asset Administration Shell Registry Servicespezifikation
- Discovery Servicespezifikation
- ConceptDescription Repository Servicespezifikation

Dadurch können alle Interfaces abgedeckt werden, bis auf das *AASX File Server API* (siehe Abbildung 3.18). Gemäß Tabelle 3.4 wird diese API lediglich von zwei der vier betrachteten Servern unterstützt und daher für diese Arbeit nicht weiter berücksichtigt.

Service Specifications: \ **Contained APIs:**	Asset Administration Shell API	Submodel API	AASX File Server API	Asset Administration Shell Registry API	Submodel Registry API	Asset Administration Shell Repository API	Submodel Repository API	Concept Description Repository API	Asset Administration Shell Basic Discovery API	Serialization API	Description API
Asset Administration Shell Service Specification	x	s								x	x
Submodel Service Specification		x								x	x
AASX File Server Service Specification			x								x
Asset Administration Shell Registry Serv. Spec				x	s						x
Submodel Registry Service Specification					x						x
Discovery Service Specification									x		x
Asset Administration Shell Repository Serv. Spec.	s	s				x	s			x	x
Submodel Repository Service Specification		s					x			x	x
ConceptDescription Repository Service Spec								x		x	x

Abbildung 3.18: Abdeckung von Servicespezifikationen (nach [8, Seite 129])

Sinnvoll wäre es auch, Serverframeworks und Clientbibliotheken zu testen. Allerdings sind diese bisher sehr wenig verbreitet, sodass im Rahmen dieser Arbeit Server direkt getestet werden. Auch sollten Verwaltungsschalenapplikationen auf Konformität geprüft werden. Dies ist aus zwei Gründen außerhalb des Rahmens dieser Arbeit: Zunächst verwenden Applikationen in der Regel intern ein SDK, die bereits als zu testende Komponente identifiziert wurden. Außerdem ist der Testaufbau für Applikationen stark von der Art der Applikation abhängig (z.B. grafische oder Konsolenanwendung). Daher ist es nicht möglich, dies in einem generischen Testaufbau abzubilden und muss von den Entwicklern der Anwendung selbst übernommen werden.

4 Sicherstellung der Konformität von passiven Verwaltungsschalen

Passive Verwaltungsschalen bilden die Grundlage des Verwaltungsschalenökosystems. Dabei handelt es sich um als Datei serialisierte Instanzen des Metamodells. Eine detaillierte Beschreibung findet sich in Abschnitt 3.3. In diesem Kapitel wird ein Verfahren vorgestellt, um die Konformität von passiven Verwaltungsschalen gegenüber der Spezifikation der Verwaltungsschale zu überprüfen. Zentraler Bestandteil dieses Verfahrens bildet die Formalisierung des Metamodells und der Constraints mithilfe von JSON-Schema.

4.1 Verwandte Arbeiten

4.1.1 Softwarelösungen

Wie in Unterabschnitt 3.5.1 beschrieben, ist die Überprüfung der Konformität von passiven Verwaltungsschalen integraler Bestandteil von SDKs für die Verwaltungsschale. Im Wesentlichen lassen sich hierbei drei verschiedene Ansätze unterscheiden: Im einfachsten Fall überprüft das SDK lediglich die Konformität zum Metamodell. Es wird nicht überprüft, ob auch die Constraints erfüllt sind. Auf diese Weise ist beispielsweise das AAS4J SDK implementiert. Während dieser Ansatz vergleichsweise einfach umzusetzen ist und für einfache Applikationen ausreicht, ist damit die volle Überprüfung der Konformität nicht möglich.

Bei einigen SDKs, z.B. denen der AAS Core Works Gruppe [67], wird die Konformität in zwei separaten Schritten geprüft. Im ersten Schritt wird die passive Verwaltungsschale deserialisiert und als Objekthierarchie abgelegt. Falls Fehler gefunden werden, wird eine Exception geworfen. Im nächsten Schritt kann dann separat die Korrektheit der Constraints überprüft werden. Der Vorteil dieser Trennung in zwei Schritte besteht darin, dass sie dem Nutzer erlauben, die Objekte anzupassen, ohne auf Korrektheit der Constraints achten zu müssen. Dies ist beispielsweise bei der Implementierung eines Editors nützlich.

Alternativ erfolgt die Überprüfung auf Konformität zum Metamodell und den Constraints in einem gemeinsamen Schritt. Dies verhindert, dass invalide Daten im Speicher vorliegen, kann aber die Implementierung von Applikationen erschweren. Bei den SDKs des BaSyx Projektes [68] erfolgt die Überprüfung beispielsweise direkt beim Lesen/Schreiben von passiven Verwaltungsschalen. Basierend auf den SDKs wurden auch schon dedizierte Compliance Checker implementiert, z.B das *AAS Compliance Tool* [69].

Im Rahmen dieser Arbeit wurde erwogen, ein SDK zur Überprüfung der Konformität zu verwenden. Darauf wurde aber aus folgenden Gründen schließlich verzichtet:

- Jedes SDK implementiert immer genau eine Version der Spezifikation der Verwaltungsschale. Dadurch gestaltet sich das Überprüfung mehrerer Versionen der Verwaltungsschale schwierig bis unmöglich.

- Die meisten SDKs werfen eine Exception bei fehlender Konformität. Dadurch kann aber nur genau ein Fehler angezeigt werden anstatt *aller* fehlerhafter Stellen. Oft fehlen außerdem die genauen Pfade der fehlerhaften Stellen. Auch das Implementieren von Hinweisen und Warnungen gestaltet sich schwierig.
- Das Formalisieren der Spezifikation ermöglicht die Generierung von Beispielinstanzen, was für das Überprüfen von SDKs von zentraler Bedeutung ist (siehe Kapitel 5).

4.1.2 Schema-Sprachen

Zur Formalisierung von Metamodellen wurden diverse Schema-Sprachen entwickelt, von denen einige wichtige im Nachfolgenden vorgestellt werden.

Datenorientierte Schemasprachen

Daten-orientierte Schemasprachen sind an bestimmte Serialisierungsformate gekoppelt. So wurde zur Beschreibung des Inhaltes von XML-Dokumenten XSD (XML Schema Definition) [30] entwickelt. Folgendes Listing zeigt ein Beispiel-Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="persons">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="person">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="name" type="xs:string"/>
                            <xs:element name="age" type="xs:int"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Mithilfe von `xs:sequence` wird eine Sequenz von Tags beschrieben, hier `name` und `age`. Auch die Datentypen der in den Tags enthaltenen Strings werden mit formalisiert.

Analog zu XML wurde auch für JSON eine Schema-Sprache entwickelt, JSON-Schema [32]. Eine ausführliche Erläuterung von JSON-Schema findet sich in Abschnitt 2.1. JSON-Schema wird im Rahmen dieser Arbeit zur Formalisierung der Verwaltungsschale verwendet. Ein weiteres, weit verbreitetes Serialisierungsformat ist CSV [70]. Hierfür wurde die CSV Schema Language [71] veröffentlicht.

Graph-orientierte Schemasprachen

Andere Daten lassen sich durch Graphen, d.h. Knoten verbunden durch Kanten, beschreiben. Zum Modellieren von Softwaresystemen hat sich UML [72] etabliert. Ein Bestandteil der UML-Spezifikation sind UML-Klassendiagramme. Constraints zwischen Klassen lassen

sich hierbei mithilfe von OCL [73] beschreiben. Um Ontologien zu beschrieben, wurde das Resource Description Framework (RDF) [58] standardisiert. Diese kann mit Sprachen wie Web Ontology Language (OWL) [74] kombiniert werden, um Constraints zwischen den Knoten zu formalisieren.

Spezielle Schemasprachen

Für besondere Anwendungsbereiche wurden spezielle Schemasprachen entwickelt. Im Kontext von Web und IoT hat sich Protocol Buffers [75] etabliert. Hiermit werden Formate von Nachrichten beschrieben, die dann leichtgewichtig übertragen werden können:

```
message Person {
  optional string name = 1;
  optional int32 age = 2;
}
```

Im Rahmen von Kryptografie hat sich das ASN.1 Format [76] etabliert. ASN.1 wird verwendet, um beispielsweise das Format digitaler Zertifikate zu beschreiben:

```
Person ::= SEQUENCE {
  name IA5String,
  age INTEGER
}
```

Beide Schemasprachen eignen sich nur eingeschränkt für diese Arbeit, da sie sehr an bestimmte Anwendungsbereiche gekoppelt sind.

4.1.3 Formalisierungen des Metamodells

Die Spezifikation der Verwaltungsschale wurde im Rahmen einiger Projekte formalisiert. In [67] formalisieren die Autoren der AAS Core Works Gruppe das Metamodell und die Constraints mithilfe einer Untermenge der Programmiersprache Python. Mithilfe dieser Formalisierung generieren die Autoren beispielsweise die SDKs des Core Works Projektes. Außerdem werden aus dieser Formalisierung einige Formalisierungen in Schemasprachen generiert, darunter JSON-Schema für JSON-Serialisierungen sowie XSD für XML-Serialisierungen. Diese enthalten aber nicht die Überprüfung der Constraints. Diese Schemata werden als Teil der Spezifikation der Verwaltungsschale veröffentlicht [61]. Das JSON-Schema dient dabei als Grundlage für die Formalisierung in dieser Arbeit.

4.1.4 Teilmodelle

Die Überprüfung von Instanzen auf Konformität zu Teilmodellen wird in der Literatur bisher wenig behandelt. Die IDTA stellt selbst Formalisierungen von Teilmodellen bereit [14]. Hierfür werden spezielle Meta-Verwaltungsschalen genutzt, die Qualifier nutzen, um z.B. Kardinalitäten zu beschreiben. Diese Formalisierungen werden in [77] genutzt, um automatisch Validierungscode in der Programmiersprache Python zu erzeugen. In [15] stellen die Autoren ein Verfahren vor, das Teilmodelle mithilfe von JSON-Schema formalisiert. Diese Formalisierung nutzen die Autoren zum einen zum Überprüfen von bestehenden

Instanzen und zum anderen zum Erzeugen von Beispielinstanzen. Die in dieser Arbeit vorgestellte Methode zum Überprüfen von Instanzen der Verwaltungsschale auf Konformität zu Teilmodellen basiert wesentlich auf dem in [15] vorgestellten Ansatz.

4.2 Lösungsansatz

Die Architektur des im Rahmen dieser Arbeit entwickelten Verfahrens zeigt Abbildung 4.1. Es sind grundsätzlich drei Pfade möglich, die vom Serialisierungsformat der Instanz der Verwaltungsschale abhängen, die auf Konformität geprüft werden soll. Wurde die Instanz als JSON-Datei serialisiert, so wird sie direkt gegen das JSON-Schema geprüft. Für Verwaltungsschalen im XML-Format wird der Inhalt zunächst nach JSON umgewandelt und dann wie im ersten Fall geprüft. Für Verwaltungsschalen, die als AASX-Paket serialisiert wurden, muss zunächst das unterliegende OPC-Format geprüft werden und die Relationships bestimmt werden. Sind diese korrekt, werden die enthaltenen Verwaltungsschalen extrahiert und dann, je nach Format, wie in der vorherigen Fällen überprüft.

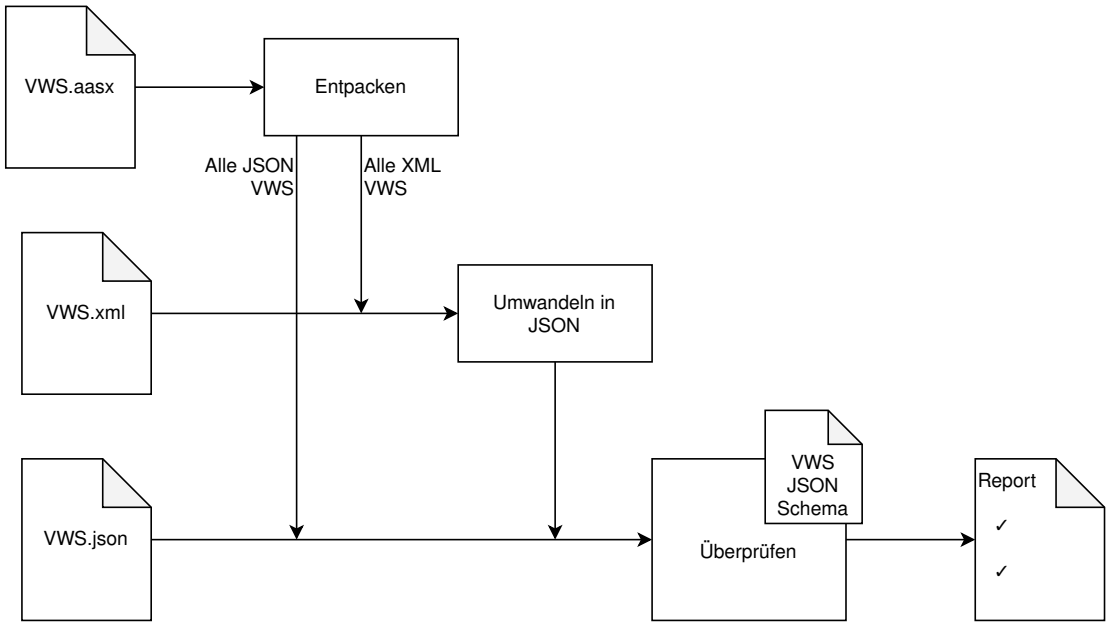


Abbildung 4.1: Lösungsansatz zum Überprüfen von passiven Verwaltungsschalen in verschiedenen Serialisierungen

4.2.1 Transformation von XML nach JSON

Serialisierungen der Verwaltungsschale im XML-Format können nicht mithilfe von JSON-Schema überprüft werden. Sie werden daher zunächst nach JSON konvertiert. Dies ist geschlossen und in einem Durchgang möglich mit dem Algorithmus, der in Listing 4.1 dargestellt ist. Dafür wird zunächst die Hilfsfunktion `get_single_child` definiert, die aus einem XML Element das erste und einzige Kindelement extrahiert. Z.B. würde die Funktion aus `<X><Y>...</Y></X>` das Element `<Y>...</Y>` extrahieren.

Anschließend werden Funktionen zum Konvertieren in bestimmte JSON-Datentypen definiert. Tabelle 4.1 zeigt einige Beispiele. `to_json_string` konvertiert ein XML-Element in

einen JSON-String, indem es den inneren Text extrahiert. Die Funktion `to_json_array` konvertiert ein XML-Element in eine Liste, indem die Kindelemente extrahiert werden. Das Array enthält XML-Elemente. Diese werden dann von der aufrufenden Funktion mittels `to_json` in JSON-Werte konvertiert. Für Objekte wird die Funktion `to_json_object` definiert. Diese Funktion iteriert über die Kindelemente und verwendet die Tags als Attributnamen im Ergebnis. Die Werte sind ebenfalls XML-Elemente, die von der aufrufenden Funktion nach JSON konvertiert werden. Zu beachten ist, dass dabei eine Sonderbehandlung für die Klassen `DataSpecificationContent` sowie `OperationVariable` nötig ist. Diese sind von der Spezifikation entsprechend vorgegeben [3].

Abschließend wird die Funktion `to_json` definiert, die ein XML-Element nach JSON konvertiert. Diese Funktion delegiert die Konvertierung je nach gewünschten Typ. Für die JSON-Datentypen `boolean`, `number` und `null` ist keine Konvertierung nötig, da diese Typen nicht im Metamodell der Verwaltungsschale vorkommen. Die Konvertierung kann fehlschlagen, z.B. kann `<value>something</value>` nicht in ein Objekt konvertiert werden. In diesem Fall wird die Verwaltungsschale als nicht konform abgewiesen.

Listing 4.1: Transformation von XML nach JSON

```
def get_single_child(xml):
  assert len(xml.children) == 1
  return xml.children[0]

def to_json_string(xml):
  return xml.text

def to_json_array(xml):
  return xml.children

def to_json_object(xml):
  if xml.tag == 'dataSpecificationContent':
    # Special handling for data specification content
    xml = get_single_child(xml)
  result = {}
  for child in xml.children:
    if xml.tag == 'OperationVariable' and child.tag == 'value':
      # Special handling for operation variable
      child = get_single_child(child)
    result[tag] = child
  return result

def to_json(xml, json_type):
  if json_type == "string":
    return to_json_string(xml)
  if json_type == 'array':
    return to_json_array(xml)
  if json_type == 'object':
    return to_json_object(xml)
```

Tabelle 4.1: Beispiele für die Konvertierung von XML nach JSON

Funktion	Eingabe	Ausgabe

`to_json_string`	`<X>`
	`  something`
	`</X>`
		`"something"`

`to_json_array`	`<X>`
	`  <Y>...</Y>`
	`  <Y>...</Y>`
	`</X>`
		`[`
		`  <Y>...</Y>,`
		`  <Y>...</Y>`
		`]`

`to_json_object`	`<X>`
	`  <a>`
	`  <AA>...</AA>`
	`  </a>`
	`  <b>`
	`  <BB>..</BB>`
	`  </b>`
	`</X>`
		`{`
		`  a: <AA>...</AA>,`
		`  b: <BB>..</BB>`
		`}`

4.2.2 Formalisierung mithilfe von JSON-Schema

Im Rahmen dieser Arbeit wird die Spezifikation der passiven Verwaltungsschale mithilfe von JSON-Schema (siehe Abschnitt 2.1) formalisiert. Dabei müssen durch die Formalisierung zwei wesentlich Aspekte abgedeckt werden: zum einen das Metamodell (siehe Abschnitt 3.1) und zum anderen die Constraints (siehe Abschnitt 3.2).

Der Einstiegspunkt ins Metamodell ist die Klasse `Environment` mit ihren drei Attributen `assetAdministrationShells`, `submodels` sowie `conceptDescriptions`. Die entsprechende Formalisierung dieser Klasse und somit die Wurzel des JSON-Schemas der Verwaltungsschale zeigt Listing 4.2. Zur verbesserten Les- und Wartbarkeit werden Referenzen (`$ref`) verwendet, um auf die Klassen der einzelnen Attribute zu verweisen. So verweist beispielsweise `#/$defs/Submodel` auf das JSON-Schema, das Listing 4.3 zeigt.

Die Vererbung der Klasse `Submodel` von den Klassen `Identifiable` etc. wird mithilfe von `$allOf` formalisiert. Die `submodelElements` erben gemäß Metamodell von der abstrakten Klasse `SubmodelElement`, Im JSON-Schema müssen alle konkreten Kindklassen der abstrakten Klasse in `anyOf` aufgezählt werden.

An dieser Stelle lassen sich einige Charakteristika der Formalisierung mithilfe von JSON-Schema beobachten: Es ist wichtig zu betonen, dass es sich um eine Formalisierung auf Ebene der *Serialisierung* handelt. So kennt JSON-Schema, so wie das JSON Format selbst, im eigentlichen Sinne keine Vererbung oder abstrakte Klassen, sondern lediglich Konzepte wie Vereinigung (`$allOf`) oder Auswahl (`$anyOf`) von Eigenschaften. Eine weitere Konsequenz der Formalisierung auf Serialisierungsebene sind Ausdrücke wie `modelType`, die

Listing 4.2: Formalisierung der Klasse `Environment`

```
type: object
properties:
  assetAdministrationShells:
    type: array
    items:
      $ref: "#/$defs/AssetAdministrationShell"
  submodels:
    type: array
    items:
      $ref: "#/$defs/Submodel"
  conceptDescriptions:
    type: array
    items:
      $ref: "#/$defs/ConceptDescription"
```

Listing 4.3: Formalisierung der Klasse `Submodel`

```
Submodel:
  allOf:
  - $ref: "#/$defs/Identifiable"
  - $ref: "#/$defs/HasKind"
  - $ref: "#/$defs/HasSemantics"
  - $ref: "#/$defs/Qualifiable"
  - $ref: "#/$defs/HasDataSpecification"
  - properties:
      submodelElements:
        type: array
        items:
          anyOf:
           - $ref: "#/$defs/Blob"
           - $ref: "#/$defs/File"
             ...
           - $ref: "#/$defs/SubmodelElementCollection"
           - $ref: "#/$defs/SubmodelElementList"
      modelType:
        const: Submodel
```

sich bei einer Formalisierung auf Ebene der *Klassenhierarchie* nicht finden würden. In Abschnitt 4.3 werden Vor- und Nachteile dieser Entscheidung diskutiert.

4.2.3 Formalisierung der Constraints

In Abschnitt 3.2 wurde eine systematische Einteilung der Constraints in vier Kategorien vorgenommen. Diese Kategorisierung wird für die Formalisierung der Constraints wieder aufgegriffen. Daher werden entsprechend der Kategorisierung vier verschiedene Möglichkeiten der Formalisierung unterschieden. **Wertebezogene Constraints** beschreiben Anforderungen an den Wert *eines* Attributes, also unabhängig von anderen Attributen. Bei dem Wert handelt es sich auf Ebene der Serialisierung bei der Verwaltungsschale immer um eine Zeichenkette. Dementsprechend lassen sich diese Constraints mithilfe von Bedingungen für Zeichenketten gut formalisieren (siehe Unterunterabschnitt 2.1.2). Als Beispiel sei hier Constraint AASd-002 [3, Seite 57] genannt:

> **Constraint AASd-002**: idShort of Referables shall only feature letters, digits, underscore ("_"); starting mandatory with a letter, i.e. [a-zA-Z][a-zA-Z0-9_]*.

In der Spezifikation ist direkt der reguläre Ausdruck vorgegeben, sodass sich eine Formalisierung mithilfe von `pattern` anbietet:

```
Constraint_AASd-002:
  type: string
  pattern: "^[a-zA-Z][a-zA-Z0-9_]*$"
```

Ein weiterer wertbezogener Constraint ist AASd-020 [3, Seite 54]:

> **Constraint AASd-020**: The value of Qualifier/value shall be consistent with the data type as defined in Qualifier/valueType.

Dieser lässt sich mithilfe der `format`-Bedingung formalisieren:

```
Constraint_AASd-020:
  anyOf:
  - properties:
      valueType:
        const: xs:date
      value:
        type: string
        format: xs:date
  - properties:
      valueType:
        const: xs:dateTime
      value:
        type: string
        format: xs:dateTime
  - properties:
      valueType:
```

```
        const: xs:base64Binary
      value:
        type: string
        format: xs:base64Binary
```

Zu beachten ist, dass für die `format` Bedingung nur wenige Formate vom JSON-Schema-Standard vorgegeben sind [32]. Die XSD-Datentypen, die die Verwaltungsschale nutzt, müssen daher separat implementiert werden.

Eine weitere Möglichkeit besteht bei **bedingten Constraints**. Bei diesen müssen manche Attribute existieren oder bestimmte Werte belegen, in Abhängigkeit von der Existenz oder Belegung anderer Attribute. Ein einfaches Beispiel ist AASd-005 [3, Seite 47]:

> **Constraint AASd-005:** If AdministrativeInformation/version is not specified, AdministrativeInformation/revision shall also be unspecified. This means that a revision requires a version. If there is no version, there is no revision. Revision is optional.

Dies lässt sich direkt mithilfe der Bedingung `dependentRequired` formalisieren:

```
Constraint_AASd-005:
  dependentRequired:
    revision:
    - version
```

Für die meisten bedingten Constraints sind verschachtelte `anyOf` und `allOf` Konstrukte nötig, wie z.B. für AASd-014 [3, Seite 70]:

> **Constraint AASd-014**: Either the attribute globalAssetId or specificAssetId of an Entity must be set if Entity/entityType is set to SSelfManagedEntity". Otherwise, they do not exist.

Dies lässt sich wie folgt formalisieren:

```
anyOf:
- properties:
    entityType:
      const: SelfManagedEntity
    specificAssetIds: false
  required:
    - globalAssetId
- properties:
    entityType:
      const: SelfManagedEntity
    globalAssetId: false
  required:
    - specificAssetIds
- properties:
    entityType:
      const: CoManagedEntity
    globalAssetId: false
    specificAssetIds: false
```

Es werden somit alle Möglichkeiten im `anyOf` ausformuliert. Im Gegensatz zu den vorherigen Constraints lassen sich **komplexe Constraints** nicht mittels JSON-Schema formalisieren. Ein Beispiel ist AASd-128 [3, Seite 93]:

> **Constraint AASd-128:** For model references, i.e. References with Reference/type = ModelReference, the Key/value of a Key preceded by a Key with Key/type=SubmodelElementList is an integer number denoting the position in the array of the submodel element list.

Dieser Constraint lässt sich nicht formalisieren, da JSON-Schema keinen Vergleich benachbarter Arrayelemente erlaubt. Diese Constraints müssen somit separat implementiert werden.

Abschließend gibt es noch **nicht prüfbare Constraints**. Diese lassen sich prinzipiell nicht maschinell überprüfen, somit auch nicht mithilfe von JSON-Schema. Ein Beispiel ist Constraint AASd-012 [3, Seite 72]:

> **Constraint AASd-012:** If both the MultiLanguageProperty/value and the MultiLanguageProperty/valueId are present, the meaning must be the same for each string in a specific language, as specified in MultiLanguageProperty/valueId.

Die Bedeutung einer Zeichenkette lässt sich nicht sicher in mehreren Sprachen überprüfen.

4.3 Evaluation

Zur Evaluation des Verfahrens bietet es sich an, zwischen flachen Serialisierungsformaten (JSON, XML) sowie Containerformaten (AASX) zu unterscheiden.

4.3.1 XML und JSON

Um die Korrektheit der Formalisierung und des Verfahrens zu überprüfen, wurde der Datensatz aus [66] verwendet. Bei diesem Datensatz handelt es sich um einen Satz an Testdateien im JSON- sowie im XML-Format für Version 3.0 der Spezifikation der Verwaltungsschale. Für jede Datei ist vorgegeben, ob sie serialisierte Verwaltungsschalen enthalten, die konform zur Spezifikation sind oder nicht. Bei Konformität muss das vorgestellte Verfahren die Datei akzeptieren, andernfalls abweisen. Die Ergebnisse zeigt Tabelle 4.2.

Tabelle 4.2: Ergebnisse der Auswertung

	JSON		XML	
	Konform	Nicht konform	Konform	Nicht konform
Akzeptiert	2556	0	2556	0
Abgewiesen	0	1950	0	1723

Es gibt in der Tat weder konforme, abgewiesene noch nicht-konforme, akzeptierte Dateien. Daraus folgt, dass das Verfahren in Bezug auf die genutzten Testdatensatz korrekt ist. Ferner wurde die Zeit gemessen, die bei der Überprüfung der Dateien vergangen ist[1]. Die Ergebnisse zeigt Abbildung 4.2. Zum Vergleich sind auch die Ausführungszeiten bei der Überprüfung mit einer direkt in Python implementierten Konformitätsprüfung erfasst. Hierfür wurde die Version 1.0 der AAS Test Engines verwendet [78].

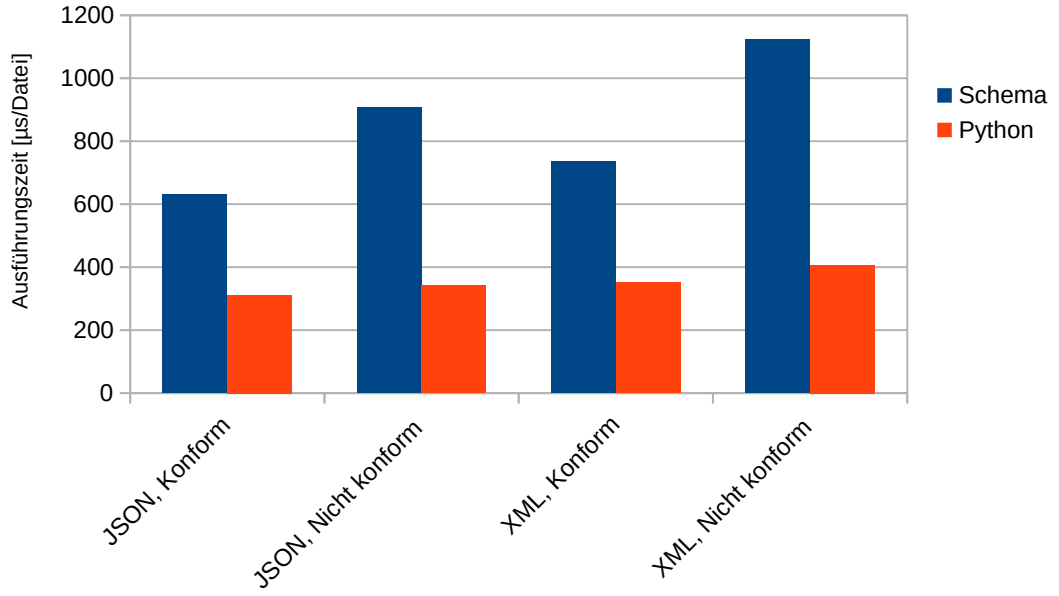


Abbildung 4.2: Es wurden verschiedene Instanzen von Verwaltungsschalen (valide und invalide) in verschiedenen Serialisierungen (JSON und XML) auf Konformität überprüft. Die Grafik vergleicht, wie lange die Überprüfung mithilfe der vorgestellten JSON-Schema-Überprüfung (blau) gegenüber einer direkten Implementierung des Metamodells in Python (rot) dauert.

Es lassen sich einige Schlussfolgerungen aus den Messungen ableiten: Zunächst lassen sich konforme Dateien schneller prüfen als nicht-konforme. Dies liegt in der Natur der genutzten Testdaten: So sind die konformen Dateien oft kleiner als die nicht-konformen, weil beispielsweise optionale Attribute in den konformen Dateien weggelassen wurden. Außerdem lassen sich JSON-Dateien schneller überprüfen als XML. Dies liegt in der Tatsache begründet, dass JSON ein leichtgewichtigeres Format ist. Außerdem müssen im Falle von JSON-Schema XML-Dateien zunächst nach JSON konvertiert werden. Zu guter Letzt ist die manuell implementierte Überprüfung in Python 2-3 mal schneller als die Überprüfung mittels JSON-Schema. Dies liegt daran, dass eine generische JSON-Schema-Implementierung nicht von Optimierungseffekten profitieren kann, die eine manuelle Implementierung ermöglicht. Dies lässt sich insbesondere bei der Überprüfung der Constraints beobachten.

4.3.2 AASX

Für das AASX-Format gibt es keinen Test-Datensatz. Eine quantitative Evaluation wie für JSON und XML ist daher nicht möglich. Stattdessen soll im folgenden auf eine beispielhafte

[1] Auf einem Rechner mit Intel Core i7-1165G7

Auswahl von Fehlern in AASX-Paketen zurückgegriffen werden. Die Fehler ergaben sich begleitend zur Entwicklung der AAS Test Engines [78], dem offiziellen Konformitätstesttool der IDTA. Die AAS Test Engines nutzen viele der in dieser Arbeit vorgestellten Verfahren.

Das Referenztool zum Erzeugen von AASX-Paketen ist der AASX Package Explorer [10]. Dieser erzeugte allerdings längere Zeit invalide AASX-Pakete [79]. Grund dafür war ein Fehler, der falsche Relationships im OPC Container anlegte. Dieser Fehler konnte nach entsprechender Nachbesserung mit dem vorgestellten Verfahren gefunden werden [80]. Da dies allerdings dazu führte, dass sehr viele AASX-Pakete (korrekterweise) abgewiesen wurden, wurde später beschlossen, dies nur als Warnung anstelle eines Fehlers zu markieren [81].

Auch andere Softwaretools erzeugten fehlerhafte AASX-Pakete. Einen häufig auftretenden Fehler bilden leere Arrays in JSON-Serialisierungen. Die Spezifikation schreibt vor, dass leere Arrays nicht erlaubt sind und stattdessen ganz weggelassen werden müssen. Diese Art Fehler konnte mit dem vorgestellten Verfahren erfolgreich gefunden werden und die Fehler in den erzeugenden Applikationen entsprechend behoben werden [82].

Eine beispielhafte Ausgabe bei der Überprüfung einer AASX-Datei mit einem Fehler zeigt Abbildung 4.4. In dieser Datei wurden die Constraints AASd-121 sowie AASd-123 verletzt. Den Hinweis darauf findet der Nutzer in den Zeilen `Reference #/$defs/Constraint_AASd-121 is invalid`. An der Größe der Ausgabe lässt sich ein Nachteil bei der Validierung mit JSON-Schema erkennen: Die Fehlermeldungen sind meist länglich und nicht auf die Eigenheiten der Verwaltungsschale bezogen. Abbildung 4.3 zeigt zum Vergleich die Ausgabe einer manuell implementierten Überprüfung in Python (mit den AAS Test Engines v1.0 [78]). Die Fehlermeldungen sind kürzer und klarer.

```
Checking AASX package ▼
    Checking [Content_Types].xml
    Checking relationships ▶
    Checking files ▼
        Checking aasx/the_aas.json ▼
            Check ▼
                Check meta model
                Check constraints ▼
                    Constraint AASd-121 violated: first key must be one of GloballyIdentifiables @
                    /submodels/0/submodel_elements/0/value
                    Constraint AASd-123 violated: first key must be one of AasIdentifiables @
                    /submodels/0/submodel_elements/0/value
```

Abbildung 4.3: Beispielausgabe für eine fehlerhafte Serialisierung (Überprüfung mit manueller Implementierung)

Abbildung 4.4: Beispielausgabe für eine fehlerhafte Serialisierung (Überprüfung mit JSON-Schema)

4.4 Teilmodelle in passiven Verwaltungsschalen

Die Spezifikationen der Verwaltungsschale definieren für passive Verwaltungsschalen neben Metamodell und Constraints auch Teilmodelle. Eine nähere Erläuterung zu Teilmodellen der Verwaltungsschale findet sich in Unterabschnitt 3.3.3. Im folgenden Abschnitt wird erläutert, inwieweit sich das vorgestellte Verfahren erweitern lässt, um Submodelle, die in passiven Verwaltungsschaleninstanzen enthalten sind, auf Konformität zu Teilmodellen zu überprüfen. Das beschriebene Verfahren konvertiert eine passive Verwaltungsschale zunächst nach JSON, falls es nicht bereits als JSON-Serialisierung vorliegt. Danach wird die Verwaltungsschale auf Konformität geprüft, indem sie gegen das JSON-Schema der Verwaltungsschale geprüft wird. Die Ausgabe des Verfahrens ist somit eine JSON-Serialisierung der Eingangsinstanz und die Aussage, ob diese Instanz konform zum Metamodell und zu den Constraints ist. Falls die Instanz nicht konform ist, dann erübrigt sich auch die Überprüfung auf Konformität der Submodelle zu Teilmodellen. Dementsprechend wird im Folgenden davon ausgegangen, dass eine JSON-Serialisierung der zu überprüfenden Verwaltungsschale vorliegt, die konform zum Metamodell und den Constraints ist. Ausgehend von dieser Eingabe erfolgt die Überprüfung wie in Abbildung 4.5 dargestellt.

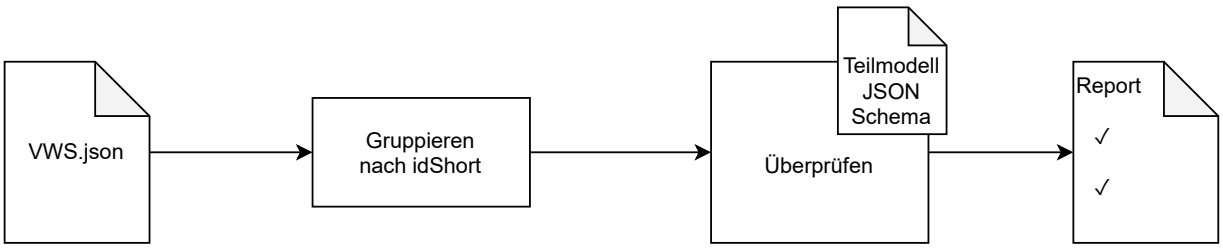


Abbildung 4.5: Überprüfung von Submodellen in einer passiven Verwaltungsschaleninstanz auf Konformität zu einem Teilmodell

Zunächst wird die Instanz transformiert, indem eine Gruppierungsoperation darauf angewandt wird. Dies dient im Wesentlichen dazu, `SubmodelElementCollection`s formalisieren zu können. Anschließend wird die transformierte Instanz gegen das JSON-Schema des Teilmodells überprüft und als Bericht ausgegeben. Soll die Instanz auf mehrere Teilmodellspezifikationen geprüft werden, kann die Überprüfung auch auf mehrere JSON-Schemata parallel angewandt werden.

4.4.1 Gruppierung

Viele Teilmodelle nutzen `SubmodelElementCollection`s [6, 7]. Diese repräsentieren die Datenstruktur einer Menge, die in JSON nicht zur Verfügung steht. In den Serialisierungen der Verwaltungsschale wird stattdessen eine Liste genutzt und die eigentlichen Unterelemente dann über deren `idShort` identifiziert. Dies würde aber die Formalisierung mittels JSON-Schema erschweren. Daher werden die Elemente in der Liste stattdessen nach ihrer `idShort` gruppiert und dann im JSON-Schema als Objekt formalisiert. Dies ist deutlich einfacher und performanter.

Der Algorithmus zum Gruppieren ist in Listing 4.4 dargestellt: Zunächst wird über alle Einträge in der Liste iteriert. Für jeden Eintrag wird die `idShort` extrahiert. Existiert die

`idShort` noch nicht im Ergebnis, wird das Attribut hinzugefügt mit dem Eintrag als Wert. Andernfalls wird der Eintrag an den Attributwert angehängt. Tabelle 4.3 zeigt ein Beispiel.

Listing 4.4: Gruppierung nach idShort

```
def group_by_id_short(array):
  result = {}
  for entry in array:
    id_short = entry["idShort"]
    if id_short not in result:
      result[id_short] = [entry]
    else:
      result[id_short].append(entry)
  return result
```

Tabelle 4.3: Beispiel zur Gruppierung nach `idShort`

Vorher	Nachher

`- idShort: Company`
`  modelType: Property`
`  value: ABC Company`
`- idShort: Language`
`  modelType: Property`
`  value: de`
`- idShort: Language`
`  modelType: Property`
`  value: en`
	`Company:`
	`- modelType: Property`
	`  value: ABC Company`
	`Language:`
	`- modelType: Property`
	`  value: de`
	`- modelType: Property`
	`  value: en`

4.4.2 Formalisierung der Teilmodelle

Die Formalisierung wird nachfolgend an den Teilmodellen *Contact Information* sowie *Digital Nameplate for Industrial Equipment* illustriert (siehe Unterabschnitt 3.3.3). Diese Teilmodelle dienen lediglich als Beispiel. Das Verfahren ist prinzipiell auf alle Teilmodellespezifikationen anwendbar.

Contact Information

Das Teilmodell *Contact Information* [6] dient dem Ablegen von Kontaktinformationen in einer Verwaltungsschale. Die Formalisierung mittels JSON-Schema ist in Listing 4.5 dargestellt. Dafür werden zunächst die Submodels der Verwaltungsschale betrachtet (Zeile 3). Diese Liste soll mindestens ein Submodel mit der `idShort ContactInformations` enthalten (Zeile 6, 7). Für dieses Submodel sollen die Submodellelemente von Typ `ContactInformationEntry` sein (Zeile 10). Jedes dieser Elemente soll wiederum eine `SubmodelElementCollection` sein (Zeile 12), wobei die `idShort ContactInformation` von der Spezifikation vorgegeben ist.

Die einzelnen Elemente der Collection werden dann nach `idShort` gruppiert (Zeile 17). Dadurch ergibt sich ein Attribut `RoleOfContactInformation`, das eine Liste von `Properties` gemäß Spezifikation sein soll (Zeile 21). Diese sollen den Wertetyp `xs:string` haben (Zeile 24). Der eigentliche Wert darf nur einen der für die Rolle vorgegebenen Werte annehmen (Zeile 26-31).

Für das Attribut `NationalCode` muss eine `MultiLanguageProperty` gemäß Spezifikation gewählt werden (Zeile 35). Über `maxItems` wird die Kardinalität aus der Spezifikation formalisiert (Zeile 36). In ähnlicher Weise können auch die übrigen Merkmale des Teilmodells *Contact Information* formalisiert werden.

Listing 4.5: JSON-Schema des Teilmodells Contact Information (gekürzt, aus [15])

```
1  ContactInformation:
2    properties:
3      submodels:
4        contains:
5          properties:
6            idShort:
7              const: ContactInformations
8            submodelElements:
9              items:
10               $ref: '#/$defs/ContactInformationEntry'
11 ContactInformationEntry:
12   $ref: '#/$defs/SubmodelElementCollection'
13   properties:
14     idShort:
15       const: ContactInformation
16     value:
17       groupBy: idShort
18       properties:
19         RoleOfContactPerson:
20           items:
21             $ref: '#/$defs/Property'
22             properties:
23               valueType:
24                 const: xs:string
25               value:
26                 enum:
27                   - 0173-1#07-AAS927#001
28                   - 0173-1#07-AAS928#001
29                   - 0173-1#07-AAS929#001
30                   - 0173-1#07-AAS930#001
31                   - 0173-1#07-AAS931#001
32           maxItems: 1
33         NationalCode:
34           items:
35             $ref: '#/$defs/MultiLanguageProperty'
36           maxItems: 1
37 ...
```

Digital Nameplate

Das Teilmodell *Digital Nameplate for Industrial Equipment* [7] dient dem Ablegen von digitalen Typenschildern in einer Verwaltungsschale. Das Schema des Teilmodells *Digital Nameplate* zeigt Listing 4.6. Wie beim Teilmodell *Contact Information* werden auch hier zunächst die Submodelle betrachtet, wobei hier nach einem Submodell mit der `idShort DigitalNameplate` gesucht wird (Zeile 7). Die Submodellelemente werden nach `idShort` gruppiert (Zeile 9). Zunächst wird formalisiert, dass mindestens eines der Merkmale `ManufacturerProductFamily` oder `ManufacturerProductType` vorhanden sein soll (Zeile 11-14). Für `URIOfTheProduct` muss gemäß Spezifikation eine `Property` mit dem Datentyp `xs:string` vorhanden sein (Zeile 16-21). Das Merkmal `ManufacturerName` soll eine `MultiLanguageProperty` sein (Zeile 24). Ferner verweist das Merkmal `ContactInformation` auf das Schema des `ContactInformationEntry` aus der Formalisierung des Teilmodells *Contact Information*. Dadurch kann die Komposition von Teilmodellen formalisiert werden. Durch `required` werden die Kardinalitäten aus der Spezifikation formalisiert (Zeile 28-30).

Listing 4.6: JSON-Schema des Teilmodells Digital Nameplate (gekürzt, aus [15])

```
1  DigitalNameplate:
2    properties:
3      submodels:
4        contains:
5          properties:
6            idShort:
7              const: DigitalNameplate
8            submodelElements:
9              groupBy: idShort
10             anyOf:
11             - required:
12               - ManufacturerProductFamily
13             - required:
14               - ManufacturerProductType
15             properties:
16               URIOfTheProduct:
17                 items:
18                   $ref: '#/$defs/Property'
19                   properties:
20                     valueType:
21                       const: xs:string
22               ManufacturerName:
23                 items:
24                   $ref: '#/$defs/MultiLanguageProperty'
25               ContactInformation:
26                 items:
27                   $ref: '#/$defs/Internal/ContactInformationEntry'
28             required:
29             - URIOfTheProduct
30             - ManufacturerName
31 ...
```

4.5 Fazit

In diesem Kapitel wurde ein Verfahren vorgestellt, das passive Verwaltungsschalen auf Konformität überprüft. Hierfür wurden das Metamodell und die Constraints mithilfe von JSON-Schema formalisiert.

FF2: Welche Besonderheiten zeichnen die Verwaltungsschale aus softwaretechnischer Sicht aus? Welche speziellen Anforderungen an ein System zum Überprüfen der Konformität der bzgl. Interoperabilität relevanten Teile des Verwaltungsschalen-Ökosystems ergeben sich daraus?

Zunächst können passive Verwaltungsschalen in verschiedenen Formaten serialisiert werden, die inhaltlich äquivalent sind. Dies hat zur Folge, dass unter Umständen eine Validierung für jedes Format separat implementiert werden muss. Um diesen Aufwand zu vermeiden, wird im vorgestellten Verfahren die Validierung ausschließlich für JSON implementiert. Alle anderen Formate werden lediglich nach JSON konvertiert und im Anschluss validiert.

Eine weitere Besonderheit besteht in der Komplexität des Metamodells und der Constraints. Zur Formalisierung des Metamodells mittels JSON-Schema sind diverse `allOf` und `anyOf` Konstrukte notwendig, um Vererbung und Instanziierung von Kindklassen korrekt abzubilden. Weiterhin ist das Metamodell rekursiv, was die Verwendung von `$ref`-Referenzen erfordert. Abschließend müssen auch die Constraints formalisiert werden. Zum Großteil können diese mithilfe von JSON-Schema direkt abgebildet werden. Andere, komplexere Constraints erfordern eine separate Implementierung.

FF3: Wie kann ein Testsystem für Softwarekomponenten der Verwaltungsschale umgesetzt werden?

Der vorgestellte Ansatz setzt auf die Formalisierung der Spezifikation mithilfe von JSON-Schema. Dadurch können JSON-Serialisierungen von Verwaltungsschalen direkt validiert werden. XML-Serialisierungen werden zunächst nach JSON konvertiert und dann validiert. AASX-Container werden zunächst entpackt und dann die enthaltenen Serialisierungen einzeln validiert. Es zeigt sich, dass die Formalisierung mithilfe von JSON-Schema einige Vorteile hat. Insbesondere können problemlos mehrere Versionen der Verwaltungsschale parallel gepflegt werden, da für jede Version lediglich ein eigenes Schema konfiguriert werden muss. Ein Nachteil von JSON-Schema sind die teilweise länglichen Fehlermeldungen, falls eine Serialisierung nicht konform zur Spezifikation ist.

FF4: Welche Fehler können mit dem Testsystem in realen Softwarekomponenten für die Verwaltungsschale aufgedeckt werden? Welche Grenzen gibt es?

Der vorgestellte Ansatz kann jegliche Abweichung vom Metamodell erkennen. Dies um-

fasst beispielsweise fehlende Attribute oder fehlerhaft instanziierte Kindklassen. Da die Formalisierung auf Ebene der Serialisierung (JSON etc.) und nicht auf Ebene der Klassenhierarchie erfolgt, können Fehler in der Serialisierung besser erkannt werden. Dies umfasst beispielsweise das Attribut `modelType`, das bei Ansätzen auf Ebene der Klassenhierarchie ggf. nicht überprüft wird [83]. Darüber hinaus erkennt das Verfahren, wenn Constraints nicht eingehalten werden. Einige Constraints können allerdings generell nicht geprüft werden, was auch mit anderen Verfahren nicht möglich wäre.

FF5: Welche Vor- und Nachteile hat die automatische Ableitung von Konformitätstests aus formalen Spezifikationen?

Bei der Überprüfung von passiven Verwaltungsschalen ermöglicht die Nutzung des bereitgestellten JSON-Schemas eine sehr schnelle Implementierung eines Validierungstools. Allerdings müssen die Constraints separat formalisiert werden. Ferner kann das JSON-Schema nur auf die JSON-Serialisierung der Verwaltungsschale angewandt werden. Für XML- und AASX-Serialisierungen sind zusätzliche Konvertierungen nötig. Ein Nachteil bei der Nutzung von JSON-Schema sind teilweise schwer verständliche Fehlermeldungen, insbesondere im Rahmen von Constraints. Bei großen Instanzen kann zudem die Performance der Validierung zu gering sein und zwei- bis dreimal länger dauern als eine direkte Implementierung der Validierung.

5 Sicherstellung der Konformität von Verwaltungsschalen-SDKs

Software Development Kits (SDKs) bilden die Grundlage für viele Softwarekomponenten im Ökosystem der Verwaltungsschale. Sie bieten diverse Grundfunktionen zum Umgang mit passiven Verwaltungsschalen, insbesondere dem (De-)Serialisieren und Überprüfen der Constraints. In diesem Kapitel wird zunächst ein Aufbau zum Testen von SDKs vorgestellt. Ein wesentlicher Bestandteil dieses Aufbaus bildet das systematische Generieren von Testinstanzen von Verwaltungsschalen. Das Verfahren zum Generieren wird im zweiten Teil dieses Kapitels erläutert. Abschließend wird das Verfahren an einigen SDKs evaluiert.

5.1 Verwandte Arbeiten

5.1.1 Verfahren zum Testen von Verwaltungsschalen-SDKs

Der Aufbau zum Testen von SDKs wurde zuerst in [12] vorgestellt. Hierfür werden Testinstanzen der Verwaltungsschale erzeugt und an das zu testende SDK übergeben. Das SDK deserialisiert die Dateien, überprüft sie und serialisiert sie wieder. Wenn dies für alle Instanzen korrekt funktioniert, wird das SDK als konform angesehen. Ein vergleichbarer Testaufbau wird auch im Rahmen dieser Arbeit verwendet. Darüber hinaus skizzieren die Autoren ein Verfahren zum systematischen Erzeugen von Verwaltungsschalen. Dieses Verfahren nutzt Property-based Testing [22] und eine Formalisierung des Metamodells mithilfe eines Subsets der Python-Programmiersprache [67]. Hierbei werden zunächst valide Verwaltungsschalen erzeugt und im Anschluss gezielt Defekte eingebracht, wie z.B. das Verletzen von Constraints.

Im Gegensatz dazu, wurde in einer vorbereitenden Arbeit zu dieser Dissertation [23] eine Formalisierung des Metamodells durch JSON-Schema genutzt. Die Autoren zeigen das Problem der Formalisierung der Constraints auf. Dieses Problem wird in [13] adressiert, um auch Constraints mithilfe von JSON-Schema zu formalisieren. Das vorgestellte Verfahren wird auch im Rahmen dieser Arbeit verwendet.

In [84] erzeugen die Autoren Instanzen von Verwaltungsschalen, indem ein bestehendes SDK automatisch analysiert wird. Darauf aufbauend werden automatisch Beispielinstanzen erzeugt. Mit diesem Verfahren sind die Autoren in der Lage, Beispiele für nahezu alle Klassen des Metamodells der Verwaltungsschale zu erzeugen. Das Verfahren schlägt fehl bei den Klassen `DataSpecificationIEC61360` sowie `EmbeddedDataSpecification` aufgrund der hohen Anzahl an Constraints.

Eine Evaluation von SDKs wurde in [66] durchgeführt. Die Autoren erzeugen hierfür Testfälle, wie in [12] beschrieben. Mithilfe der Testfälle testen die Autoren im Anschluss diverse SDKs auf Konformität. Die besten Ergebnisse erzielen dabei die SDKs aus dem AAS Core Projekt. Obwohl die Bewertung noch auf Basis von Version 3.0RC02 der Spezifikation

der Verwaltungsschale durchgeführt wurden, decken sie sich im Wesentlichen mit denen aus Abschnitt 5.5. Auch in [5] wird eine Evaluation verschiedener SDKs durchgeführt. Die Autoren nutzen das in [23] für Version 3.0RC01 vorgestellte Verfahren zur Erzeugung von Testfällen für Version 3.0 der Verwaltungsschale. Die Ergebnisse decken sich mit denen in Abschnitt 5.5, was zu erwarten ist, weil im Rahmen dieser Arbeit ein ähnliches Verfahren zur Erzeugung von Testfällen verwendet wurde.

5.1.2 Generische Verfahren zur Erzeugung von Testfällen

In Unterabschnitt 2.3.1 wurden verschiedene Methoden zum Erzeugen von Testfällen vorgestellt. In diesem Abschnitt werden diese Methoden konkret im Kontext der Erzeugung von Testdaten betrachtet, die in Testfällen verwendet werden. Hervorzuheben ist, dass es generell abzuwägen gilt, wie viel Aufwand in die Vorbereitung und Konfiguration der Generierung investiert wird: Mit steigendem Aufwand steigt im Allgemeinen auch die Qualität der Testfälle. Die nachfolgenden Abschnitte sind (grob) nach dem nötigen manuellem Aufwand sortiert.

Zufallsbasierte Verfahren

Zufallsbasierte Verfahren erzeugen Testdaten zufällig [42]. Ein großer Vorteil dieser Verfahren ist, dass nur wenig Konfiguration der Generierung und Wissen über das SUT nötig sind. Ein Tool zum zufallsbasierten Erzeugen von Testdaten ist QuickFuzz [85]. QuickFuzz erzeugt Testdaten in diversen Formaten, u.a. JSON und XML. In [86] erzeugen die Autoren zufällige Testdaten, um REST APIs zu überprüfen. Das zufällige Erzeugen von JSON Daten basierend auf JSON-Schema wird in [87] beschrieben. Die Autoren verwenden einen ähnlichen Ansatz wie in dieser Arbeit. Rekursive Schemata, die systematische Abdeckung sowie das Formalisieren komplexer Constraints werden allerdings nicht im Rahmen von [87] betrachtet.

Suchbasierte Verfahren

Suchbasierte Verfahren [45, 46] erzeugen Testdaten iterativ. Bei jeder Iteration werden bestehenden Testdaten mutiert oder neue erzeugt, wobei eine gegebene Zielfunktion maximiert werden soll. In [88] erzeugen die Autoren XML-Dateien, um Webservices zu überprüfen und Schwachstellen aufzudecken. Sie evaluieren dabei verschiedene Zielfunktionen und testen diese in verschiedenen Kontexten.

Modellbasierte Verfahren

Beim modellbasierten Testen [50, 51] werden Testdaten anhand eines Systemmodells abgeleitet. In [89] erzeugen die Autoren XML-Testdaten mithilfe einer eigenen Sprache zum Beschreiben der Grammatik. Die Autoren von [90] beschreiben das Tool *Datagen*, Dieses verwendet ebenfalls eine eigene Grammatiksprache zum Erzeugen von JSON- und XML-Daten. Eine Unterform des modellbasierten Testen ist Property-based-Testing [52], bei dem Testdaten anhand eines Datenmodells abgeleitet werden. Ein verbreitetes Tool ist Hypothesis [53], das Property-based-Testing für die Programmiersprache Python implementiert. Hypothesis wird im Rahmen dieser Arbeit als Vergleichstool für die Evaluation verwendet.

5.2 Testaufbau

Ziel des vorgestellten Verfahrens ist das Überprüfen von SDKs auf Konformität zum Standard der Verwaltungsschale [3]. Im Kern stehen dabei die Funktionen zum Serialisieren und Deserialisieren von Dateien in den Formaten JSON, XML und AASX. Daraus resultiert der in Abbildung 5.1 dargestellte Testaufbau, wie in [12] erstmals vorgestellt. Das Verfahren verwendet das mithilfe von JSON-Schema formalisierte Metamodell als Eingabe. Dies ist idealerweise dasselbe Schema, welches bereits im vorherigen Kapitel 4 zur *Überprüfung* von passiven Verwaltungsschalen verwendet wurde. Basierend auf dem Schema werden systematisch Verwaltungsschalen erzeugt. Systematisch bedeutet hierbei, dass alle Aspekte des Metamodells sowie alle Serialisierungsformate abgedeckt werden.

Im Anschluss sendet ein Testadapter jede Testdatei an ein SDK. Dieses soll die Verwaltungsschale deserialisieren, überprüfen und, wenn fehlerfrei, wieder serialisieren. Dieser Testadapter muss für jedes SDK separat implementiert werden. Abschließend erfolgt die Auswertung. Hierbei müssen zwei Fälle unterschieden werden: Im ersten Fall ist die Testverwaltungsschale korrekt, genügt also dem Metamodell und den Constraints (positiver Test). Dann muss das SDK die Verwaltungsschale akzeptieren und wieder serialisieren. Die serialisierte Verwaltungsschale muss dann zusätzlich mit der Eingangsverwaltungsschale übereinstimmen. Im zweiten Fall ist die Testverwaltungsschale inkorrekt, genügt also bewusst *nicht* dem Metamodell oder einem der Constraints. Dann muss das SDK diese abweisen (negativer Test).

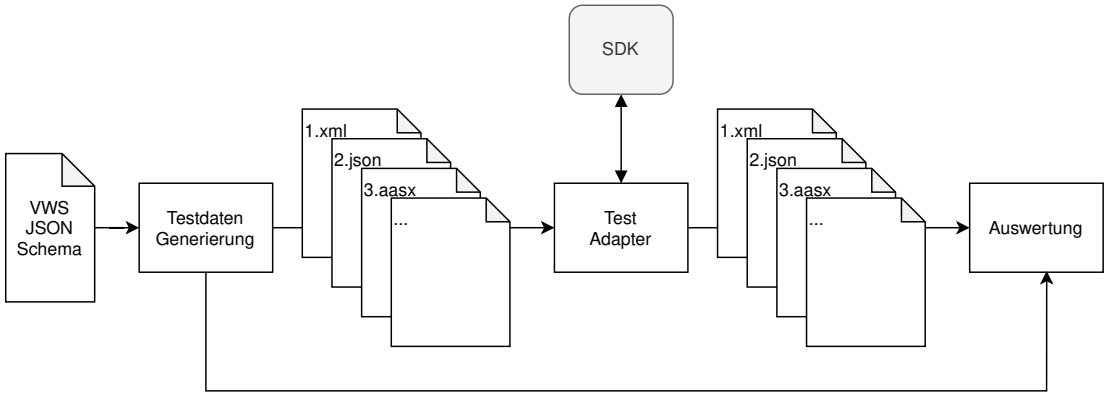


Abbildung 5.1: Aufbau zum Testen eines Verwaltungsschalen-SDKs (nach [5, 12])

Nur wenn das SDK alle korrekten Verwaltungsschalen akzeptiert und alle inkorrekten ablehnt, wird es als konform zur Spezifikation angesehen. Zu beachten ist, dass dies kein Beweis für die Konformität des SDKs ist, sondern lediglich ein sehr starker Indikator dafür. Dies liegt daran, dass die Abwesenheit von Fehlern durch Testen nicht nachgewiesen werden kann. Ferner kann, anders als in Abbildung 5.1 dargestellt, die Generierung der Testdaten separat von der eigentlichen Testausführung erfolgen. Dies kann z.B. sinnvoll sein, um mehrere SDKs effizient zu überprüfen oder die Testdaten als Teil eines Releases der Spezifikation zu veröffentlichen.

5.3 Generierung von Testdaten

Die Aussagekraft des obigen Testaufbaus hängt maßgeblich von den generierten Test-Verwaltungsschalen ab. Diese müssen ausreichend viele Aspekte des Metamodells abdecken, um die Konformität überprüfen zu können. Hierfür wird im folgenden ein Verfahren vorgestellt, das basierend auf dem JSON-Schema der Verwaltungsschale Testinstanzen erzeugt.

Die Grundidee hierbei ist, dass bei der Instanziierung von Verwaltungsschalen gemäß Metamodell an diversen Stellen Entscheidungen getroffen werden müssen. Beispiele hierfür sind die Wahl von konkreten Werten für Attribute oder die Wahl eines konkreten Submodel-Elements. Diese Entscheidungen werden mithilfe eines Flussgraphen modelliert. In diesem steht jeder Knoten für eine bestimmte Entscheidung. Wenn man einem Pfad durch den Flussgraph folgt, also eine bestimme Serie von Entscheidungen trifft, dann erhält man somit genau eine bestimmte Verwaltungsschaleninstanz. Wird diese Auswahl solange wiederholt, bis man alle Knoten abgedeckt hat, dann ist sichergestellt, dass alle Möglichkeiten zum Treffen von Entscheidungen abgedeckt wurden. Dadurch ist gleichzeitig sichergestellt, dass alle Freiheitsgrade des Metamodells in den Testdaten abgedeckt sind. Dieses Verfahren garantiert daher, dass der Test anhand der generierten Testinstanzen der Verwaltungsschale bzgl. der möglichen Varianten von Verwaltungsschalen vollständig ist.

Den Ablauf dieses Verfahrens zeigt Abbildung 5.2. Es basiert maßgeblich auf dem in [13] beschriebenen Algorithmus. Zunächst wird das JSON-Schema der Verwaltungsschale vereinfacht, normalisiert und Referenzen isoliert, um ein normalisiertes Schema zu erhalten. Das normalisierte Schema ist äquivalent zum Eingangsschema. Äquivalent bedeutet, dass die beiden Schemata jedes potenzielle JSON-Dokument entweder gleichermaßen akzeptieren oder ablehnen.

Anschließend wird aus dem Schema ein Flussgraph konstruiert und erweitert um Konstrukte, die speziell für die Verwaltungsschale nötig sind. Dies dient dazu, die möglichen Entscheidungen zu modellieren, die bei der Instanziierung auftreten können. Abschließend wählt das Verfahren Pfade aus dem Flussgraph und führt sie aus, um die Testdaten zu erhalten. Die einzelnen Schritte werden im Folgenden im Detail erläutert[1].

5.3.1 Schema vereinfachen

Zur Vereinfachung des Schemas werden einige Schlüsselworte durch andere ersetzt. Einige dieser Ersetzungen zeigt Tabelle 5.1. Diese Vorverarbeitung vereinfacht nachfolgende Schritte, weil die Anzahl an Schlüsselwörtern reduziert wird. Z.B. wird das Schlüsselwort `const` entfernt, indem es durch das äquivalente Schlüsselwort `enum` ersetzt wird. Ferner ist der Typ `integer` im übergeordneten Datentyp `number` enthalten und wird entsprechend ersetzt. `dependentRequired` kann ersetzt werden, indem die möglichen Belegungen der einzelnen Attribute explizit ausformuliert werden. Auch die boolschen Schemata `true` und `false` können durch die entsprechenden Schemata ersetzt werden.

[1] Die folgenden Erläuterungen sind Übersetzungen von [13] mit einigen vertiefenden Anmerkungen.

Tabelle 5.1: Ersetzungen zur Vereinfachung eines JSON-Schemas

Vorher	Nachher	Anmerkung
`const: value`	`enum:` `- value`	Entfernt das Schlüsselwort `const`.
`type: integer`	`type: number` `multipleOf: 1`	Entfernt den Typ `integer`.

`dependentRequired:`
`  a: ["b"]`
	`anyOf:`
	`- properties:`
	`  a: False`
	`  b: True`
	`- required:`
	`  - a`
	`  - b`
	`  properties:`
	`  a: True`
	`  b: True`
		Entfernt `dependentRequired`.

True	{}	Ersetzt das triviale Schema.
False	`enum: []`	Ersetzt das triviale Schema.

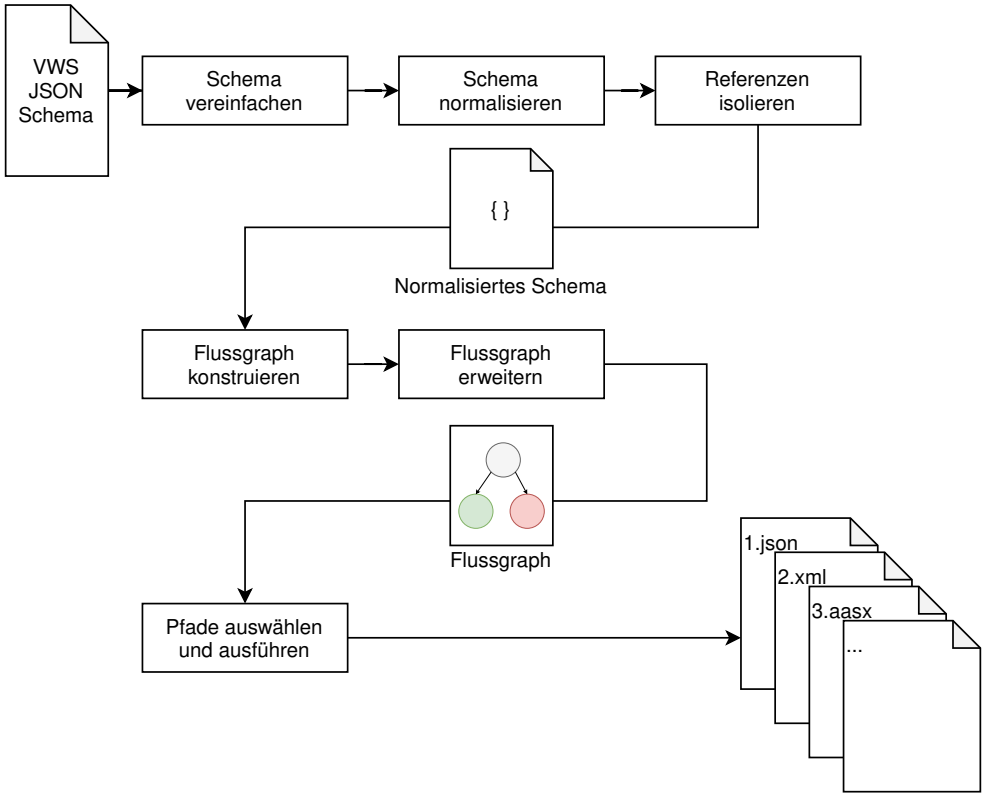


Abbildung 5.2: Erzeugen von Testdaten (nach [13])

5.3.2 Schema normalisieren

Beim Erzeugen von Beispielen für ein gegebenes JSON-Schema stellt man fest, dass das Erzeugen von korrekten Beispielen deutlich schwieriger ist, als das Erzeugen von inkorrekten Beispielen. Das liegt daran, dass der Generator für das Erzeugen von inkorrekten Beispielen lediglich einzelne Schlüsselworte des Schemas betrachten muss. Dahingegen erfordert das Erzeugen von korrekten Beispielen, dass alle Schlüsselwörter betrachtet werden müssen. Ein Beispiel soll diese Problematik verdeutlichen. Gegeben sei folgendes Schema:

```
allOf:
- type: integer
  minimum: 10
- anyOf:
  - maximum: 20
  - multipleOf: 5
```

Um invalide Daten für dieses Schema zu erzeugen reicht es beispielsweise `type: integer` zu betrachten. Es ergibt sich direkt ein beliebiger String wie `*yz"` als inkorrektes Sample. Um allerdings ein valides Sample zu erzeugen, müssen alle Schlüsselwörter betrachtet werden. Um dies zu ermöglichen, wird die Normalisierung des JSON-Schemas durchgeführt. Das wesentliche Ziel ist es, alle Schlüsselwörter in separaten Subschemata zusammenzuführen. Dadurch können diese im Verbund betrachtet werden. Die normalisierte Darstellung des obigen Schemas ist beispielsweise folgende:

```
anyOf:
- type: number
```

```
  mimimum: 10
  maximum: 20
- type: number
  minimum: 10
  multipleOf: 5
```

Es sind alle Schlüsselwörter in separaten Subschemata gesammelt, sodass man für das erste Subschema beispielsweise die Zahl 13 als korrektes Beispiel findet, für das zweite Subschema die Zahl 15. Das Ziel der Normalisierung ist somit das Entfernen aller logischen Applikatoren (vergl. Tabelle 2.1) bis auf `anyOf`.

Boolesche Algebra

Für die folgenden Abschnitte wird eine Abbildung eines JSON-Schemas als Boolesche Gleichung definiert. Dadurch kann boolesche Algebra auf JSON-Schemata angewendet werden. Schemata werden im Folgenden mit einem Großbuchstaben bezeichnet. Gemäß Abschnitt 2.1 besteht ein Schema aus mehreren Bedingungen. Jeder Bedingung wird eine boolesche Variable zugeordnet, im Folgenden mit Kleinbuchstaben bezeichnet. Diese boolesche Variable wird genau dann `wahr`, wenn die zugehörige Bedingung das zu prüfende JSON-Dokument akzeptiert. Ferner wird jedem logischen Applikator gemäß Tabelle 2.1 eine boolesche Verknüpfung zugeordnet. Mit diesen Regeln kann jedes Schema als boolesche Gleichung dargestellt werden. Als Beispiel sei folgendes Schema gegeben:

```
maximum: 100
anyOf:
- minimjm: 10
- multipleOf: 2
```

Dieses Schema S wird auf die Gleichung $S = s_1 \wedge (s_2 \vee s_3)$ abgebildet mit

$$s_1 \mapsto \texttt{maximum:100}$$
$$s_2 \mapsto \texttt{minimum:10}$$
$$s_3 \mapsto \texttt{multipleOf:2}$$

Definition 1. *Ein triviales Schema $S \mapsto \{s_1, ..., s_n\}$ ist ein Schema, das keine logischen Applikatoren enthält. Es wird abgebildet auf die boolesche Gleichung $S = \bigwedge_{i=1}^{n} s_i$.*

Zum Beispiel sind die Schemata {`"minimum":12, "maximum":24`} und {`"$ref":"#/"`} trivial. Das Schema {`"allOf":[]`} ist nicht trivial, da es den logischen Applikator `allOf` enthält.

Definition 2. *Ein normalisiertes Schema besteht aus genau einem Schlüsselwort, dem* ***`anyOf`*** *Schlüsselwort. Zusätzlich sind alle Einträge im Array* ***`anyOf`*** *triviale Schemata. Ein normalisiertes Schema $S \mapsto \{"anyOf" : [A_1, .., A_n]\}$ wird abgebildet auf die boolesche Gleichung $S = \bigvee_{i=1}^{n} A_i$, wobei die Subschemata A_i trivial sind.*

Damit kann die normalisierte Form als das JSON-Schema-Äquivalent der disjunktiven Normalform von booleschen Gleichungen interpretiert werden. Zum Beispiel ist folgendes Schema normalisiert:

```
anyOf:
- $ref: #/
- minimum: 10
  maximum: 100
```

Basierend auf dieser Zuordnung wird im Folgenden beschrieben, wie jedes (beliebige) Schema in seine normalisierte Form (nach Definition 2) gebracht werden kann. Dafür werden schrittweise alle logische Applikatoren aus dem Schema ersetzt, bis auf den logischen Applikator `anyOf`.

Ersetzen von Implikationen

Zuerst wird die Schlüsselwortkombination $S \mapsto$ {"if": A, "then": B, "else": C} ersetzt. Dafür wird boolesche Algebra angewandt:

$$\begin{aligned} S =& (A \implies B) \land (\overline{A} \implies C) \\ =& (A \land B) \lor (\overline{A} \land C) \end{aligned}$$

Somit kann das Schema wie folgt geschrieben werden:

```
anyOf:
- allOf:
  - A
  - B
- allOf:
  - not: A
  - C
```

Ersetzen von oneOf

Als nächstes wird das `oneOf` Schlüsselwort aus einem Schema $S \mapsto$ {"oneOf": [A, B]} ersetzt. Wieder kann boolesche Algebra angewandt werden:

$$\begin{aligned} S &= A \oplus B \\ &= (A \land \overline{B}) \lor (\overline{A} \land B) \end{aligned}$$

Somit kann S geschrieben werden als:

```
anyOf:
- allOf:
  - A
  - not: B
- allOf:
  - not: B
  - A
```

Diese Regel kann auch auf Schemata mit mehr als zwei `oneOf`-Einträgen angewandt werden.

Ersetzen von allOf

Gegeben Definition 1 wird eine Funktion `merge*`(A, B) definiert, die zwei triviale Schemata $A = \bigwedge a_i$ und $B = \bigwedge b_i$ zusammenführt.

$$\begin{aligned}\text{merge}^*(A, B) &= A \wedge B \\ &= \left(\bigwedge_i a_i\right) \wedge \left(\bigwedge_i b_i\right) \\ &= \bigwedge_i (a_i \wedge b_i)\end{aligned}$$

Das bedeutet, dass `merge*` implementiert werden kann, indem paarweise identische Schlüsselwörter zusammengeführt werden, wie in Tabelle 5.2 illustriert.

Tabelle 5.2: Beispiele für paarweises Zusammenführen von Schlüsselwörtern

Schlüsselwort a	Schlüsselwort b	$a \wedge b$
`"minimum":a`	`"minimum":b`	`"minimum":max(a,b)`
`"multipleOf":a`	`"multipleOf":b`	`"multipleOf":lcm(a,b)`
`"properties":{"x":A}`	`"properties":{"x":B}`	`"properties":{"x":merge(A,B)}`

Basierend auf Definition 2 kann nun eine Funktion `merge`(S, T) definiert werden, die *normalisierte* Schemata $S = \bigvee_i A_i$ and $T = \bigvee_j B_j$ zusammenführt:

$$\begin{aligned}\text{merge}(S, T) &= S \wedge T \\ &= \bigvee_i A_i \wedge \bigvee_j B_j \\ &= \bigvee_i \bigvee_j (A_i \wedge B_j) \\ &= \bigvee_i \bigvee_j \text{merge}^*(A_i, B_j) \\ &= U\end{aligned}$$

Das Ergebnis U entspricht somit folgendem Schema:

```
anyOf:
- merge*(A1, B1)
- merge*(A1, B2)
- ...
- merge*(A2, B1)
- ...
```

Das Verfahren soll anhand des folgenden Beispiels illustriert werden. Gegeben ist das normalisierte Schema S:

```
anyOf:
- minimum: 10
- type: ["string", "number"]
```

sowie das normalisierte Schema T:

```
anyOf:
- minimum: 20
```

Dann liefert merge(S, T) $\mapsto$ `{"allOf":[S, T]}`

```
anyOf:
- mimimum: 20
- type: ["string","number"]
  minimum: 20
```

Somit kann mithilfe von `merge` das `allOf` Schlüsselwort in jedem Schema ersetzt werden.

Ersetzen von not

Basierend auf Definition 1 kann eine Funktion `invert*`(A) definiert werden, die ein triviales Schema $A = \bigwedge a_i$ invertiert:

$$\begin{aligned} \text{invert}^*(A) = \overline{A} &= \overline{\bigwedge_i a_i} \\ &= \bigvee_i \overline{a_i} \end{aligned}$$

Das bedeutet, dass `invert*` implementiert werden kann, indem jedes Schlüsselwort einzeln invertiert wird und die Ergebnisse mit `anyOf` vereint werden. Tabelle 5.3 illustriert dies anhand einiger Beispiele.

Tabelle 5.3: Beispiele für die Inversion von Schlüsselwörtern

Schlüsselwort a	Inversion $\overline{a}$
`"minimum":x`	`{"exclusiveMaximum":x}`
`"required":["x"]`	`{"properties":{"x":false}}`
`"type":x`	`{"type":["number","bool",...,"array"]\ [x]}`

Ferner kann eine Funktion `invert`(S) definiert werden, die ein normalisiertes Schema $S = \bigvee_i A_i$ invertiert:

$$\begin{aligned} \text{invert}(S) = \overline{S} &= \overline{\bigvee_i A_i} \\ &= \bigwedge_i \overline{A_i} = U \\ &= \text{merge}(\text{invert}^*(A_1), \text{invert}^*(A_2), ...) \end{aligned}$$

Das Zwischenergebnis U lässt sich als folgendes Schema darstellen:

```
allOf:
- invert*(A1)
- invert*(A2)
- ...
```

U kann dann, wie im vorherigen Abschnitt beschrieben, mithilfe von `merge` in ein normalisiertes Schema überführt werden. Zur Illustration sei folgendes normalisiertes Schema S gegeben:

```
anyOf:
- type: ["number"]
  minimum: 10
- type: ["string", "object"]
```

Die Anwendung von `invert`$(S) \mapsto$ {`"not":S`} liefert:

```
anyOf:
- type: ["null", "boolean", "array"]
- type: ["number"]
  exclusiveMaximum: 10
```

Somit kann mithilfe von `invert` das Schlüsselwort `not` in jedem Schema ersetzt werden.

Implementierungsdetails

Die Funktionen `invert` und `merge` erwarten normalisierte Schemata als Eingabe. Wenn dies nicht der Fall ist, müssen sie zunächst normalisiert werden. Dadurch wird die Normalisierung rekursiv. Durch die JSON-Schema-Spezifikation [32] ist garantiert, dass alle logischen Applikatoren beliebig tief, aber nicht rekursiv verschachtelt werden. Dadurch ist sichergestellt, dass die vorgestellte Normalisierung terminiert.

Bei der Implementierung von `merge` gilt, wie oben beschrieben:

$$\text{merge}(S, T) = \bigvee_i \bigvee_j \text{merge*}(A_i, B_j)$$

Es müssen folglich alle möglichen Kombinationen $A_i \times B_j$ gebildet und mit `merge*` zusammengeführt werden. Dies kann bei großen Schemata wie dem der Verwaltungsschale zu komplex werden. Es hat sich daher bewährt, die Unterschemata nur *paarweise* mit `merge*` zusammenzuführen. So wächst der Aufwand nicht mehr quadratisch sondern nur noch linear mit der Anzahl der Einträge in den Subschemata. Des weiteren wird die vorgestellte Normalisierung nicht schrittweise implementiert, sondern alle vorgestellten booleschen Gleichungen kombiniert. Dadurch kann die Normalisierung in einem Schritt durchgeführt werden und ist schneller.

5.3.3 Referenzen isolieren

Ein Schema kann Referenzen `$ref` enthalten, z.B:

```
properties:
  children:
    $ref: "#/"
    type: object
```

Ein `$ref` vermischt mit anderen Schlüsselwörtern führt zu Problemen bei der Sample-Erzeugung, außer wenn dieses das einzige Schlüsselwort im Schema ist. Um dies zu erreichen, muss das Ziel der Referenz isoliert werden, d.h. alle Inhalte des Ziels werden in das

Schema kopiert, das das Schlüsselwort `$ref` enthält. Allerdings kann dies zu Endlosschleifen in rekursiven Schemata führen. Um dies zu verhindern, werden zunächst Platzhalter-Referenzen für jedes Schema, das Referenzen enthält, eingeführt.

```
properties:
  children:
    $ref: "#/$defs/DUMMY"
$defs:
  DUMMY:
    properties:
      children:
        $ref: "#/"
        type: object
```

Im nächsten Schritt können in den Platzhalter-Schemata Referenzen isoliert werden. Im Beispiel wird dafür `type: object` „verschoben“:

```
  properties:
    children:
      $ref: "#/$defs/DUMMY
  $defs:
    DUMMY:
      type: object
      properties:
        children:
          $ref: "#/$defs/DUMMY"
```

Dadurch wurde die Referenz isoliert, was nachfolgende Schritte vereinfacht.

5.3.4 Flussgraph konstruieren

Mithilfe des Flussgraphen werden alle Freiheitsgrade des Metamodells modelliert. Die Konstruktion des Flussgraphen aus dem normalisierten Schema wird im Folgenden beschrieben.

Elemente eines Flussgraphen

Ein Flussgraph ist ein gerichteter Graph, dessen Knoten über Kanten verbunden sind. Die verschiedenen Knoten des Flussgraphen zeigt Abbildung 5.3.

Jedem Knoten ist eine bestimmte Operation zugeordnet, die von der Art des JSON-Wertes abhängt, der erzeugt werden soll. Beispiele für Operationen sind das Anlegen von Attributen in einem Objekt oder das Anfügen von Elementen an ein Array. Beim Durchlaufen eines Pfades durch den Flussgraphen werden alle Operationen der Knoten auf dem Pfad entsprechend deren Reihenfolge ausgeführt.

Bei Knoten wird zwischen inneren Knoten und Blättern unterschieden. Ein Blatt ist ein Knoten ohne ausgehende Kanten. Diese werden als valide oder invalide markiert. Einem **validen Blatt** ist eine Operation zugeordnet, die korrekt ist. Einem **invaliden Blatt** ist eine Operation zugeordnet, die inkorrekt ist und (absichtlich) einen Fehler im JSON-Objekt einfügt.

Ein innerer Knoten ist ein Knoten mit ausgehenden Kanten. Im Gegensatz zu Blättern sind inneren Knoten immer korrekte Operationen zugeordnet. Bei **Entscheidungsknoten** wird bei der Pfadauswahl genau ein Nachfolgerknoten ausgewählt. Im Gegensatz dazu gibt

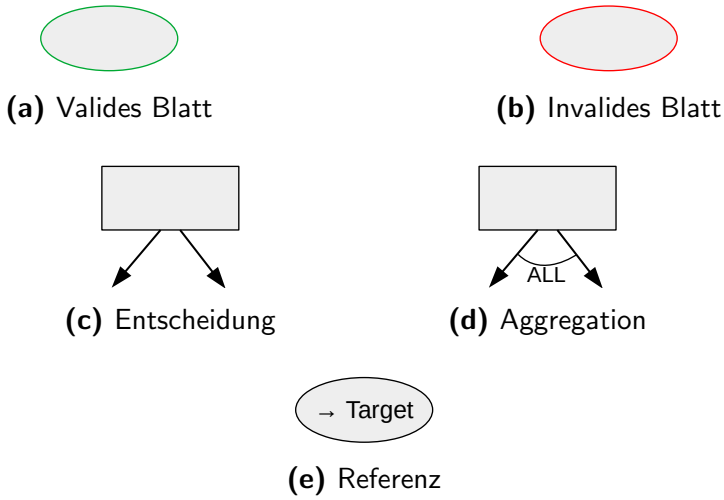


Abbildung 5.3: Knoten eines Flussgraphen

es **Aggregationsknoten**. Hier wird bei der Pfadauswahl *allen* nachfolgenden Knoten (in vorgegebener Reihenfolge) gefolgt.

Zuletzt gibt es noch ein spezielles Blatt, die **Referenz**. Diese zeigt auf einen bestimmten anderen Knoten, der nach der Referenz ausgeführt werden muss.

Abbildung

Das normalisierte JSON-Schema wird auf Elemente des Flussgraphen abgebildet. Gemäß Definition 2 besteht das normalisierte Schema aus einem einzigen `anyOf` Eintrag. Dieser wird auf eine Entscheidung ohne Operation abgebildet. Für jeden Eintrag im `anyOf` Array wird dann genau ein separater Subgraph erzeugt. Gemäß Definition 2 ist jeder Eintrag ein triviales Schema. Daher wird zuerst die `type` Bedingung betrachtet. Für jeden erlaubten Typen wird dann wieder ein Subsubgraph erzeugt. Die generelle Struktur des Flussgraphen zeigt Abbildung 5.4.

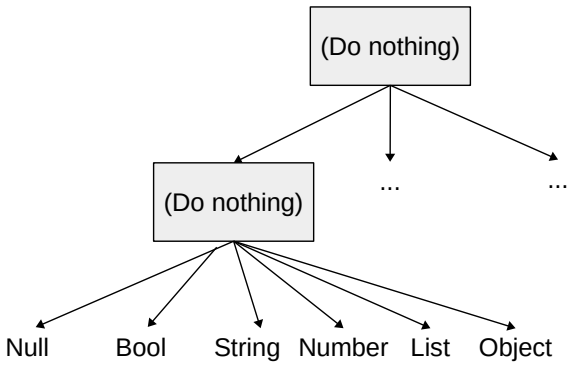


Abbildung 5.4: Abbildung eines normalisierten Schemas auf einen Flussgraphen

Für boolesche Werte und den `null` Wert existieren keine Bedingungen in JSON-Schemata. Hierfür werden Blätter mit den Operationen erstellt, vordefinierte Werte einzufügen, nämlich `true` und `false` bzw. `null`.

Für Zahlen liefert jede Bedingung eine Gleichung. Folglich liefern alle Bedingungen für Zahlwerte ein System von Gleichungen. Die Lösungen dieser Gleichungen bilden mögliche valide Werte. Für den Flussgraphen werden Lösungen an den Rändern gewählt als valide Werte bzw. Lösungen außerhalb als invalide Werte. Dies wird illustriert in Abbildung 5.5.

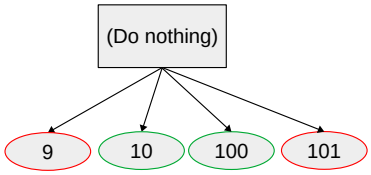


Abbildung 5.5: Abbildung von `{"minimum":10, "maximum":100}`

Eine ähnliche Methode wird für die Abbildung von Zeichenketten verwendet, wie Abbildung 5.6 illustriert.

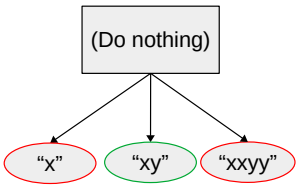


Abbildung 5.6: Abbildung von `{"minLength: 2, "maxLength: 3}`

Für Objekte erzeugt der Wurzelknoten des Subgraphen zunächst ein leeres Objekt: `{}`. Für jedes Attribut wird dann eine Kante zu einem passenden Subgraph angelegt (Abbildung 5.7). An dieser Stelle muss nach dem Wurzelknoten *allen* nachfolgenden Knoten gefolgt werden, da alle Attribute betrachtet werden sollen.

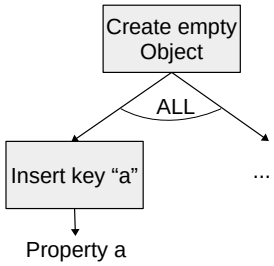


Abbildung 5.7: Abbildung von `{"properties":{"a":...}}`

Für Arrays wird der Flussgraph aus Abbildung 5.8 erzeugt. Der Subgraph enthält eine Schleife, bei dem entweder weitere Elemente ans Array angehängt werden oder die Iteration beendet wird.

Referenzen auflösen

Jedes Schema mit dem Schlüsselwort `$ref` wird auf einen Referenzknoten abgebildet. Später werden alle diese Referenzen aufgelöst. Dies bedeutet, dass jeder Referenzknoten ersetzt wird durch eine Kante zum Ziel der Referenz. Das Ergebnis dieses Schritts ist somit ein großer Gesamtflussgraph, der keine Referenzen mehr enthält. Ein Beispiel für diesen Prozess zeigt Abbildung 5.9.

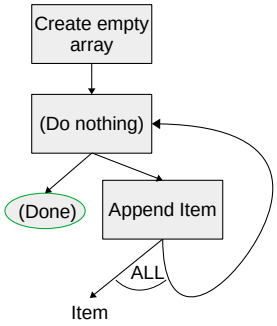
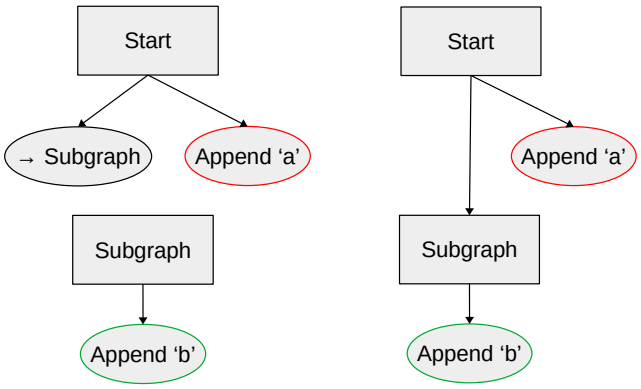


Abbildung 5.8: Abbildung, von `{"items":... }`



(a) Vorher **(b)** Nachher

Abbildung 5.9: Auflösung von Referenzen

5.3.5 Flussgraph erweitern

Die bisherigen Schritte sind unabhängig von der Verwaltungsschale und können mit beliebigen JSON-Schemata umgehen. Das Ziel des nächsten Schrittes ist, den Flussgraph für den Einsatz für die Verwaltungsschale zu erweitern. Dafür werden spezielle Knoten manuell eingefügt.

Wie in Unterabschnitt 4.2.3 erörtert, lassen sich nur wertbezogene und bedingte Constraints im JSON-Schema formalisieren. Daher erzeugt der bis zu diesem Schritt konstruierte Flussgraph nur Beispieldaten, die diesen Constraints (und dem Metamodell) genügen. Allerdings werden diese Beispieldaten im Allgemeinen nicht den *komplexen* Constraints genügen. Dafür werden im Flussgraph entsprechende Knoten eingefügt, die die erzeugten Daten reparieren, sodass sie den komplexen Constraints genügen. Als Beispiel sei Constraint AASd-119 gegeben:

> **Constraint AASd-119:** If any Qualifier/kind value of a Qualifiable/qualifier is equal to TemplateQualifier and the qualified element inherits from "hasKind", the qualified element shall be of kind Template (HasKind/kind = "Template").

Hierfür wird ein Knoten eingefügt, der `kind` auf den Wert `Template` setzt, wenn mindestens einer der Qualifier ein `TemplateQualifier` ist. Diese Knoten werden für jeden

komplexen Constraint implementiert und im Flussgraph an den passenden Stellen eingehängt.

5.3.6 Pfade auswählen und ausführen

Zuletzt werden Pfade im Flussgraph ausgewählt. Indem man einem Pfad folgt, werden alle Operationen an den Knoten entlang des Pfades ausgeführt. Dadurch entsteht eine Testdatei, d.h. eine konkrete Instanz einer Verwaltungsschale. Die Besonderheit bilden die in Unterabschnitt 5.3.4 eingeführten Aggregationsknoten. Ohne diese ließen sich existierende Algorithmen zur Pfadauswahl verwenden, wie z.B. A* oder der Dijkstra-Algorithmus [91]. Allerdings können Pfade im Rahmen dieser Arbeit „Abzweigungen“ enthalten, was bei Pfaden in Graphen normalerweise nicht der Fall ist. Ferner können Pfade durch Aggregationsknoten auch mehrere Blätter enthalten.

Ein Pfad ist eine Liste von Knoten, die an Stellen ausgewählt werden, an denen nach Entscheidungsknoten der nächste Knoten gewählt wird. In diesem Kontext ist ein **valider Pfad** ein Pfad, der nur valide Blätter erreicht. Im Gegensatz dazu ist ein **invalider Pfad** ein Pfad, der nur valide Blätter erreicht und zusätzlich genau ein invalides Blatt. Dieses invalide Blatt führt zu genau einem Defekt in der erzeugten Verwaltungsschale, der später vom SDK erkannt werden muss.

Die Auswahl von Pfaden wird solange durchgeführt, bis *alle* Knoten erreicht wurden. Dadurch ist sichergestellt, dass alle Freiheitsgrade des Metamodells durch die Testdaten abgedeckt werden. Die Pfadauswahl basiert auf dem in [23] vorgestellten Algorithmus, der in Listing 5.1 gezeigt ist. Die Schwierigkeit liegt in der korrekten Handhabung von Aggregationsknoten.

Zunächst wird eine Liste von noch zu besuchenden Knoten angelegt, die mit allen Blättern im Flussgraph initialisiert wird. Anschließend werden solange Pfade generiert, bis alle Blätter erreicht sind. Hierfür wird jeweils ein Blatt ausgewählt, für das als nächstes ein Pfad erzeugt werden soll. Dies geschieht durch den Aufruf der Methoden `backward` und `forward`.

Listing 5.1: Pfadauswahl: Einstiegspunkt

```
def select_paths():
  to_visit = all_leafs()
  result = []
  while to_visit:
    target = to_visit[0]
    backward_path = backward(target)
    forward_path = forward(backward_path)
    result.append(forward_path)
    # Remove all visited nodes from to_visit
  return result
```

Die Methode `backward` ist in Listing 5.2 dargestellt. Sie findet, gegeben ein Startknoten, einen Pfad von diesem Knoten zur Wurzel des Flussgraphen. Dabei werden die Aggregationsknoten zunächst nicht gesondert behandelt.

Listing 5.2: Pfadauswahl: Rückwärtsschritt

```
def backward(node, path):
  if not node.incoming_edges:
    return
  # Select incoming edge with shortest path to root
  idx, previous = select1(self.incoming_edges)
  path.append(idx)
  backward(previous, path)
```

Der gefundene Pfad zur Wurzel wird dann an `forward` übergeben, die nun die Aggregationsknoten korrekt einbezieht. Dafür wird für alle Nachfolgerknoten eines Aggregationsknotens die Methode `generate` aufgerufen, die den Pfad für den Nachfolgerknoten komplettiert (siehe Listing 5.4).

Listing 5.3: Pfadauswahl: Vorwärtsschritt

```
def forward(node, backward_path, forward_path):
  if len(backward_path) == 0:
    return
  path_idx = backward_path.pop(-1)
  if node.all_transitions:
    for idx, edge in node.outgoing_edges:
      if idx == path_idx:
        forward(edge.target, backward_path, forward_path)
      else:
        generate(edge.target, forward_path)
  else:
    edge = self.outgoing_edges[path_idx]
    forward_path.append(path_idx)
    forward(edge.target, backward_path, forward_path)
```

Listing 5.4: Pfadauswahl: Komplettierung eines Teilpfades

```
def generate(node, forward_path):
  if not node.outgoing_edges:
    return
  if node.all_transitions:
    for edge in node.outgoing_edges:
      generate(edge.target, forward_path)
  else:
    # Select edge with shortest path to a valid leaf
    idx, edge = select2(node.outgoing_edges)
    forward_path.append(idx)
    generate(edge.target, forward_path)
```

Abschließend werden die gefundenen Pfade ausgeführt, um die Testdaten zu erhalten.

Beispiel

Folgendes Schema soll als Beispiel dienen:

```
anyOf:
- type: number
  minimum: 10
  maximum: 20
- type: array
  items:
    $ref: "#/"
```

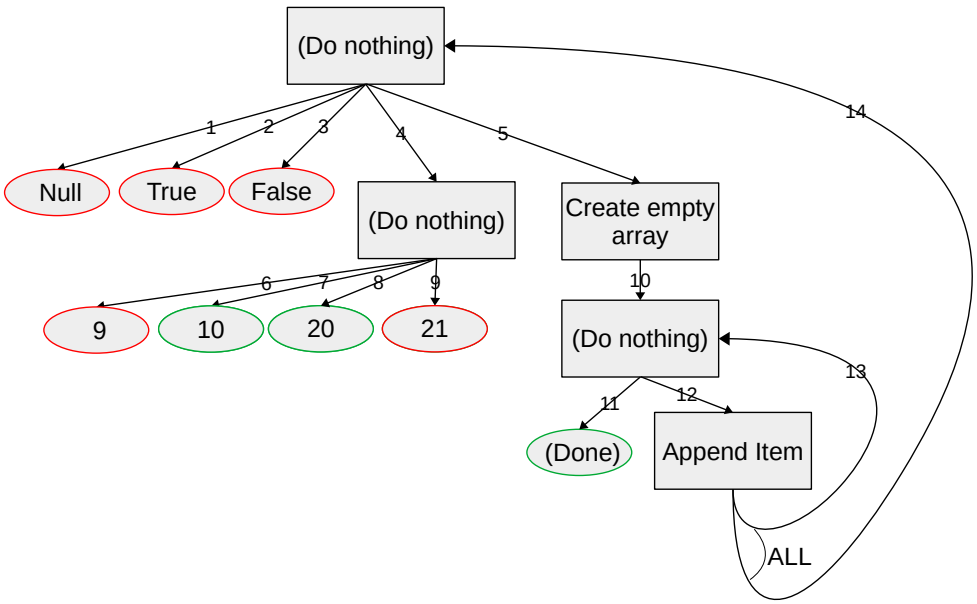


Abbildung 5.10: Beispiel für die Abbildung eines Flussgraphen

Das Schema wird auf den Flussgraph aus Abbildung 5.10 abgebildet. Hervorzuheben ist hierbei, dass der Flussgraph eine Schleife enthält. Diese entsteht durch das rekursive Schema.

Für die Pfadauswahl werden nun, wie in Listing 5.1 dargestellt, solange Pfade ausgewählt, bis alle Knoten erreicht wurden. Angenommen, der Algorithmus würde zunächst den `Append Item` Knoten auswählen. Dann wird hierfür mithilfe von Listing 5.2 der `backward_path` bestimmt: 12, 10, 5. Danach wird der `forward_path` berechnet, gemäß Listing 5.3. Dieser arbeitet den `backward_path` zunächst rückwärts ab: 5, 10, 12. Nun muss der Pfad noch komplettiert werden. Es muss nach `Append Item` *allen* ausgehenden Kanten gefolgt werden. Die Kante 13 kann direkt geschlossen werden mit 11, sodass der Pfad bis hier lautet: 5, 10, 12, 13, 11. Nun muss noch der Pfad über 14 geschlossen werden, z.B. mit 4, 6. Somit lautet der komplette Pfad: 5, 10, 12, 13, 11, 14, 4, 6. Führt man diesen Pfad wiederum aus, ergibt sich das JSON-Dokument `[10]`, also eine Liste mit einem Element. Bei dem gewählten Pfad handelt es sich um einen validen Pfad. Somit ist auch das erzeugte Dokument valide und erfüllt in der Tat das Ausgangsschema.

5.4 Sicherstellung der Qualität des Verfahrens

Im Nachfolgenden werden einige Maßnahmen vorgestellt, um die Korrektheit des Verfahrens sicherzustellen. Das Verfahren ist allgemeingültig in der Art, dass es beliebige JSON-Schemata verarbeiten kann. Der einzige Schritt, der spezifisch für die Verwaltungsschale ist, ist die Erweiterung des Schemas (siehe Unterabschnitt 5.3.5). Daher sind die nachfolgenden Maßnahmen ebenfalls allgemeingültig. Dementsprechend werden für die Sicherstellung der Korrektheit die folgenden Testkorpora verwendet:

- Die offizielle JSON-Schema-Testsuite [92].
- Das in Kapitel 4 entwickelte JSON-Schema für die Verwaltungsschale.

Bei der JSON-Schema-Testsuite handelt es sich um einen Satz von synthetischen JSON-Schemata. Für jedes Schema gibt es einen Datensatz von JSON-Dokumenten, von denen vorgegeben ist, ob sie das Schema erfüllen / verletzen.

5.4.1 Korrektheit

Ein wesentlicher Bestandteil des vorgestellten Verfahrens besteht darin, das Ausgangsschema in ein normalisiertes Schema zu überführen. Durch die Anwendung von boolescher Algebra sind das Ausgangsschema und das normalisierte Schema äquivalent. Dies bedeutet, dass jedes JSON-Dokument, das vom Ausgangsschema akzeptiert wird, auch vom normalisierten Schema akzeptiert wird. Für diese Betrachtung wird der Ablauf verwendet, der in Abbildung 5.11 dargestellt ist.

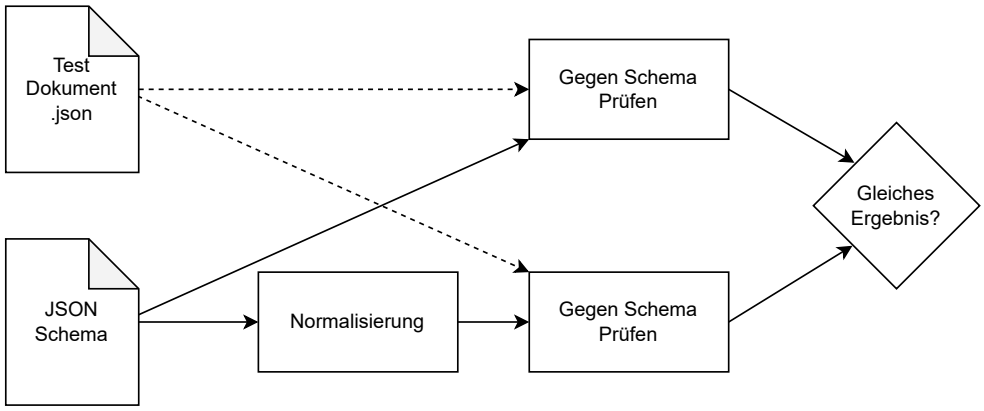


Abbildung 5.11: Sicherstellung der Korrektheit der Normalisierung (nach [13])

Jedes Schema der JSON-Schema-Testsuite wird zunächst normalisiert. Im Anschluss wird überprüft, dass für jedes Testdokument das Ausgangsschema und das normalisierte Schema dasselbe Ergebnis liefern. Da dies der Fall ist, können Implementierungsfehler bei der Normalisierung weitestgehend ausgeschlossen werden.

Weiterhin kann die Korrektheit der erzeugten Beispieldaten überprüft werden. Dafür werden sie gegen das Eingangsschema validiert. Korrekte Testdaten müssen dann akzeptiert werden, inkorrekte abgelehnt werden.

Tabelle 5.4: Ergebnisse der Beispielgenerierung für die JSON-Schema-Testsuite (aus [13])

	Manuell	Hypothesis	Fences
Samples	489	11.048	1.944
Schema Abdeckung	100%	96%	100%
Zeit	-	145s	< 1s

Beide Methoden zur Sicherstellung der Korrektheit wurden auf die Implementierung des vorgestellten Verfahrens angewandt. Dabei konnten weder mit Schemata aus der JSON-Schema-Testsuite noch mit dem Schema der Verwaltungsschale Fehler in der Implementierung gefunden werden. Damit konnte die Korrektheit des Verfahrens und der Implementierung experimentell sichergestellt werden.

5.4.2 Vollständigkeit

Im zweiten Schritt wird die gesamte Testdatengenerierung betrachtet mit dem Fokus auf Vollständigkeit. Mit Vollständigkeit ist hierbei gemeint, dass tatsächlich alle Freiheitsgrade des Metamodells durch Testdaten abgebildet werden. Hierfür wird das in [13] definierte Maß der Schema-Abdeckung verwendet. Zur Illustration sei folgendes Beispiel gegeben:

```
required:
- name
properties:
  address:
    type: string
```

Für jedes JSON-Schema kann die Anzahl der Bedingungen gezählt werden. So hat das obige Beispiel zwei Bedingungen, nämlich `/requried/name` und `/properties/address/type`.

Wird nun ein JSON-Dokument gegen das Schema geprüft, dann werden oft nicht alle Bedingungen evaluiert. Beispielsweise evaluiert das JSON-Dokument {`"name: ["Peter", "Pan"]`} nur die Bedingung `/required/name`. Die zweite Bedingung `/properties/address/type` kann nicht evaluiert werden, weil es kein `address` Attribut in dem Dokument gibt. Somit kann auch die Anzahl tatsächlich evaluierter Bedingungen bestimmt werden. Damit kann die Abdeckung definiert werden als:

$$\text{Schema-Abdeckung} = \frac{\text{Anzahl evaluierter Bedingungen}}{\text{Anzahl aller Bedingungen}}$$

Also im obigen Beispiel 50%. Werden nun weitere Dokumente gegen das Schema geprüft, können diese weitere Bedingungen evaluieren und damit die Anzahl evaluierter Bedingungen und somit die gesamte Abdeckung erhöhen.

Zur Bewertung des vorgestellten Verfahrens wird zunächst die Schema-Abdeckung für die JSON-Schema Test Suite berechnet. Das Ergebnis zeigt Tabelle 5.4.

Es zeigt sich, dass Fences in der Lage ist, die volle Abdeckung zu erreichen - so wie die manuellen Beispiele aus der JSON-Schema Test Suite. Hypothesis erreicht ebenfalls eine sehr hohe Abdeckung, benötigt hierfür aber deutlich länger.

Eine vergleichbare Betrachtung kann auch für das in Kapitel 4 entwickelte JSON-Schema durchgeführt werden. Hypothesis scheitert am JSON-Schema der Verwaltungsschale aufgrund der Rekursion und der Komplexität. Mithilfe des in dieser Arbeit vorgestellten Verfahrens kann eine Schema-Abdeckung von 98% erreicht werden. Es bleibt die Frage, wie aussagekräftig die Schema-Abdeckung als Metrik für die Vollständigkeit ist. Dafür ist in Abbildung 5.12 die Abdeckung für eine steigende Anzahl von Testdaten dargestellt.

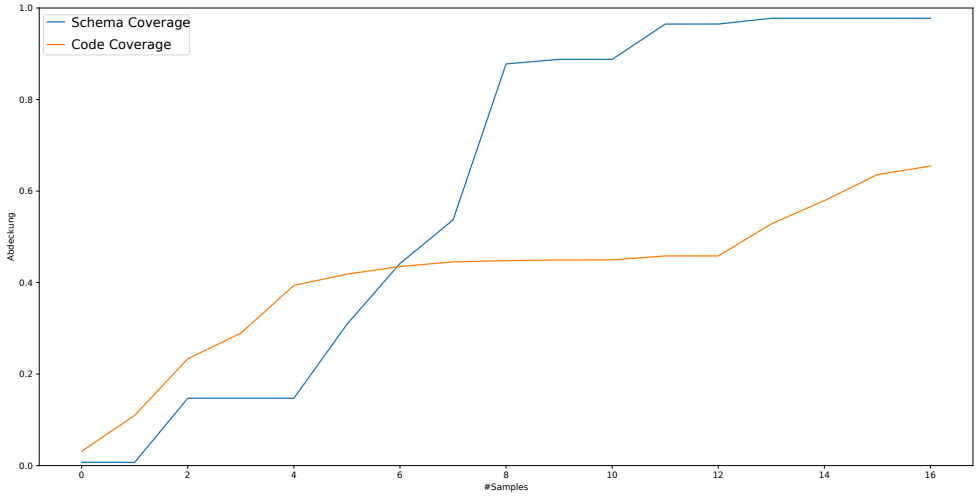


Abbildung 5.12: Code-Abdeckung und Schema-Abdeckung

Verglichen werden hier die oben beschriebene Schema-Abdeckung und die Code Abdeckung. Die Code Abdeckung wurde hierbei gemessen, indem die Testdaten mit dem Testadapter aus Listing 5.5 ans AAS Core SDK gereicht wurden. Anschließend wurde gemessen, wieviele Zeilen Code dadurch ausgeführt wurden. Beide Abdeckungskriterien sind lose korreliert: Wenn die Schema-Abdeckung steigt, steigt auch die Code-Abdeckung. Wenn die Code-Abdeckung ihr Maximum erreicht, liegt die Code-Abdeckung erst bei 65%. Das liegt daran, dass die Codedateien mehr Funktionen enthalten, als durch den Testadapter aufgerufen werden, wie z.B. das Iterieren über Submodellelemente.

5.5 Evaluation

In diesem Abschnitt wird der Testaufbau für verschiedene SDKs durchgeführt um deren Konformität zu messen und die Leistungsfähigkeit der vorgestellten Methode zu evaluieren.

5.5.1 AAS Core SDKs

Bei den AAS Core SDKs handelt es sich um eine Familie von SDKs, die alle aus derselben Quelle, einer Formalisierung des Metamodells in Python, stammen. Im Nachfolgenden wird das AAS Core C# SDK (Version 1.0.3) betrachtet, repräsentativ für alle AAS Core SDKs. Zunächst wird gemäß Abbildung 5.1 ein Testadapter speziell für dieses SDK implementiert, der in Listing 5.5 dargestellt ist.

Listing 5.5: Testadapter für das AAS Core C# SDK

```
void Check(inFile, outFile) {
  var text = File.ReadAllText(inFile);
  var jsonNode = JsonNode.Parse(text);

  // Check metamodel
  Aas.Environment environment;
  try {
    environment = EnvironmentFrom(jsonNode);
  } catch(Jsonization.Exception e) {
    return;
  }

  // Check constraints
  var errors = Verify(environment);
  if(errors.Any()) {
    return;
  }

  // Serialize again
  var jsonObject = ToJsonObject(environment);
  File.WriteAllText(outFile, jsonString);
}
```

Der Testadapter nimmt eine Datei mit einer serialisierten Verwaltungsschale als Eingabe. Diese wird mithilfe des AAS Core C# SDKs eingelesen und anschließend die Constraints überprüft. Wenn hierbei keine Fehler auftreten, wird die Verwaltungsschale in eine andere Datei wieder serialisiert.

Zur Auswertung werden die Ausgabedateien analysiert: Eingabedateien mit validen Verwaltungsschalen müssen vom SDK akzeptiert werden und somit eine entsprechende Ausgabedatei geschrieben werden. Eingabedateien mit invaliden Verwaltungsschalen müssen vom SDK abgewiesen werden und es existiert somit keine entsprechende Ausgabedatei. Dies wird für alle generierten Dateien durchgeführt. Die Ergebnisse zeigt Tabelle 5.5.

Tabelle 5.5: Testergebnisse AAS Core C#

	Valide	Invalide	Insgesamt
Akzeptiert	764	80	844
Abgewiesen	0	3896	3896
Gesamt	764	3976	4740

Es werden alle validen Verwaltungsschalen wie zu erwarten akzeptiert. Allerdings werden von den invaliden Verwaltungsschalen 80 fälschlicherweise akzeptiert. An dieser Stelle ist das AAS Core C# SDK nicht konform zur Spezifikation. Diese Testdateien werden manuell analysiert, um die Fehler im AAS Core C# SDK zu finden, die zu dem nicht-konformen Verhalten führen.

Einer der Fehler ist die fehlende Überprüfung des `modelType` Attributes. Dieses gibt bei der Deserialisierung an, welche Klasse des Metamodells instanziiert werden soll. Wird der `modelType` also weggelassen, dann stellt dies eine Verletzung der Spezifikation dar. Im Listing 5.6 fehlt das Attribut und dürfte somit nicht akzeptiert werden. Das AAS Core C# SDK akzeptiert die Instanz dennoch, was somit nicht konform ist.

Listing 5.6: Invalide Verwaltungsschale (`modelType` fehlt)

```
assetAdministrationShells:
- extensions:
  - name: x
  id: x
  assetInformation:
    assetKind: Instance
    globalAssetId: x
```

Als nächstes prüft das AAS Core C# SDK Constraint AASd-109 nicht korrekt, der besagt [3]:

> **Constraint AASd-109:** If SubmodelElementList/typeValueListElement is equal to Property or Range, SubmodelElementList/valueTypeListElement shall be set and all first level child elements in the SubmodelElementList shall have the value type as specified in SubmodelElementList/valueTypeListElement

Dieser Constraint impliziert, dass die Existenz von `typeValueListElement` die Existenz von `valueTypeListElement` fordert. Dies wird vom AAS Core C# SDK nicht geprüft, weswegen Listing 5.7 fälschlicherweise akzeptiert wird.

Listing 5.7: Invalide AAS (`valueTypeListElement` fehlt)

```
submodels:
- modelType: Submodel
  id: x
  submodelElements:
  - modelType: SubmodelElementList
    idShort: x
    typeValueListElement: Property
```

5.5.2 BaSyx Python SDK

Für das BaSyx Python SDK muss ebenfalls ein eigener Testadapter implementiert werden, der dieselbe Schnittstelle bietet, wie derjenige für das AAS Core C# SDK. Die Ergebnisse der Auswertung zeigt Tabelle 5.6.

Das BaSyx Python SDK ist ebenfalls an einigen Stellen nicht konform zur Spezifikation.

5.5.3 AAS4J

Zuletzt wurde das AAS4J SDK getestet. Die Ergebnisse zeigt Tabelle 5.7. Alle validen Verwaltungsschalen wurden korrekterweise akzeptiert. Allerdings wurde ein Großteil der

Tabelle 5.6: Testergebnisse BaSyx Python

	Valide	Invalide	Total
Akzeptiert	715	20	735
Abgewiesen	49	3956	4005
Gesamt	764	3976	4740

invaliden Verwaltungsschalen vom AAS4J SDK ebenfalls akzeptiert, was eine Verletzung der Spezifikation darstellt. Diese Verwaltungsschalen enthalten Verletzungen der Constraints, nicht des Metamodells. Da das AAS4J SDK die Prüfung von Constraints nicht implementiert [93], ist dieses Verhalten somit zu erwarten.

Tabelle 5.7: Testergebnisse AA4J

	Valide	Invalide	Total
Akzeptiert	764	1753	2517
Abgewiesen	0	2223	2223
Gesamt	764	3976	4740

5.5.4 Diskussion der Testergebnisse

Konformität der SDKs

In Unterabschnitt 2.3.2 werden verschiedene Metriken zur Messung von Konformität basierend auf Konfusionsmatrizen dargestellt. Ferner werden diese Metriken in [5] im Kontext von Verwaltungsschalen-SDKs diskutiert. Mithilfe dieser Metriken lässt sich die Konformität der SDKs beziffern. Damit können die SDKs verglichen werden, wie Tabelle 5.8 zeigt. Je nach verwendeter Metrik hat BaSyx Python oder AAS Core C# einen höhere Wertung. AAS4J hingegen, hat in jeder Metrik eine niedrigere Bewertung. Dies reflektiert die Tatsache, dass AAS4J die Constraints nicht implementiert.

Tabelle 5.8: Vergleich der Testergebnisse der SDKs

SDK	A_u	A_b
AAS Core C#	98%	**99%**
BaSyx Python	**99%**	97%
AAS4J	63%	78%

Eignung des Verfahrens

Das vorgestellte Verfahren ist in der Lage, Fehler in real existierenden SDKs aus dem Verwaltungsschalen Ökosystem aufzudecken. Insbesondere ist das Verfahren in der Lage mit dem komplexen Metamodell der Verwaltungsschale umzugehen, woran einige Verfahren

(wie z.B. Hypothesis) scheitern. Dabei werden auch komplexere Fehler gefunden, wie die fehlende Überprüfung von Constraints.

Ein großer Vorteil im Vergleich zu anderen Verfahren (wie z.B aus [66]), ist die Generierung von Testdaten auf Ebene der Serialisierung (JSON/XML) statt auf Ebene des Metamodells. Dadurch können auch diffizile Fehler gefunden, wie fehlende `modelType` Attribute. Außerdem ist das Verfahren aufgrund der Konstruktion vollständig, d.h. es werden tatsächlich alle Freiheitsgrade des Metamodells getestet.

Allerdings hat das Verfahren auch Grenzen: Zunächst werden Referenzen nicht aufgelöst und weiterverfolgt. Dies vereinfacht den Testaufbau deutlich, kann aber in der Praxis zu Problemen führen. Außerdem hat sich herausgestellt, dass manche Constraints nicht formal prüfbar sind. Diese können mit dem vorgestellten (aber auch keinem vergleichbaren) Verfahren nicht geprüft werden. Ferner erfolgt die Auswertung der gefundenen Fehler manuell. Das Verfahren liefert keine konkreten Hinweise auf Codestellen o.ä. Es bewertet auch nicht die Kritikalität eines gefundenen Fehlers.

5.6 Teilmodelle in Verwaltungsschalen-SDKs

Wie in Unterabschnitt 3.3.3 beschrieben, können Instanzen von Verwaltungsschalen vordefinierte Teilmodelle instanziieren. Zum Zeitpunkt der Veröffentlichung dieser Arbeit gibt es wenig dedizierte Tools zum Modellieren von Teilmodellen. So unterstützt der AASX Package Explorer [10] Teilmodelle, allerdings in einer generischen Weise und auch nur rudimentär. Ein dediziertes SDK für Teilmodelle wird in [77] vorgestellt, hierbei handelt es sich allerdings eher um eine Machbarkeitsstudie. Auch die AAS Test Engines [9] unterstützen das Überprüfen von Instanzen auf Konformität zu Teilmodellen. Hierbei wird die in Abschnitt 4.4 vorgestellte Methode verwendet.

5.6.1 Erzeugung von Instanzen

Die Bedeutung von Teilmodellen wird in Zukunft weiter zunehmen, ebenso die Anzahl der veröffentlichen Teilmodelle [14]. Daher wird auch in Zukunft die Anzahl an Tools und Software zunehmen, die für bestimmte Teilmodelle entwickelt wird. Im Rahmen dieser Arbeit wird daher ein Ansatz vorgestellt, um SDKs zusätzlich für bestimmte Teilmodelle zu testen. Der Ansatz basiert auf den in [15] vorgestellten Ergebnissen und erweitert die in diesem Kapitel vorgestellte Methode. Der grundlegende Testaufbau bleibt dabei derselbe: Zunächst werden Beispielinstanzen für ein Teilmodell automatisch erzeugt. Diese werden anschließend an das zu testende SDK übergeben, das die Instanzen deserialisert und überprüft. Dabei muss das zu testende SDK valide Instanzen akzeptieren sowie invalide Instanzen abweisen.

Die Erzeugung der Beispielinstanzen erfolgt automatisiert wie in Abbildung 5.13 dargestellt. Hierfür werden die Teilmodelle zunächst mithilfe von JSON-Schema beschrieben, wie bereits in Kapitel 4 vorgestellt. Um nun Beispielinstanzen für ein bestimmtes Teilmodell zu erzeugen, wird dessen JSON-Schema mit dem grundlegenden JSON-Schema der Verwaltungsschale kombiniert. Für das kombinierte Schema werden mithilfe der vorgestellten Flussgraphmethode nun Beispielinstanzen erzeugt. Abschließend muss noch der Prozess der Gruppierung wieder rückgängig gemacht werden. Die Gruppierung wurde in Kapitel 4 ursprünglich eingeführt, um Teilmodelle mithilfe von JSON-Schema effizient beschrieben

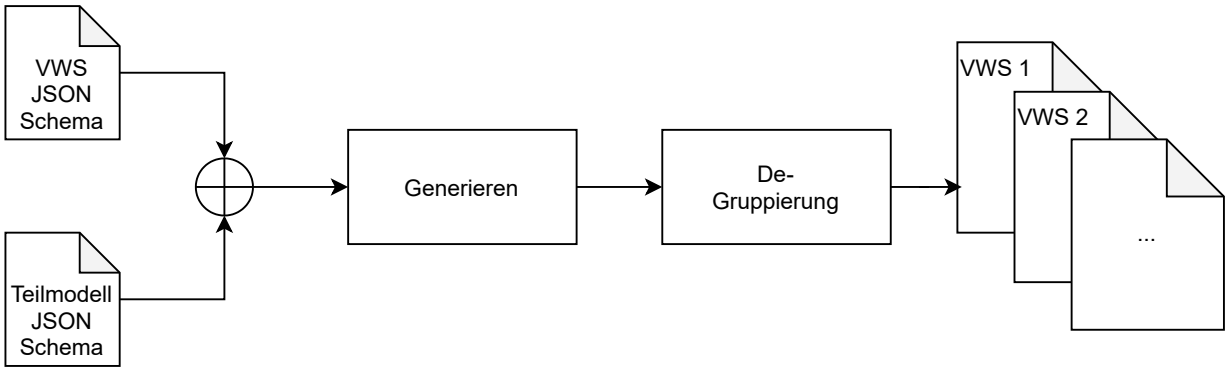


Abbildung 5.13: Erzeugung von Beispielinstanzen für Teilmodelle

zu können. Der Algorithmus zum De-Gruppieren ist in Listing 5.8 dargestellt. Hierbei wird das gruppierte Objekt wieder in eine Liste umgewandelt. Dafür wird über die Attribute im Objekt iteriert. Nach Konstruktion ist jedes Attribut hierbei wiederum eine Liste, deren Einträge an die Ergebnisliste angehängt werden.

Listing 5.8: De-Gruppierung nach idShort

```
def ungroup_by_id_short(obj):
  result = []
  for id_short, entries in obj.items():
    for entry in entries:
      entry["idShort"] = id_short
      result.append(entry)
  return result
```

5.6.2 Evaluation

Das Verfahren wurde anhand der Teilmodelle *Contact Information* sowie *Digital Nameplate for Industrial Equipment* evaluiert. Die Ergebnisse der Erzeugung zeigt Tabelle 5.9. Es wurden insgesamt 1140 Instanzen erzeugt, wobei für Digital Nameplate mehr Instanzen erzeugt wurden. Dies liegt daran, dass das Teilmodell Digital Nameplate das Teilmodell Contact Information enthält und somit zwangsweise auch alle Beispielinstanzen hierfür enthalten sind. Außerdem werden mehr valide als invalide Instanzen erzeugt, weil die Teilmodelle wenig Einschränkungen vorgeben. Die wesentlichen Einschränkungen sind die Anzahl der Elemente für eine bestimmte `idShort`. Außerdem sind einige Werte durch Aufzählungen vorgegeben, wie bei `RoleOfContactPersion`. Ferner gibt das Teilmodell *Digital Nameplate* vor, dass mindestens `ManufacturerProductFamily` oder `ManufacturerProductType` vorhanden sein muss. Dies führt zu einer weiteren invaliden Instanz, bei der diese beiden Merkmale absichtlich entfernt wurden.

Das Verfahren erbt die Vor- und Nachteile der grundlegenden Flussgraphenmethode. So erlaubt das Verfahren *systematisch* alle Freiheitsgrade von Teilmodellen abzudecken. Dies erhöht die Qualität der Tests und reduziert gleichzeitig den manuellen Aufwand. Auf der anderen Seite erfordert die Formalisierung, dass Entwickler und Tester sich mit JSON-Schema vertraut machen. Dies kann eine gewisse Hürde darstellen. Gerade im Zusammenhang mit der raschen Veröffentlichung von neuen Spezifikationen für Teilmodelle

Tabelle 5.9: Erzeugung von Beispielinstanzen für zwei Teilmodelle (aus [15])

	Contact Information	Digital Nameplate
Valide Instanzen	420	545
Invalide Instanzen	62	113
Gesamt	482	658

kann es daher sinnvoll sein, eine einfachere Formalisierung zu verwenden.

5.7 Fazit

SDKs bilden eine herausragende Rolle im Ökosystem der Verwaltungsschale, da sie sehr grundlegende Funktionen implementieren und somit in nahezu jeder Komponente verwendet werden. In diesem Kapitel wurde daher ein Testaufbau vorgestellt zum Testen der Konformität von SDKs. Hierbei werden valide und invalide Test-Verwaltungsschalen an das testende SDK übergeben, das diese dann entsprechend akzeptieren oder abweisen muss, um konform zu sein.

Die wesentliche Aussagekraft dieses Verfahrens hängt von den generierten Testdaten ab. Hierfür wurde ein Verfahren vorgestellt, um aus dem JSON-Schema der Verwaltungsschale automatisiert Testdaten abzuleiten. Dabei wird ein Flussgraph konstruiert, der alle Freiheitsgrade des Metamodells und der Constraints abbildet. Durch Ablaufen aller Knoten dieses Flussgraphen garantiert das Verfahren, dass auch alle Freiheitsgrade des Metamodells durch die Testdaten abgedeckt werden. Eine entsprechende Korrelation zwischen Schema-Abdeckung und Code-Abdeckung belegt diese Eigenschaft auch experimentell.

Abschließend wurde das Verfahren für verschiedene SDKs aus dem Ökosystem der Verwaltungsschale angewandt, um deren Konformität zu messen. Es zeigt sich, dass das Verfahren in der Lage ist, Fehler in diesen Implementierungen zu finden. Dabei werden auch diffizile Fehler gefunden, wie z.B. falsch implementierte Constraints.

Damit lassen sich Forschungsfragen 2-5 im Kontext von SDKs beantworten:

FF2: Welche Besonderheiten zeichnen die Verwaltungsschale aus softwaretechnischer Sicht aus? Welche speziellen Anforderungen an ein System zum Überprüfen der Konformität der bzgl. Interoperabilität relevanten Teile des Verwaltungsschalen-Ökosystems ergeben sich daraus?

Das Metamodell der Verwaltungsschale ist im Vergleich zu andren Anwendungen sehr komplex. Es besteht aus diversen Klassen und ist rekursiv. Darüber hinaus wird es durch Constraints eingeschränkt. Ein Testsystem für SDKs muss mit diesen besonderen Eigenschaften umgehen können.

FF3: Wie kann ein Testsystem für Softwarekomponenten der Verwaltungsschale umgesetzt werden?

Für den Konformitätstests muss ein Testdatensatz aus validen und invaliden Verwaltungsschalen erzeugt werden. Diese müssen vom zu testenden SDK akzeptiert bzw. abgelehnt werden. Die Testdaten sollten automatisiert erzeugt werden, um Anforderungen an Korrektheit und Vollständigkeit zu genügen. Hierfür wurde in dieser Arbeit ein Verfahren vorgestellt, dass Testdaten aus dem JSON-Schema der Verwaltungsschale erzeugt.

FF4: Welche Fehler können mit dem Testsystem in realen Softwarekomponenten für die Verwaltungsschale aufgedeckt werden? Welche Grenzen gibt es?

Das Verfahren kann grundsätzlich eine Vielzahl von syntaktischen Fehlern in SDKs finden. Dazu zählen z.B. fehlende oder falsch gesetzte Attribute. Fehler, die auf semantischer Ebene anzusiedeln sind, können nicht gefunden werden. Dazu zählen z.B. nicht-existierende Referenzen. Auch sollte das Lokalisieren, Identifizieren und Beheben von Fehlern in SDKs in Zukunft besser unterstützt werden.

FF5: Welche Vor- und Nachteile hat die automatische Ableitung von Konformitätstests aus formalen Spezifikationen?

Die Verwendung von JSON-Schema für die Generierung von Testdaten hat einige Vorteile: So können sehr viele und aussagekräftige Beispielinstanzen in kurzer Zeit erzeugt werden. Mithilfe des in dieser Arbeit vorgestellten Abdeckungsmaßes konnte sogar die Vollständigkeit dieser Testdaten gezeigt werden. Ein Nachteil der erzeugten Testdaten ist, dass sie nicht realitätsnah wirken sondern rein syntaktische Aspekte abdecken. Dies kann die Verständlichkeit für menschliche Tester erschweren.

6 Sicherstellung der Konformität von reaktiven Verwaltungsschalen

Die reaktive Verwaltungsschale wird in Form von Servern implementiert. Dafür wird in Part 2 der Spezifikation der Verwaltungsschale [8] eine HTTP/REST Schnittstelle beschrieben, um auf Server zuzugreifen. In diesem Kapitel wird ein Verfahren vorgestellt, um Verwaltungsschalenserver auf Konformität zu überprüfen. Um manuellen Aufwand zu minimieren wird auch hier auf eine formale Spezifikation zurückgegriffen, die OpenAPI-Beschreibung der Verwaltungsschale. Anschließend wird das Verfahren auf verschiedene Server aus dem Ökosystem der Verwaltungsschale angewandt. Das Ergebnis ist eine Bewertung der Konformität der Server sowie die Eignung des vorgestellten Verfahrens.

6.1 Anforderungen

Ein Testsystem für reaktive Verwaltungsschalen muss einigen Anforderungen genügen, die sich aus Anforderungen für REST APIs im allgemeinen sowie der Verwaltungsschale im speziellen ergeben [94]. Aus Sicht eines Testsystems für REST APIs im Allgemeinen ergeben sich folgende Anforderungen: Das Testsystem muss die generellen CRUD-Operationen überprüfen. Dies korrespondiert mit dem Lebenszyklus von Ressourcen, die erstellt, gelesen, geändert und gelöscht werden können. Diese vier Operationen müssen für alle Ressourcen der Verwaltungsschale aufgerufen werden, d.h. für Verwaltungsschalen, Submodelle und Submodellelemente. Ferner muss das Testsystem Pagination unterstützen und überprüfen. Dies bezeichnet den Vorgang, dass lange Listen von Ressourcen nicht komplett, sondern seitenweise ausgelesen werden. Ein Testsystem muss dies überprüfen und auch Fehlerfälle wie z.B. invalide Seitenzahlen betrachten.

Zu diesen allgemeinen Anforderungen kommen noch die speziellen Anforderungen der Verwaltungsschale hinzu: Zunächst finden sich die Anforderungen, die sich für passive Verwaltungsschalen ergeben, auch für reaktive Verwaltungsschalen wieder. Dazu zählen das rekursive, komplexe Metamodell sowie das Überprüfen der Constraints. Außerdem gruppiert die Spezifikation der Verwaltungsschale die Operationen nach Profilen (siehe Unterabschnitt 3.4.4). Ein Testsystem muss diese Gruppierung unterstützen. Weiter ermöglicht eine reaktive Verwaltungsschale den Zugriff auf Daten mithilfe von Superpfaden. Das bedeutet, dass einige Schnittstellen nur über dynamische Präfixe erreichbar sind. Für den Zugriff auf Submodellelemente über das Submodell-Interface ist die Bildung von idShort-Pfaden nötig. Dafür müssen `idShorts` aus dem Ergebnis traversiert und konkateniert werden. Eine weitere Besonderheit sind die drei Serialisierungsmodifier, die das Verhalten der Ausgabe steuern. Ein Testsystem muss diese implementieren und für jede sinnvolle Kombination überprüfen.

6.2 Verwandte Arbeiten

6.2.1 Verfahren zum Testen von reaktiven Verwaltungsschalen

Die Autoren von [95] betrachten verschiedene Implementierungen der reaktiven Verwaltungsschale. Dafür identifizieren sie sechs frei verfügbare Implementierungen der Version 3.0RC0 der Spezifikation. Drei davon werden auch im Rahmen dieser Arbeit betrachtet (AASX Server, Basyx Java sowie FA³ST). Die anderen drei Implementierungen unterstützen nicht die neueste Version 3.0 der Verwaltungsschale, auf die sich diese Arbeit bezieht. Ferner vergleichen die Autoren diese Implementierungen hinsichtlich ihrer Eigenschaften wie Features und Funktionsumfang. Darüber hinaus testen die Autoren verschiedene Endpunkte anhand von wenigen, manuell definierten Requests. Dadurch evaluieren sie Konformität der Implementierungen zum AAS Repository Interface, AAS Interface und dem Submodel Interface.

In [94] werden die Anforderungen an ein Testsystem für die Verwaltungsschale systematisch untersucht. Basierend darauf skizzieren die Autoren ein Testsystem. Eine erste automatisierte Erzeugung von Requests wird in [24] vorgestellt. Die Autoren verwenden hierfür die OpenAPI-Beschreibung der Verwaltungsschale. Außerdem heben sie hervor, dass die Operationen nicht unabhängig voneinander sind. Parameter und Bodys werden semi-automatisch erstellt. Der Ansatz wird in [9] aufgegriffen und verbessert. Die in dieser Arbeit vorgestellte Methode baut auf Erkenntnissen aus [9, 24] auf.

6.2.2 Generische Verfahren zum Testen von REST-APIs

In Unterabschnitt 2.3.1 wurden verschiedene Methoden zum Erzeugen von Testfällen vorgestellt. In diesem Abschnitt werden diese Methoden konkret im Kontext der Erzeugung von Testfällen für REST APIs betrachtet. Hervorzuheben ist, dass es generell abzuwägen gilt, wie viel Aufwand in die Vorbereitung und Konfiguration der Generierung investiert wird: Mit steigendem Aufwand steigt im Allgemeinen auch die Qualität der Testfälle. In den nachfolgenden Abschnitten werden bekannte Verfahrensprinzipien zusammenfassend beschrieben und bezüglich der Aufgabe des Testens reaktiver AAS-Systeme bewertet. Allen Verfahren ist gemeinsam, dass sie einen Satz von Requests erzeugen, die an den zu testenden Server gesendet werden. Anschließend werden die Antworten des Server darauf überprüft, ob sie den Erwartungen entsprechen.

Zufallsbasierte Verfahren

Bei zufallsbasierten Verfahren werden Requests zufällig erzeugt. Allerdings erreichen rein zufällige Requests in der Regel keine hohe Testqualität, weil sie z.B. nicht einmal gültige URLs treffen. Daher wurden diverse Verfahren entwickelt, um dieses Problem abzuschwächen. Eines davon ist Fuzzing, bei dem der Server bei der Ausführung der Tests beobachtet wird, um iterativ immer bessere Testrequests erzeugen können. Beispielsweise stellen die Autoren in [96] das Tool *RESTler* vor. Dieses Tool verwendet mehrere Quellen, um bessere Testergebnisse zu erzielen. Zum einen verwendet *RESTler* die OpenAPI-Beschreibung des Servers, um gültige URLs auszulesen und Abhängigkeiten zwischen den Endpunkten zu erraten. Zum anderen werden die Antworten des Servers auf vorherige Requests analysiert. `RESTler` eignet sich allerdings nur bedingt für Konformitätstests, wie die

meisten zufallsbasierten Verfahren. Das Tool dient vorrangig dazu, 5xx Fehler im Server auszulösen, die den Server ggf. zum Absturz bringen können.

Zufallsbasierte Verfahren sind nur sehr eingeschränkt für Konformitätstests einsetzbar. Dies liegt vor allem daran, dass sie nicht deterministisch sind und die eigentlichen Testfälle vom SUT abhängig sind. Dadurch sind die Ergebnisse zwischen verschiedenen SUTs nicht vergleichbar. Außerdem ist schwer sicherzustellen, dass alle relevanten Aspekte der Spezifikation abgedeckt sind.

Suchbasierte Verfahren

Bei suchbasiertem Testen werden iterativ Requests erzeugt, um eine bestimmte Zielfunktion zu maximieren. Ein bekanntes suchbasiertes Werkzeug ist *EvoMaster* [97]. EvoMaster nutzt einen Whitebox-Ansatz für Java-basierte Programme [98], um iterativ Testrequests zu erzeugen. Dafür verwenden die Autoren einen genetischen Suchalgorithmus, um iterativ die Testqualität zu verbessern. Auch hier wird die OpenAPI-Beschreibung des Servers als Hilfsmittel herangezogen. Suchbasierte Verfahren haben ähnliche Nachteile wie zufallsbasierte für Konformitätstests, weswegen das im Rahmen dieser Arbeit vorgestellte Verfahren nicht suchbasiert ist.

Kombinatorisches Testen

Beim kombinatorischen Testen werden vom Tester manuell Testrquests vorgegeben. Diese werden dann automatisch zu neuen Testrequests kombiniert. Für REST APIs wird dies beispielsweise in *RestCT* [99] implementiert. RestCT testet REST APIs systematisch in der Art, dass sowohl Interaktionen zwischen Operationen als auch spezifische Eingabeparameter-Interaktionen kombinatorisch abgedeckt werden. *RestCT* erreicht dies durch einen zweiphasigen Testfallgenerierungsprozess, der auf einer OpenAPI-Spezifikation basiert, um Abhängigkeiten automatisch zu erkennen und Testfälle ohne menschliches Eingreifen auszuführen. Experimentelle Ergebnisse mit realen APIs zeigen, dass *RestCT* effektiv und effizient ist und acht neue Bugs entdeckt, wobei nur einer davon mit dem modernsten Testtool für REST APIs gefunden werden konnte.

Rein kombinatorisches Testen ist oft sehr statisch, weil vorgefertigte Testfälle kombiniert werden. Das im Rahmen dieser Arbeit vorgestellte Verfahren verwendet Ideen aus dem kombinatorischen Testen, um manuell definierte Parameterwerte sinnvoll zu kombinieren. Es lässt sich im engeren Sinne aber nicht den kombinatorischen Testmethoden zuordnen.

Modellbasierte Verfahren

Bei modellbasierten Verfahren wird die zu testende REST API durch ein Systemmodell beschrieben und daraus automatisch Requests abgeleitet. Ein Großteil der Verfahren setzen dabei auf OpenAPI-Beschreibungen, um Testfälle abzuleiten. Ein modellbasiertes Tool ist *QuickRest* [100], das Property-based Testing einsetzt. Die Autoren erzeugten dabei sowohl Testrquests aus der OpenAPI-Beschreibung als auch die Überprüfung der Korrektheit der Antworten. Experimentelle Ergebnisse bei industriellen und Open-Source-Diensten zeigen, dass die Methode effektiv reale Fehler aufdeckt und Abweichungen zwischen Spezifikationen und Implementierungen aufzeigt.

Das Werkzeug *RestTestGen* [101] erzeugt ebenfalls Tests aus einer OpenAPI-Beschreibung. Die Autoren heben hervor, dass sich viele potenzielle Fehler erst durch

Interaktionen zwischen den einzelnen Operationen abdecken lassen. Dafür verwenden die Autoren einen Abhängigkeitsgraph, um systematisch Testfälle abzuleiten. Mit dem Werkzeug konnten Fehler in 87 realen REST APIs aufgedeckt werden.

Die genannten modellbasierten Werkzeuge und Konzepte decken am ehesten die Anforderungen für Konformitätstests für reaktive Verwaltungsschalen ab. Allerdings sind sie entweder zu unflexibel um Features wie Superpfade oder idShort-Pfade abzudecken. Oder die Beschreibung der komplexen reaktiven Verwaltungsschale ist nicht effizient genug. Daher verwendet der im Rahmen dieser Arbeit vorgestellte Ansatz Konzepte aus dem modellbasierten Testen lediglich um negative Testfälle abzuleiten.

6.3 Testaufbau

Zum Testen eines Servers werden an diesen ein Satz von Requests gesendet, die der Server mit einem Satz an entsprechenden Responses beantwortet. Wenn jeder Response den Erwartungen entspricht, wird der Server als konform zur Spezifikation angesehen. Voraussetzung dafür ist, dass der Satz der Requests alle Aspekte der Spezifikation abdeckt.

Ein korrekter Response bedeutet zunächst, dass der Status Code korrekt sein muss. Ferner muss, falls vorhanden, der Body des Response korrekt sein. Teilweise haben Requests auch Seiteneffekte, die überprüft werden, z.B. beim Anlegen von neuen Verwaltungsschalen. Die Test-Requests können manuell definiert werden, was allerdings zeitaufwendig und fehleranfällig ist. Daher werden im vorgestellten Verfahren die Requests aus der OpenAPI-Beschreibung der Verwaltungsschale semi-automatisch generiert. Den entsprechenden Aufbau zeigt Abbildung 6.1.

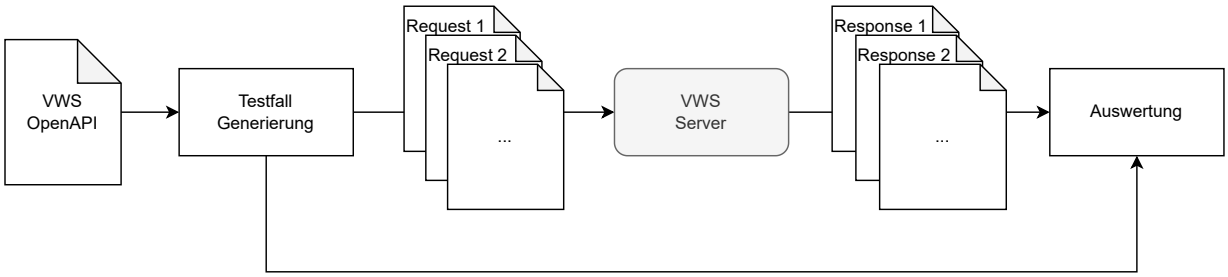


Abbildung 6.1: Überprüfung der Konformität einer reaktiven Verwaltungsschale

6.3.1 Struktur der Testfälle

Für diese Arbeit werden negative und positive Testfälle unterschieden. Wie in Unterabschnitt 2.2.1 beschrieben, enthält ein HTTP-Request einen Satz an Parametern sowie einen optionalen Body. Für **negative Testfälle** ist immer genau einer dieser Parameter auf einen invaliden Wert gesetzt. Alle weiteren Parameter werden entweder weggelassen, wenn sie optional sind, oder mit einem validen Wert belegt. Bei negativen Testfällen muss der Server den invaliden Parameterwert erkennen und abfangen, d.h. mit einem Statuscode zwischen 400 und 499 antworten. Außerdem muss die Antwort eine korrekt formatierte Fehlermeldung enthalten. Dadurch, dass immer genau ein Parameter auf einen invaliden Wert gesetzt wird, ist sichergestellt, dass es zu keiner Überlagerung von Fehlern kommen kann. Durch

Iteration über alle Parameter und alle möglichen Klassen von invaliden Parameterwerten können somit alle möglichen Fehlerfälle behandelt werden.

Für **positive Testfälle** werden alle Parameter auf valide Werte gesetzt. Der Server muss positive Testfälle akzeptieren und mit einem Statuscode zwischen 200 und 299 antworten. Außerdem muss die Antwort, je nach Operation, die korrekten Nutzdaten enthalten. Während negative Testfälle primär die Robustheit des Servers sicherstellen, decken positive eher reale Anwendungsfälle ab.

Die Struktur der Testfälle ist in Abbildung 6.2 dargestellt. Für einen Testlauf werden alle Operationen durchlaufen, die das zu überprüfende Profil unterstützt. Für jede Operation gibt es drei Phasen: Während des Setups werden die Voraussetzungen für die eigentlichen Tests geschaffen. Anschließend werden die negativen Testfälle ausgeführt, d.h. alle invaliden Parameterwerte iteriert. Zuletzt werden die positiven Testfälle ausgeführt. Zu beachten ist, dass die Operationen getrennt betrachtet werden, d.h. die einzelnen Testfälle unabhängig voneinander sind. Dies hat mehrere Gründe. Zum einen können so leicht verschieden Profile unterstützt werden. Diese werden dann durch die Auswahl der zu testenden Operationen definiert. Außerdem können einzelne Ergebnisse der Testausführung leichter reproduziert werden, was die Fehlerbehebung für Entwickler vereinfacht.

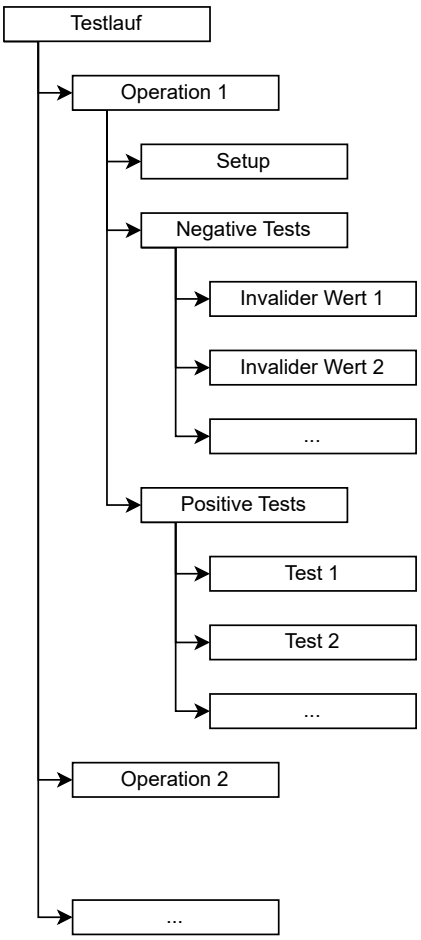


Abbildung 6.2: Generelle Struktur der Servertests

6.4 Generierung von Testfällen

Die Basis des Verfahrens bilden aussagekräftige Testfälle. Ein Testfall besteht aus einem Request, der an den zu testenden Server gesendet wird. Dafür müssen die einzelnen Parameter mit Werten belegt werden. Darüber hinaus enthält ein Testfall Erwartungen über den Inhalt der Antwort des Servers.

Die Erzeugung der Testfälle ist in Abbildung 6.3 dargestellt. Zunächst werden aus der OpenAPI-Beschreibung der Verwaltungsschale die Schemata und die Operationen extrahiert. Anhand der Schemata werden danach invalide Parameterwerte generiert. Valide Parameterwerte hingegeben werden manuell definiert oder während der Testausführung aus dem Server ausgelesen. Das Ergebnis ist für jeden Parameter ein Satz an validen und invaliden Werten. Im nächsten Schritt werden die Operationen der OpenAPI-Beschreibung betrachtet. Diese werden zunächst gefiltert entsprechend dem Profil, das überprüft werden soll. Abschließend erhält man die finalen Testfälle durch Einsetzen der Parameterwerte in die Parameter der gefilterten Operationen. D.h. für negative Testfälle werden valide Werte und jeweils genau ein invalider Wert eingesetzt. Für positive Testfälle werden ausschließlich valide Werte eingesetzt.

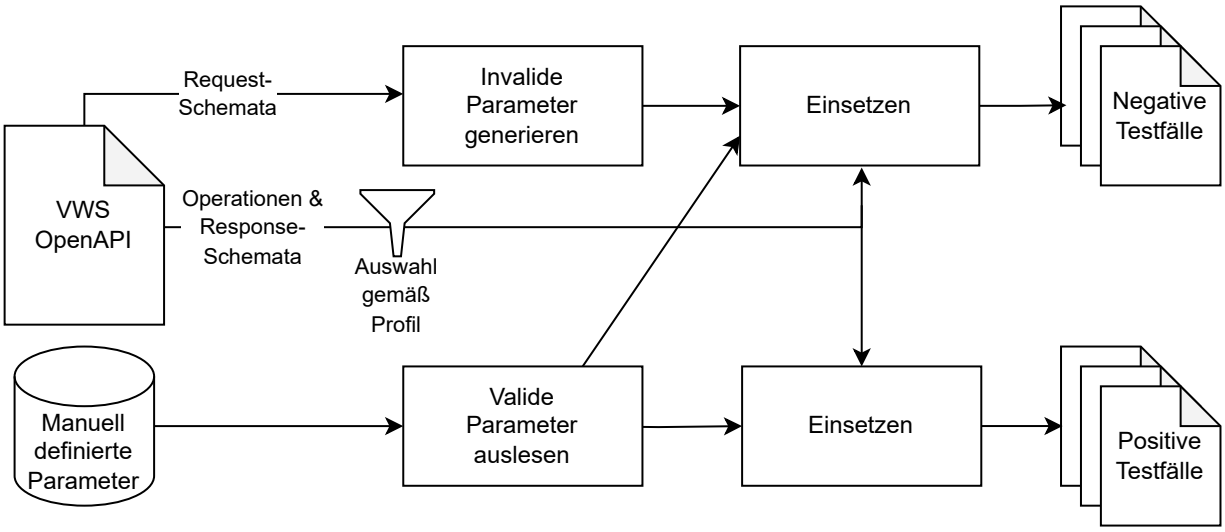


Abbildung 6.3: Generierung der Servertests

Die Generierung von Testfällen erfolgt separat für jede Operation. Das hat den Vorteil, dass die Testfälle unabhängig voneinander sind und somit einfacher reproduziert werden können. Außerdem vereinfacht es den Umgang mit Servicespezifikationen und Profilen erheblich, weil damit der Test eines Profils durch eine einfache Auswahl von Operationen durchgeführt werden kann. Der Nachteil dieser Vorgehensweise ist, dass bestimmte Requests unnötigerweise mehrfach durchgeführt werden.

6.4.1 Formalisierung mithilfe von OpenAPI

Grundlage des Verfahrens bildet die Formalisierung der Verwaltungsschale mithilfe der OpenAPI-Beschreibung (siehe Abschnitt 2.2 für eine allgemeine Beschreibung). Die OpenAPI-Beschreibung wird als Teil der Spezifikation veröffentlicht. Für diese Arbeit wird diese offizielle Beschreibung mit wenigen Anpassungen verwendet. Im

Nachfolgenden wird die OpenAPI-Beschreibung anhand eines Beispiels, der Operation `GetAllAssetAdministrationShells` illustriert.

Die Operation `GetAllAssetAdministrationShells` wird in [8, Seite 56] spezifiziert. Sie ist Teil des Interfaces für Verwaltungsschalen-Repositorys und dient dazu, alle Verwaltungsschalen in einem Repository aufzulisten. Dafür gibt es drei Eingabeparameter:

- **Limit** Die maximale Anzahl an Verwaltungsschalen, die zurückgegeben werden soll.
- **Cursor** Wenn sich mehr Verwaltungsschalen auf dem Server befinden, als mit `limit` angefragt wurden, gibt der Server einen `cursor` zurück. Dieser kann beim nächsten Aufruf mit angeben werden, um den nächsten Satz an Verwaltungsschalen abzurufen. Der `cursor` definiert somit eine Art Offset. Einige Implementierungen behandeln der `cursor` tatsächlich als numerischen Offset und geben `limit+1` zurück.
- **SerializationModifier** Es gibt drei verschiedene Modifier, die den Umfang der zurückgegebenen Daten beschreiben: Level, Content und Extend. Bei Verwaltungsschalen kann nur der Content-Modifier angewandt werden. Hiermit kann spezifiziert werden, ob die gesamten Verwaltungsschalen oder nur deren Referenzen zurückgegeben werden sollen.

Wenn die Operation erfolgreich ist, dann gibt der Server eine Liste von Verwaltungsschalen zurück. Außerdem kann, wie bereits beschrieben, ein `cursor` zurückgegeben werden, der für das Aufrufen weiterer Verwaltungsschalen genutzt werden kann (Pagination).

Die Syntax der Operation kann mithilfe von OpenAPI beschrieben werden. Dies ist in Listing 6.1 dargestellt.

Listing 6.1: Formalisierung von `GetAllAssetAdministrationShells`

```
paths:
  /shells:
    get:
      operationId: GetAllAssetAdministrationShells
      parameters:
      - name: limit
        in: query
        required: false
        schema:
          minimum: 0
          type: integer
      - name: cursor
        in: query
        required: false
        schema:
          type: string
      responses:
        "200":
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/GetAllShellsResult'
```

```
23          "400":
24            content:
25              application/json:
26                schema:
27                  $ref: '#/components/schemas/Result'
28    /shells/$reference:
29      ...
```

Die Operation wird über die URL `/shells` aufgerufen (Zeile 2), die zu verwendende Methode ist GET (Zeile 3). Die beiden Parameter werden einzeln aufgeführt (Zeile 5 ff.): Der Parameter `limit` wird als Queryparameter übergeben (Zeile 9) und ist optional (Zeile 8). Der Parameter selbst ist eine nicht-negative Zahl (Zeile 9-11). Der Parameter `cursor` ist ebenfalls ein optionaler Queryparameter und ein String.

Die Operation kann unterschiedliche Antworten liefern. Im Erfolgsfall wird ein Statuscode 200 zurückgegeben (Zeile 18). Der Body enthält dann eine Liste von Verwaltungsschalen, serialisiert als JSON. Im Fehlerfall wird ein Statuscode 400 zurückgegeben (Zeile 23). Der Body enthält dann Fehlerinformationen.

Der SerializationModifier `Content` wird - im Gegensatz zu den beiden anderen Modifiern `level` und `extent` - nicht direkt übergeben. Stattdessen ist dieser Teil der URL. Daher gibt es die beiden Endpunkte `/shells` (`content=normal`) sowie `/shells/$reference` (`content=reference`) (Zeile 28). Das Testsystem muss daher alle Testfälle der Operation für *beide* Endpunkte ausführen.

6.4.2 Invalide Parameterwerte generieren

Invalide Parameterwerte können voll-automatisch aus der OpenAPI-Beschreibung hergeleitet werden. Es stellt sich heraus, dass dafür das Verfahren aus Kapitel 5 genutzt werden kann, da dieses allgemein für beliebige JSON-Schemata funktioniert. Daher werden die JSON-Schemata der Parameter extrahiert und mit dem vorgestellten Verfahren Beispielwerte erzeugt. Zur Erläuterung sei das Schema des `limit` Parameters gegeben (aus Listing 6.1):

```
minimum: 0
type: integer
```

Mit der Flussgraphmethode aus dem vorherigen Kapitel werden damit folgende invalide Beispiele automatisch erzeugt:

```
"a_string"
null
true
false
{}
[]
-1
```

Der Parameter `limit` ist ein Query-Parameter, d.h. es können keine vom Schema abweichenden Typen anstelle von `integer` übergeben werden. Daher werden Typabweichungen bei der Beispielgenerierung deaktiviert, sodass als invalider Wert `limit=-1` bleibt. Abweichungen vom Typ sind nur beim Body eines Requests zugelassen.

Ähnlich verhält es sich beim Parameter `cursor`. Dieser wird vom Server zurückgeliefert, d.h. unterliegt keinen syntaktischen Einschränkungen. Dementsprechend ist das Schema für `cursor` lediglich:

```
type: string
```

Da `cursor` ebenfalls ein Query-Parameter ist, sind auch hier Typabweichungen ausgeschlossen. Daher können auch keine syntaktisch invaliden Werte erzeugt werden, da dies von der eigentlichen Serverimplementierung abhängen würde. Somit gibt es für `cursor` keine negativen Tests. Ein weiteres Beispiel sind die Serialisierungsmodifier wie `level` mit folgendem Schema:

```
type: string
enum:
  - deep
  - core
```

Hier ergibt sich als invalider Parameterwert jeder String, der nicht in der Aufzählung enthalten ist.

6.4.3 Valide Parameterwerte auslesen

Das Verfahren aus Kapitel 5 ließe sich genauso verwenden, um valide Werte für Parameter aus dem JSON-Schema zu erzeugen. Es gibt allerdings mehrere Probleme mit diesen automatisch erzeugten, vermeintlich validen Parameterwerten: Zunächst sind die Werte nur *syntaktisch* valide, aber nicht unbedingt auch *semantisch* valide. Als Beispiel sei dafür abermals der Parameter `cursor` gegeben. Hier ist jeder beliebige String ein syntaktisch korrekter Wert. Da der Wert aber implementierungsabhängig ist, sind nur wenige Strings auch tatsächlich gültige Cursor aus Sicht des Servers. Dies gilt für die meisten Parameter: Es ist nicht möglich, aus dem JSON-Schema automatisch valide Werte zu generieren. Daher müssen gültige Parameterwerte manuell definiert werden. Für einige Parameter erfordert dies sogar einen separaten Request, um einen gültigen Wert beim Server abzurufen. So auch für den `cursor` Parameter: Um einen validen Cursor zu erhalten, muss zunächst `GetAllAssetAdministrationShells` mit `limit=1` aufgerufen werden. Dann kann aus der Antwort des Servers ein gültiger Wert für `Cursor` ausgelesen werden. Dies geschieht vor der eigentlichen Testausführung während der Setup-Phase (siehe Unterabschnitt 6.3.1).

Ferner sind manche Parameter in Kombination zu betrachten, so z.B. `cursor` und `limit`. Wenn `cursor` vorhanden ist, dann liefert die Seitennummerierung andere Einträge. Dasselbe gilt für die Kombination von Serialisierungsmodifiern. Außerdem müssen bei positiven Tests die Effekte von gewählten, validen Parameterwerten überprüft werden, z.B. ob tatsächlich `limit` Einträge zurückgegeben werden. Da diese Überprüfungen ohnehin manuell zu implementieren sind, ist es effizienter, in diesem Zusammenhang auch die validen Parameterwerte manuell vorzugeben.

Abschließend könnten die automatisch generierten, vermeintlich validen Parameterwerte dennoch verwendet werden, um die Anzahl der Testfälle zu erhöhen. Allerdings wäre hierbei nicht bekannt, welche Antwort der Server liefern sollte. Die Testausführung würde lediglich länger dauern, ohne die Testabdeckung zu erhöhen. Für einige Operationen wäre es nützlich, um den Aufwand zu vermeiden, irrelevante Parameterwerte füllen zu müssen. Z.B. beim Anlegen neuer Verwaltungsschalen ist der Inhalt der Verwaltungsschale irrele-

vant, es wird lediglich irgendeine valide Verwaltungsschale als Body benötigt. Dies kann dann automatisch erzeugt werden.

6.4.4 Operationen filtern

In der OpenAPI-Beschreibung der Verwaltungsschale sind *alle* Operationen enthalten, die die Spezifikation definiert. Bei der Testausführung werden allerdings nicht alle Operationen ausgeführt, weil ein zu testendes System in der Regel nicht alle Operationen implementiert. Stattdessen wird ein Profil oder eine Service Specification gewählt, gegen die getestet wird. Das bedeutet, dass die entsprechende Untermenge an Operationen mit den dazugehörigen Testfällen ausgewählt wird. Nur für diese werden dann auch Testfälle erzeugt. Manche Operationen sind je nach Profil lediglich über Superpfade enthalten. Diese sind in der OpenAPI Spezifikation explizit formalisiert und als zusätzliche Operationen enthalten. Somit können auch Superpfade mit dem Verfahren getestet werden.

6.4.5 Parameterwerte einsetzen

Abschließend werden die Parameterwerte in die gemäß Profil ausgewählten Operationen eingesetzt. Für negative Testfälle wird für jeden Parameter und hierbei für jeden invaliden Parameterwert ein Testfall erzeugt. Dafür wird der jeweilige Parameter auf den invaliden Wert gesetzt. Alle anderen Parameter werden weggelassen bzw. auf valide Werte gesetzt.

Für positive Testfälle werden ausschließlich valide Werte verwendet. Diese werden in manuell definierten Kombinationen eingesetzt. Ein Vorteil der manuellen Vorgehensweise ist, dass die Bedeutung der Parameterwerte einfach getestet werden kann. Während bei invaliden Parameterwerten, d.h. bei negativen Tests, der Server lediglich einen Fehler zurückliefern soll, werden bei validen Parameterwerten, d.h. bei positiven Tests, tatsächlich Aktionen ausgeführt. Beispielsweise wird für den Parameterwert `limt=1` geprüft, dass der Server tatsächlich eine Liste mit genau einer Verwaltungsschale zurückliefert.

6.5 Positive Testfälle

Wie bereits erläutert, werden negative Testfälle automatisch erzeugt. Im Folgenden werden die positiven Testfälle vorgestellt, die für die einzelnen Operationen der ausgewählten Profile manuell definiert werden.

6.5.1 AAS Repository

Nachfolgend werden die Testfälle des Profils `https://admin-shell.io/aas/API-/3/0/AssetAdministrationShellRepositoryServiceSpecification/SSP-002` gelistet. Dieses Profil enthält alle Operationen, um lesend auf ein Verwaltungsschalen-Repository zuzugreifen. Tabelle 6.1 listet alle Operationen, die in diesem Profil enthalten sind.

GetAllAssetAdministrationShells

Die Operation `GetAllAssetAdministrationShells` dient dem Auslesen aller Verwaltungsschalen aus einem Repository. Folgende Testfälle sind definiert:

Tabelle 6.1: AAS Repository Read Profil (nach [8, Seite 147])

Operation	Endpunkt	Testfälle
AAS Repository API:		
GetAllAssetAdministrationShells	/shells	16
GetAssetAdministrationShellById	/shells/AAS-ID	4
AAS API by superpath:		
GetAllSubmodelReferences	/shells/AAS-ID/submodel-refs	4
GetAssetInformation	/shells/AAS-ID/asset-information	1
GetThumbnail	/shells/AAS-ID/asset-information/thumbnail	1
Submodel Repository API by superpath:		
GetSubmodelById	/shells/AAS-ID/submodels/SUBMOD-ID	15
Submodel API by superpath:		
GetAllSubmodelElements	.../submodel-elements	50
GetSubmodelElementByPath	.../submodel-elements/PATH	26
GetFileByPath	.../submodel-elements/PATH/attachment	1
Serialization API:		
GenerateSerializationByIds	/serialization	9
Description API:		
GetDescription	/description	1

- Die Operation wird ohne Parameter aufgerufen. Es wird lediglich überprüft, ob eine Liste von Verwaltungsschalen zurückgegeben wird.
- Es wird nach einer bestimmten Asset-ID gefiltert, d.h. der Parameter `assetId` wird übergeben. Es wird geprüft, dass eine Liste mit Verwaltungsschalen zurückgegeben wird, die alle der Asset-ID zugeordnet sind.
- Es wird nach einer bestimmten idShort gefiltert, d.h. der Parameter `idShort` wird übergeben. Es wird geprüft, dass eine Liste mit Verwaltungsschalen zurückgegeben wird, die alle der idShort zugeordnet sind.
- Es wird nach einer assetId gefiltert, die nicht existiert. Es wird geprüft, dass eine leere Liste zurückgegeben wird.
- Es wird nach einer idShort gefiltert, die nicht existiert. Es wird geprüft, dass eine leere Liste zurückgegeben wird.
- Genau eine Verwaltungsschale auslesen, d.h. mit `limit=1` aufrufen: Es wird geprüft ob eine Liste von Verwaltungsschalen mit genau einem Eintrag zurückgegeben wird.
- Es wird eine Abfrage mit Offset gemacht, d.h. der Parameter `cursor` wird gesetzt. Wenn das Repository nur eine Verwaltungsschale enthält, kann dieser Test nicht durchgeführt werden.

Einige Testfälle benötigen gültige Parameterwerte für `idShort` und `assetId`. Diese Werte werden vor der Testausführung aus dem Repository ausgelesen. Außerdem wird ein gültiger Wert für den Parameter `curosr` ausgelesen, indem ein Aufruf mit `limit=1` vor der Testausführung durchgeführt wird. Die genannten Testfälle werden zweimal durchlaufen: einmal ohne Content-Modifier und einmal mit dem Content-Modifier `$reference`.

GetAssetAdministrationShellById

Die Operation `GetAssetAdministrationShellById` liest eine Verwaltungsschale anhand ihrer ID aus. Daher gibt es neben den negativen Testfällen lediglich einen positiven Test. Dieser ruft die Operation mit einer gültigen ID auf und prüft, dass tatsächlich die Verwaltungsschale mit dieser ID zurückgeben wird. Das Auslesen einer nicht-existierenden Verwaltungsschale ist Teil der automatisch-generierten negativen Tests. Die genannten Testfälle werden zweimal durchlaufen: einmal ohne Content-Modifier und einmal mit dem Content-Modifier `$reference`.

GetAllSubmodelReferences

Die Operation liest alle Submodell-Referenzen einer Verwaltungsschale aus. Hier gibt es neben den negativen Tests zwei positive Testfälle:

- Ohne Parameter aufrufen: Es wird geprüft, dass eine Liste aller Referenzen zurückgeben wird.
- Genau eine Referenz auslesen, d.h. mit `limit=1` aufrufen. Es wird geprüft, ob eine Liste von Referenzen mit genau einem Eintrag zurückgegeben wird.
- Es wird eine Abfrage mit Offset gemacht, d.h. der Parameter `cursor` wird gesetzt. Wenn das Repository nur ein Submodell enthält, dann kann dieser Test nicht durchgeführt werden.

GetAssetInformation

Die Operation `GetAssetInformation` liest Asset-Informationen einer Verwaltungsschale aus. Hier gibt es einen positiven Testfall, der überprüft, dass tatsächlich gültige Asset-Informationen zurückgegeben werden.

GetThumbnail

Die Operation `GetThumbnail` liest das Thumbnail einer Verwaltungsschale aus. Hier gibt es einen positiven Testfall, der überprüft, dass tatsächlich ein Bild zurückgegeben wird. Wenn die Verwaltungsschale kein Thumbnail hat, kann dieser Test nicht ausgeführt werden.

GetSubmodelById

Die Operation `GetSubmodelById` liest ein Submodell einer Verwaltungsschale aus. Der einzige positive Test zu dieser Operation prüft, dass tatsächlich ein gültiges Submodell zurückgegeben wird. Die genannten Testfälle werden fünfmal durchlaufen: einmal ohne Content-Modifier sowie mit dem Content-Modifiern `$reference`, `$value`, `$metadata` sowie `$path`.

GetAllSubmodelElements

Die Operation `GetAllSubmodelElements` liest alle Submodell-Elemente eines Submodells aus. Folgende positive Testfälle sind definiert:

- Aufruf ohne Parameter: Es wird geprüft, dass eine Liste von Submodell-Elementen zurückgeliefert wird.

- Aufruf mit `level=core` bzw. `level=deep`: Es wird geprüft, dass eine Liste von Submodell-Elementen zurückgeliefert wird, bei der Elemente entsprechend dem `level` Parameter traversiert wurden.

- Aufruf mit `extent=withBlobValue` sowie `extent=withoutBlobValue`: Es wird geprüft, dass eine Liste von Submodell-Elementen zurückgeliefert wird, bei der Blob-Elemente einen `value` enthalten bzw. nicht enthalten.

- Genau ein Submodell-Element auslesen, d.h. mit `limit=1` aufrufen: Es wird geprüft, ob eine Liste von Submodell-Element mit genau einem Eintrag zurückgegeben wird.

- Es wird eine Abfrage mit Offset gemacht, d.h. der Parameter `cursor` wird gesetzt. Wenn das Submodell nur ein Submodell-Element enthält, dann kann dieser Test nicht durchgeführt werden.

Die genannten Testfälle werden fünfmal durchlaufen: einmal ohne Content-Modifier sowie mit dem Content-Modifiern `$reference`, `$value`, `$metadata` sowie `$path`.

GetSubmodelElementByPath

Die Operation `GetSubmodelElementByPath` liest zu einem Submodell ein bestimmtes Submodell-Element aus, dessen `idShort`-Pfad zu übergeben ist. Folgende positiven Testfälle sind definiert:

- Die Operation wird ohne Parameter aufgerufen.

- Aufruf mit `level=core` bzw. `level=deep`: Es wird geprüft, dass eine Liste von Submodell-Elementen zurückgeliefert wird, bei der Elemente entsprechend dem Parameter `level` traversiert wurden.

- Aufruf mit `extent=withBlobValue` sowie `extent=withoutBlobValue`: Es wird geprüft, dass eine Liste von Submodell-Elementen zurückgeliefert wird, bei der Blob-Elemente einen `value` enthalten bzw. nicht enthalten.

Die obigen Testfälle werden für die Kombination aus den 14 Typen von Submodell-Elementen sowie aller fünf Content-Modifier ausgeführt.

GetFileByPath

Die Operation `GetFileByPath` dient dazu, für Submodell-Element vom Type `File` den dazugehörigen Dateiinhalte auszulesen. Da diese Operation keine Parameter besitzt, gibt es hierfür lediglich einen Testfall, der die Operation ohne Parameter aufruft.

GenerateSerializationByIds

Die Operation `GenerateSerializationByIds` veranlasst den Server, den Inhalt zu serialisieren. Es sind vier Testfälle definiert:

- Ohne Parameter aufrufen. Es wird überprüft, dass eine korrekte Serialisierung zurückgegeben wird.
- Nach Asset-ID filtern. Es wird überprüft, dass eine korrekte Serialisierung zurückgegeben wird, die nur die gefilterten Werte enthält.
- Nach Submodell-ID filtern. Es wird überprüft, dass eine korrekte Serialisierung zurückgegeben wird, die nur die gefilterten Werte enthält.
- Mit `includeConceptDescriptions=true` aufrufen. Es wird überprüft, dass Concept-Descriptions in der Antwort enthalten sind.

GetDescription

Die Operation `GetDescription` liest die Liste der Profile aus, die vom Server unterstützt werden. Der einzige positive Testfall prüft, dass die Liste die Zeichenkette `https://admin-shell.io/aas/API/3/0/AssetAdministrationShellRepositoryServiceSpecification/SSP-002` enthält.

6.5.2 AAS Registry

Nachfolgend werden die Testfälle des Profils `https://admin-shell.io/aas/API/3/0/AssetAdministrationShellRegistryServiceSpecification/SSP-002` gelistet. Das Profil dient dazu lesend auf eine Verwaltungsschalen-Registry zuzugreifen. Tabelle 6.2 gibt eine Übersicht. Kern dieses Profils sind *Deskriptoren*. Ein Deskriptor assoziiert eine Verwaltungsschale mit einem dazugehörigen Server. Daher sind Informationen wie der Endpunkt des Servers (URL) und das Protokoll im Deskriptor enthalten.

Tabelle 6.2: AAS Registry Read Profil (nach [8, Seite 140])

Operation	Endpunkt	Testfälle
AAS Registry API:		
GetAllAssetAdministrationShellDescriptors	/shell-descriptors	9
GetAssetAdministrationShellDescriptorById	/shell-descriptors/ID	2
Submodel Registry API by superpath:		
GetAllSubmodelDescriptors	/shell-descriptors/ID/submodel-descriptors	4
GetSubmodelDescriptorById	/shell-descriptors/ID/submodel-descriptors/ID	2
Description API:		
GetDescription	/description	1

GetAllAssetAdministrationShellDescriptors

Diese Operation gibt alle Deskriptoren zurück, die in einer Registry abgelegt sind. Für die Operation sind folgende Testfälle definiert:

- Die Operation wird ohne Parameter aufgerufen
- Nach `assetId` filtern: Die Ergebnisse müssen alle die angeforderte `assetId` enthalten.
- Nach `assetType` filtern: Die Ergebnisse müssen alle den angeforderten `assetType` enthalten.
- Tests zur Seitennummerierung, d.h. die Parameter `limit` bzw. `cursor` werden gesetzt.

GetAssetAdministrationShellDescriptorById

Die Operation `GetAssetAdministrationShellDescriptorById` liefert für die ID einer Verwaltungsschale die dazugehörigen Deskriptoren. Die Operation hat nur einen Parameter, die ID der Verwaltungsschale. Es gibt daher nur einen Testfall, der die Operation mit einer gültigen ID aufruft und das Ergebnis überprüft.

GetAllSubmodelDescriptors

Diese Operation gibt alle Deskriptoren für Submodelle zurück, die in einer Submodell-Registry abgelegt sind. Die Operation hat neben den Seitennummerierungsparametern `limit` und `cursor` keine weiteren Parameter. Dementsprechend gibt es einen Test, der den Endpunkt ohne Parameter aufruft sowie weitere Tests, die die Seitennummerierung überprüfen.

GetSubmodelDescriptorById

Diese Operation liest für eine Submodell-ID den Deskriptor aus. Dieser Endpunkt hat keine Parameter, sodass nur ein positiver Test definiert ist, der den Endpunkt ohne Parameter aufruft.

GetDescription

Die Operation `GetDescription` liest die Liste der Profile aus, die vom Server unterstützt werden. Der einzige positive Testfall prüft, dass die Liste die Zeichenkette `https://admin-shell.io/aas/API/3/0/AssetAdministrationShellRegistryServiceSpecification/SSP-002` enthält.

6.5.3 Discovery Service

Nachfolgend werden die Testfälle des Profils `https://admin-shell.io/aas/API/3/0/DiscoveryServiceSpecification/SSP-002` gelistet. Das Profil dient dazu, lesend auf einen Discovery-Server zuzugreifen. Ein Discovery-Server hat die Aufgabe, für eine Asset-ID die entsprechenden Verwaltungsschalen-IDs zu ermitteln und umgekehrt. Tabelle 6.3 gibt eine Übersicht über die Operationen.

Tabelle 6.3: AAS Discovery Read Profil (nach [8, Seite 143])

Operation	Endpunkt	Testfälle
AAS Basic Discovery API:		
GetAllAssetAdministrationShellIdsByAssetLink	/lookup/shells	7
GetAllAssetLinksById	/lookup/shells/ID	2
Description API:		
GetDescription	/description	1

GetAllAssetAdministrationShellIdsByAssetLink

Diese Operation liefert alle gespeicherten Zuordnungen von Asset-ID zu Verwaltungsschalen-IDs. Damit ruft der erste positive Test die Operation ohne Parameter auf, um die gesamte Liste abzurufen. Zusätzlich kann diese Liste auf bestimmte Asset-IDs beschränkt werden. Daher ruft der zweite positive Test den Endpunkt auf, wobei nach einer Asset-ID gefiltert wird. Es wird überprüft, dass die Antwort tatsächlich nur die angegebenen Asset-IDs enthält.

GetAllAssetLinksById

Diese Operation liefert für eine gegebene Verwaltungsschalen-ID die Liste aller zugeordneten Asset-IDs. Damit wird gewissermaßen die Operation `GetAllAssetAdministrationShellIdsByAssetLink` umgekehrt. Da die Operation lediglich die Verwaltungsschalen-ID als Parameter erwartet, gibt es nur einen positiven Test, der diesen Parameter entsprechend setzt.

GetDescription

Die Operation `GetDescription` liest die Liste der Profile aus, die vom Server unterstützt werden. Der einzige positive Testfall prüft, dass die Liste die Zeichenkette `https://admin-shell.io/aas/API/3/0/DiscoveryServiceSpecification/SSP-002` enthält.

6.5.4 Concept Description Repository

Nachfolgend werden die Testfälle des Profils `https://admin-shell.io/aas/API-/3/0/ConceptDescriptionRepositoryServiceSpecification/SSP-002` gelistet. Dieses Profil enthält alle Operationen, um lesend auf ein Konzeptbeschreibungsrepository zuzugreifen. Tabelle 6.4 listet alle Operationen, die in diesem Profil enthalten sind.

GetAllConceptDescriptions

Diese Operation liefert alle gespeicherten Konzeptbeschreibungen. Es sind folgende positive Tests definiert:

- Die Operation wird ohne Parameter aufgerufen.
- Die Konzeptbeschreibungen werden nach `idShort` gefiltert. Es wird überprüft, dass das Ergebnis nur Einträge mit der angegebenen `idShort` enthält.

Tabelle 6.4: Concept Description Read Profil (nach [8, Seite 153])

Operation	Endpunkt	Testfälle
Concept Description Repository API:		
GetAllConceptDescriptions	/concept-descriptions	8
GetConceptDescriptionById	/concept-descriptions/ID	2
Serialization API:		
GenerateSerializationByIds	/serialization	9
Description API:		
GetDescription	/description	1

- Die Konzeptbeschreibungen werden nach `isCaseOf` gefiltert. Es wird überprüft, dass das Ergebnis nur Einträge mit der angegebenen `isCaseOf` enthält.
- Die Konzeptbeschreibungen werden nach `dataSpecificationRef` gefiltert. Es wird überprüft, dass das Ergebnis nur Einträge mit der angegebenen `dataSpecificationRef` enthält.
- Tests zur Seitennummerierung, d.h. die Parameter `limit` bzw. `cursor` werden gesetzt.

GetConceptDescriptionById

Diese Operation liefert die Konzeptbeschreibung für eine bestimmte ID. Da die Operation lediglich die ID als Parameter erwartet, ruft der einzige positive Test die Operation mit einer validen ID auf. Es wird überprüft, dass die richtige Konzeptbeschreibung zurückgegeben wird.

GenerateSerializationByIds

Die Operation `GenerateSerializationByIds` veranlasst den Server, den Inhalt zu serialisieren. Es sind vier Testfälle definiert:

- Ohne Parameter aufrufen: Es wird überprüft, dass eine korrekte Serialisierung zurückgegeben wird.
- Nach Asset-ID filtern: Es wird überprüft, dass eine korrekte Serialisierung zurückgegeben wird, die nur die gefilterten Werte enthält.
- Nach Submodell-ID filtern: Es wird überprüft, dass eine korrekte Serialisierung zurückgegeben wird, die nur die gefilterten Werte enthält.
- Mit `includeConceptDescriptions=true` aufrufen: Es wird überprüft, dass Concept-Descriptions in der Antwort enthalten sind.

GetDescription

Die Operation `GetDescription` liest die Liste der Profile aus, die vom Server unterstützt werden. Der einzige positive Testfall prüft, dass die Liste die Zeichenkette `https://admin-shell.io/aas/API/3/0/ConceptDescriptionRepositoryService-Specification/SSP-002` enthält.

6.6 Evaluation

Nachfolgend wird das Verfahren auf vier Server angewendet, um deren Konformität zu überprüfen und um die Eignung des Verfahrens zu evaluieren. Eine nähere Beschreibung der Server findet sich in Unterabschnitt 3.5.2. Für die Evaluation wurden in Kapitel 3 die folgenden vier Service-Spezifikationen ausgewählt, da sie 10 von 11 der Interfaces abdecken:

- Asset Administration Shell Repository Servicespezifikation
- Asset Administration Shell Registry Servicespezifikation
- Discovery Servicespezifikation
- ConceptDescription Repository Servicespezifikation

6.6.1 Initiale Testdaten

Für die Testausführung des AAS Repository Profils müssen eine bestimmte Menge von Verwaltungsschalen und Submodellen im Server vorhanden sein. Wenn dies nicht der Fall ist, können einige Tests nicht durchgeführt werden. Beispielsweise müssen beim Testen der Operationen `GetAllAssetAdministrationShells` und `GetAllSubmodelElements` mindestens zwei Verwaltungsschalen bzw. Submodell-Elemente vorhanden sein, um den Parameter `cursor` testen zu können. Hier wird mit `limit=1` ein `cursor` angefordert, was nur funktioniert, wenn mehr als ein Eintrag vorhanden ist. Für die Operation `GetFileByPath` muss ein Submodell-Element vom Typ `File` vorhanden sein. Ferner erfordert die Operation `GetThumbnail`, dass ein solches hinterlegt ist. Und um die Operation `GetSubmodelElementByPath` für alle möglichen Arten von Submodell-Elementen durchführen zu können, müssen ebensolche vorhanden sein.

Um diese Testdaten vorzuhalten, wird eine spezielle AASX-Datei bereitgestellt. Diese deckt die obigen Punkte ab und muss vor der Testausführung im Repository abgelegt werden. Da es sich bei dem zu testenden Profil um ein lesendes Profil handelt, muss dies durch den Entwickler geschehen. Bei einem schreibenden Profil können die Testdaten direkt geschrieben werden. Falls es nicht möglich ist, die AASX-Datei im Vorhinein zu laden, dann können nicht alle Testfälle geprüft werden. Das hat zur Folge, dass die Anzahl der ausgeführten Testfälle je nach Server variiert.

Für ein Concept-Description-Repository können initiale Testdaten ebenfalls mithilfe der AASX-Datei in den Server geladen werden. Ähnliches gilt auch für die Registry- und Discovery-Profile. Hier muss eine bestimmte Menge an Deskriptoren und Asset-IDs im Server vorhanden sein. Diese können allerdings nicht mittels AASX-Datei erstellt werden. Daher werden diese über die passenden Schreib-Operationen in die Server geladen. D.h. es werden vor der Testausführung zwei Shell-Deskriptoren

via `PostAssetAdministrationShellDescriptor` sowie zwei Submodell-Deskriptoren via `PostSubmodelDescriptor` in einen Registry-Server geladen. Ferner werden zwei Asset-IDs mithilfe der Operation `PostAllAssetLinksById` in einen Discovery-Server geladen.

6.6.2 Testausführung

Für alle Server stellen die Entwickler fertige Docker-Images [102] bereit, die für die Testausführung verwendet wurden. Die Orchestrierung lässt sich mithilfe von `compose.yml` Dateien beschreiben. Listing 6.2 zeigt ein Beispiel für den AASX Server. Mit dem Schlüsselwort `image` wird hier das Image ausgewählt (Zeile 4). Ferner wird in Zeile 5-8 der Port definiert, auf dem der Server lauschen soll. Dieser wird bei der Testausführung genutzt, um Requests zu senden. Mit dem Schlüsselwort `volume` werden die initialen Testdaten in den Server geladen (Zeile 9). Abschließend werden bei allen Servern Security-Mechanismen deaktiviert, wenn vorhanden (Zeile 11). Das Testen von Security-Mechanismen ist nicht Teil dieser Arbeit und würde die eigentliche Testausführung behindern.

Listing 6.2: Testaufbau für den AASX Server (`compose.yml`, aus [9])

```
1  services:
2   aasx-server:
3    image:
4     adminshellio/aasx-server-blazor-for-demo
5    ports:
6     - 5001:5001
7    environment:
8     - Kestrel__Endpoints__Http__Url=http://*:5001
9    volumes:
10    - ./aasxs:/AasxServerBlazor/aasxs
11   command: --no-security
```

6.6.3 AASX Server

Zunächst wurde der *AASX Server* in der Version 0.3.1 getestet. Für den AASX Server stellen die Entwickler fertige Docker-Images bereit, die für die Testausführung verwendet wurden. Die Ergebnisse der Testausführung zeigt Tabelle 6.5. Beim VWS-Repository-Profil besteht der AASX Server alle positiven Tests. Bei den negativen Tests erkennt der AASX Server beispielsweise nicht, wenn Werte nicht base64-kodiert sind. Auch invalide Werte für die Serialisierungsmodifier akzeptiert der AASX Server fälschlicherweise.

Beim VWS-Registry-Profil konnte ein Großteil der Tests nicht ausgeführt werden, weil es keine funktionierende Möglichkeit gibt, Deskriptoren zu laden. Dasselbe gilt für das Discovery-Profil. Hier konnten keine Asset-IDs in den Server geladen werden. Beim Repository für Konzeptbeschreibungen konnte ein Großteil der Testfälle erfolgreich abgeschlossen werden. Lediglich die Filterung z.B. nach `idShort` und wenige negative Tests schlugen fehl. Insgesamt ist der AASX Server somit für die Profile VWS- und Concept-Description-Repository produktiv einsetzbar.

Tabelle 6.5: Testergebnisse AASX Server

	Negative Tests	Positive Tests	Gesamt
VWS-Repository	77%	100%	93%
VWS-Registry	33%	58%	50%
CD-Repository	80%	80%	80%
Discovery	0%	14%	10%

6.6.4 BaSyx Java

Im Anschluss wurde das Eclipse BaSyx Java V2 SDK getestet, nachfolgend mit *BaSyx Java* abgekürzt. Dafür wurde das von den Entwicklern bereitgestellte Docker Image des Projektes in der Version 2.0.0 verwendet. Die Ergebnisse der Testausführung zeigt Tabelle 6.6.

Tabelle 6.6: Testergebnisse BaSyx Java

	Negative Tests	Positive Tests	Gesamt
VWS-Repository	43%	16%	24%
VWS-Registry	0%	92%	61%
CD-Repository	40%	47%	45%
Discovery	33%	86%	70%

Beim Repository-Profil lösen einige negative Tests Antworten mit einem Statuscode 500 aus. Der Statuscode 500 wird vom Server zurückgegeben, wenn ein schwerwiegender Fehler im Server auftritt, z.B., wenn eine Exception geworfen wird. Im konkreten Fall scheitert der BaSyx Java Server, weil er versucht, einen nicht base64 kodierten Wert zu dekodieren. Außerdem versucht der Server, negative Werte für `limit` zu verarbeiten. Beide Fehlerfälle werden im Server nicht ordnungsgemäß behandelt und somit wird Statuscode 500 zurückgegeben.

Bei den positiven Testfällen funktionieren 16%. Dies hat diverse Gründe. Im Falle der Operation `GetAllAssetAdministrationShells` ist die Filterung nach Asset-ID bzw. idShort nicht korrekt implementiert. Wird der Content-Modifier `$reference` mit übergeben, wird dieser fälschlicherweise als Asset-ID der Operation `GetAssetAdministrationShellById` interpretiert. Allgemein ist dieser Content-Modifier falsch implementiert, was zu diversen Fehlern auch in anderen Operationen führt. Ferner ist die Operation `GetThumbnail` inkorrekt implementiert und gibt keinen Thumbnail zurück.

Außerdem ist die Operation `GetSubmodelById` inkorrekt implementiert, da das angefragte Submodell nicht gefunden wird. Dies hat auch zur Folge, dass die Operation `GetAllSubmodelElements` nicht funktioniert. Eine weitere Folge ist, dass Operationen wie `GetSubmodelElementByPath` nicht getestet werden können Um auf ein Submodellelement zugreifen zu können, muss dessen idShort bekannt sein. Da die Operation `GetAllSubmodelElements` nicht funktioniert, können auch die idShorts nicht ausgelesen werden und alle nachfolgenden Testfälle nicht ausgeführt werden.

Beim Registry Profil lassen sich bei den negativen Testfällen die gleichen Fehler aufdecken, wie beim Repository Profil. Von den positiven Testfällen läuft ein Groß-

teil ohne Fehler durch, lediglich die Filterung nach Asset-Typ bei der Operation `GetAllAssetAdministrationShellDescriptors` schlägt fehl. Beim Concept-Description-Repository funktioniert die Operation `GenerateSerializationByIds` nicht und liefert unabhängig von den gesetzten Parametern einen 400 Fehlercode. Bei der Operation `GetAllConceptDescriptions` ist die Filterung nach `isCaseOf` fehlerhaft implementiert. Die wenigsten Fehler finden sich beim Discovery Profil. Negative Tests finden ähnliche Fehler wie beim Repository Profil. Auch hier ist die Filterung bei der Operation `GetAllAssetAdministrationShellIdsByAssetLink` fehlerhaft implementiert. Bei der Filterung nach einer nicht existierenden Asset-ID liefert der Server fälschlicherweise alle Asset IDs statt einer leeren Liste. Insgesamt ist der BaSyx Java Server eher nicht für den produktiven Einsatz zu empfehlen.

6.6.5 BaSyx Python

Anschließend wurde das Eclipse BaSyx Python SDK betrachtet, nachfolgend mit *BaSyx Python* abgekürzt. Verwendet wurde Version 1.0.0 des SDKs. Für BaSyx Python stellen die Entwickler kein Docker-Image bereit, lediglich eine Anleitung zum Erstellen eines solchen. Zum Ausführen der Tests wurde das Image anhand der Anleitung erstellt. BaSyx Python implementiert lediglich das VWS Repository Profil. Die Ergebnisse der Testausführung zeigt Tabelle 6.7.

Tabelle 6.7: Testergebnisse BaSyx Python

	Negative Tests	Positive Tests	Gesamt
VWS-Repository	43%	23%	29%
VWS-Registry	*nicht implementiert*		
CD-Repository	*nicht implementiert*		
Discovery	*nicht implementiert*		

Von den negativen Tests schlagen 57% fehl, weil sie einen 502 Fehler auslösen. Ein Großteil von diesen Tests löst eine Exception im Server aus. Für die Operation `GenerateSerializationByIds` wird ein 501 Fehler erzeugt, weil diese Operation nicht implementiert ist.

Von den positiven Tests sind 23% erfolgreich durchgelaufen. Dies hat verschiedene Ursachen. Für die Operation `GetAllAssetAdministrationShells` funktioniert die Filterung nach einer bestimmten Asset-ID nicht. Die Operation `GetThumbnail` ist nicht implementiert und liefert einen entsprechenden 501 Fehlercode, genauso wie die Operationen `GenerateSerializationByIds` und `GetDescription`. Die übrigen positiven Tests liefern einen 502 Fehlercode aus, was auf eine nicht behandelte Exception im Server schließen lässt. Einige Tests ließen sich aufgrund dieser Fehler nicht durchführen, insbesondere die Tests für die Operation `GetSubmodelElementByPath`. Der BaSyx Python Server ist somit für den produktiven Einsatz nicht zu empfehlen.

6.6.6 FA³ST

Als nächstes wurden die Fraunhofer Advanced Asset Administration Shell Tools (FA³ST) in der Version 1.2.0 getestet. Die Entwickler stellen ein fertiges Docker-Image bereit, das

für die Testausführung verwendet wurde. Die Ergebnisse zeigt Tabelle 6.8.

Tabelle 6.8: Testergebnisse FA³ST

	Negative Tests	Positive Tests	Gesamt
VWS Repository	94%	95%	95%
VWS Registry	17%	100%	72%
Konzeptbeschr. Repository	100%	73%	80%
Discovery	67%	86%	80%

Beim VWS Repository Profil werden alle negativen Tests korrekterweise abgewiesen bis auf Tests mit base64-kodierten Werten. Bei den positiven Tests, bei der Operation `GenerateSerializationByIds` ist die Filterung fehlerhaft implementiert. Die übrigen fehlgeschlagenen positiven Tests lassen sich auf fehlerhafte Implementierung der Serialization-Modifier zurückführen. So wird z.B. für die Operation `GetAllSubmodelElements` der Parameter `level` abgelehnt, für die Operation `GetSubmodelElementByPath` der Content-Modifier `$path` für einige Submodell-Elemente fälschlicherweise zugelassen.

Beim Registry Profil lassen sich sehr viele fehlgeschlagene, negative Tests verzeichnen. Diese lassen sich auf zwei Klassen aufteilen: Zum einen werden base64-kodierte Daten nicht geprüft, zum anderen der `limit` Parameter. Von den positiven Tests schlägt kein Test fehl. Ähnlich schlagen beim Konzeptbeschreibungsprofil keine negativen Tests fehl. Bei den positiven Tests gibt es ähnliche Fehler wie bereits beim Repository Profil. Das Discovery-Profil liefert vergleichbare Fehler. Insgesamt ist der FA³ST Server für den produktiven Einsatz eingeschränkt zu empfehlen. Es sollte allerdings genau abgewägt werden, welche Operationen benötigt werden, da nicht alle komplett fehlerfrei implementiert sind.

6.6.7 Diskussion der Testergebnisse

Konformität der Server

Gemäß Unterabschnitt 2.3.2 lässt sich aus den Ergebnissen die Genauigkeit berechnen. Diese entspricht dem Anteil bestandener Testfälle an der Gesamtanzahl des Testfälle. Einen Vergleich der Server bzgl. dieser Metrik zeigt Tabelle 6.9.

Tabelle 6.9: Vergleich der Konformität der getesteten Server (Genauigkeit)

	AASX Server	BaSyx Java	BaSyx Python	FA³ST
VWS Repository	93%	24%	29%	95%
VWS Registry	50%	61%	-	72%
Konzeptbeschr. Repository	80%	45%	-	80%
Discovery	10%	70%	-	80%

Die umfangreichste Testsuite findet sich für das VWS Repository Profil. Hier schneiden der AASX Server und FA³ST am besten ab. Die Server des BaSyx Projektes können aufgrund diverser Fehlimplementierungen wie nicht überprüfte Eingabewerte sowie fehlender Implementierungen nicht überzeugen. Für das VWS Registry Profil findet sich die besten

Ergebnisse bei FA³ST, die übrigen Implementierungen scheitern insbesondere an den negativen Testfällen. Beim Konzeptbeschreibungsprofil liefern der AASX Server sowie FA³ST gute Ergebnisse, die BaSyx Java Implementierung scheitert an einigen positiven sowie negativen Testfällen. Beim Discovery Profil liefern BaSyx Java sowie FA³ST gute Ergebnisse. Der AASX Server scheitert bei den meisten Tests, insbesondere, weil vorab keine initialen Testdaten geladen werden konnten.

Insgesamt zeigt sich auch, dass zwei spezielle Aspekte Probleme bei der Implementierung bereiten: das Filtern z.B. nach Asset-ID und Serialisierungs-Modifier. Das liegt daran, dass beide Aspekte in der Spezifikation zu ungenau spezifiziert sind und damit zu große Freiheitsgrade bei der Implementierung erlauben. Wenn eine Implementierung für den Produktiveinsatz ausgewählt werden soll, dann sind am ehesten die Implementierungen des FA³ST Projektes zu empfehlen, die bei allen vier getesteten Profilen gute Ergebnisse erzielen.

Eignung des Verfahrens

Es zeigt sich, dass das vorgestellte Verfahren in der Lage ist, die Konformität von Verwaltungsschalen-Servern zu messen und damit sicherzustellen. Grundlegende Features jeder REST-API sind abgebildet, da grundlegend alle CRUD-Operationen ausführbar sind. Aufbauend darauf werden auch die Besonderheiten von reaktiven Verwaltungsschalen berücksichtigt. Superpfade können getestet werden, indem diese in der OpenAPI-Beschreibung separat formalisiert werden. Dadurch sind auch alle Profile der Verwaltungsschale testbar, auch jene, die Interfaces über Superpfade einbinden.

Es zeigt sich, dass die Aussagekraft der geprüften Lese-Profile von den initialen Testdaten abhängt. Wenn beispielsweise keine Asset-IDs in einen Discovery Server geladen werden können, so ist dieser nicht sinnvoll testbar. Dies ist beim AASX Server der Fall. Außerdem wird die Testausführung durch die Superpfade eingeschränkt. Wenn z.B. beim VWS Repository keine Verwaltungsschalen auflistbar sind, können in Folge auch keine Submodelle oder Submodellelemente getestet werden.

Weitere Features wie idShort-Pfade und Serialisierungsmodifier werden ebenfalls unterstützt, müssen aber manuell definiert werden. Obwohl die manuelle Definition zunächst aufwändiger erscheint, wurde es für das vorgestellte Verfahren dennoch gewählt. Zum einen ist dies die flexibelste Lösung, zum anderen zeigt sich, dass die Formalisierung sehr komplex wird und mit OpenAPI teilweise gar nicht möglich ist. Aufgrund der Konstruktion ist sichergestellt, dass durch die negativen Testfälle für jede Operation und für alle Parameter systematisch invalide Wertebereiche abgedeckt werden. Außerdem sind die Testfälle insoweit korrekt, als dass sie erfüllbar sind. Dies wurde durch die Evaluation anhand verschiedener Server nachgewiesen.

6.7 Fazit

Server für die Verwaltungsschale nehmen im industriellen Kontext eine wichtige Rolle ein, da sie es ermöglichen, *dynamische* Daten über ein Asset bereitzustellen. Dies ermöglicht eine Vielzahl von Anwendungen entlang des gesamten Lebenszyklus. In diesem Kapitel wurde ein Verfahren vorgestellt, um Verwaltungsschalen-Server auf Konformität zu überprüfen. Kern des Verfahrens bildet die Formalisierung der Operationen von reaktiven Verwal-

tungsschalen mithilfe einer OpenAPI-Beschreibung. Diese ermöglicht das automatisierte Erzeugen von negativen Testfällen. Für positive Testfälle dagegen ist manueller Aufwand erforderlich, weil valide Parameterwerte nicht automatisch abgeleitet werden können. Dies liegt daran, dass die OpenAPI-Beschreibung Parameter nur auf syntaktischer Ebene formalisiert, die gewählten Parameterwerte aber auch auf *semantischer* Ebene korrekt sein müssen.

Die erzeugten Testfälle ermöglichen es, eine Vielzahl von nicht-konformen Verhalten nachzuweisen bzw. konformes Verhalten zu bestätigen. Zu nicht-konformem Verhalten zählen fehlende Fehlerbehandlungen, wenn z.B. negative Werte übergeben werden. Weiter zählen dazu auch nicht implementierte Operationen. Es zeigt sich aber auch, dass die Spezifikation an ein einigen Stellen zu große Freiheitsgrade zulässt, die dann bei der Implementierung zu Fehlern führen können.

Damit lassen sich Forschungsfragen 2-5 im Kontext von Servern beantworten:

FF2: Welche Besonderheiten zeichnen die Verwaltungsschale aus softwaretechnischer Sicht aus? Welche speziellen Anforderungen an ein System zum Überprüfen der Konformität der bzgl. Interoperabilität relevanten Teile des Verwaltungsschalen-Ökosystems ergeben sich daraus?

Zunächst gelten alle Besonderheiten, die bereits für passive Verwaltungsschalen definiert wurden, auch für reaktive Verwaltungsschalen, da diese dasselbe Metamodell verwenden. Die API für die Verwaltungsschale ist sehr umfangreich. Sie enthält diverse Operationen und Parameter. Darüber hinaus ist sie komplex, weil viele Parameter, insbesondere die Serialisierungsmodifier, in Wechselwirkung zueinander stehen. Ein Testsystem muss diese Komplexität durch die sinnvolle Auswahl von Testfällen und Parameterwerten berücksichtigen, damit aussagekräftige Testfälle entstehen.

FF3: Wie kann ein Testsystem für Softwarekomponenten der Verwaltungsschale umgesetzt werden?

Im Rahmen dieser Arbeit wurde ein Verfahren vorgestellt, dass auf der OpenAPI-Beschreibung der Verwaltungsschale basiert. Daraus werden, soweit möglich, Testfälle automatisiert erzeugt. Diese Testfälle lassen sich grundsätzlich in negative und positive Testfälle unterscheiden. Negative Testfälle überprüfen unerwartetes Verhalten, indem einzelne Parameter auf invalide Werte gesetzt werden. Diese werden automatisiert erzeugt. Positive Testfälle dagegen überprüfen zu erwartendes Verhalten. Sie werden manuell definiert, sodass sie systematisch alle Operationen und die jeweiligen Parameter abdecken.

FF4: Welche Fehler können mit dem Testsystem in realen Softwarekomponenten für die Verwaltungsschale aufgedeckt werden? Welche Grenzen gibt es?

Mithilfe des vorgestellten Verfahrens lassen sich diverse Fehler in realen Implementie-

rungen der Verwaltungsschale aufdecken. Dazu zählt die fehlende Überprüfung von Eingabewerten sowie die fehlende oder fehlerhafte Implementierung von Operationen. Es zeigt sich, dass viele Testfälle das Vorhandensein bestimmter Daten im Server erfordern. Dies kann dazu führen, dass bei lesenden Profilen Testfälle nicht ausgeführt werden können.

FF5: Welche Vor- und Nachteile hat die automatische Ableitung von Konformitätstests aus formalen Spezifikationen?

Im Rahmen dieser Arbeit wurde die weit verbreitete OpenAPI-Beschreibung als formale Spezifikation gewählt. Für negative Tests, also Tests, die invalide Parameterwerte an den Server übergeben, haben OpenAPI-Beschreibungen einige Vorteile bei der automatischen Testfallerzeugung. Zunächst reduzieren sie den benötigten manuellen Aufwand drastisch, da alle Requests automatisch abgeleitet werden können. Darüber hinaus ermöglichen sie ein *systematisches* Testen aller möglichen Klassen von invaliden Parameterwerten. Dies wäre beim manuellen Testen schwierig sicherzustellen. Außerdem können die Antworten des Servers direkt auf syntaktische Korrektheit geprüft werden, indem gegen die hinterlegten JSON-Schemata geprüft wird.

Für positive Tests sind OpenAPI-Beschreibungen nur eingeschränkt einsetzbar. Das liegt insbesondere daran, dass OpenAPI es nur eingeschränkt ermöglicht, Beziehungen *zwischen* den einzelnen Operationen zu beschreiben. Dadurch lassen sich schlecht valide Parameterwerte automatisch ableiten. Diese müssen oft vom Server zunächst ausgelesen werden, weil eine rein syntaktisch korrekte Erzeugung nicht ausreichen würde. Darüber hinaus bietet OpenAPI nur eine syntaktische Überprüfung der Korrektheit der Antwort. Eine Überprüfung der Seiteneffekte einer Operation muss manuell definiert werden. Daher verwendet der vorgestellte Ansatz einen semi-automatischen Ansatz zur Erzeugung der positiven Tests.

7 Formale Spezifikationen in Softwarelebenszyklen von Verwaltungsschalen

Die Entwicklung von Software durchläuft unabhängig von der Domäne einen Lebenszyklus. Dieser erstreckt sich von der initialen Anforderungsanalyse und -spezifikation über die eigentliche Implementierung bis zur Auslieferung und Wartung der Software. In diesem Kapitel wird der Lebenszyklus von Software aus dem Ökosystem der Verwaltungsschale im Allgemeinen und den *AAS Test Engines* im Speziellen betrachtet. Besonderer Fokus liegt dabei auf der Rolle von formalen Spezifikationen im Softwarelebenszyklus.

7.1 Der allgemeine Softwarelebenszyklus

Abbildung 7.1 zeigt die Phasen des Softwarelebenszyklus [103]. Dieser erstreckt sich von der anfänglichen Anforderungsanalyse und -spezifikation über die eigentliche Entwicklung bis hin zum finalen Testen und der Verwendung im produktiven Einsatz.

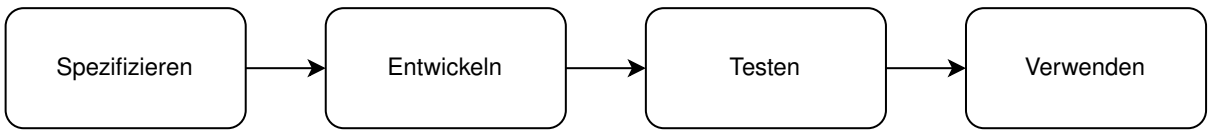


Abbildung 7.1: Lebenszyklus einer Softwareanwendung

Die einzelnen Phasen des Softwarelebenszyklus sind:

- **Spezifizieren:** In dieser Phase werden die Anforderungen des Kunden und Endanwenders analysiert und spezifiziert. In der Regel werden diese in Form eines Lasten- und Pflichtenheftes festgehalten.

- **Entwickeln:** Die Entwicklung gliedert sich in mehrere Unterphasen. Zunächst werden projektbezogene Ressourcen geplant, wie z.B. Budgets und Mitarbeiter. Im Anschluss wird die Architektur der Software geplant, z.B. Schnittstellen und zu verwendende Technologien. Abschließend wird die Software implementiert, d.h. in Form von Quellcode umgesetzt.

- **Testen:** In der Testphase wird die korrekte Funktionalität der Software sichergestellt. Abschluss dieser Phase bilden Abnahmetests, in der der Kunde die gewünschten Funktionen überprüft. Abschnitt 2.3 gibt einen Überblick über Softwaretests, insbesondere über Konformitätstests.

- **Verwenden:** Nach Bestehen der Abnahmetests wird die Software ausgeliefert und gewartet. In dieser Phase werden Fehlerbehebungen und neue Features nachgeliefert.

Zu beachten ist, dass diese Aufteilung einige Aspekte vereinfacht darstellt. So werden, je nach Domäne, einzelne Phasen weiter untergliedert. Auch können Phasen mehrfach oder iterativ durchlaufen werden, z.B. bei der agilen Entwicklung [104].

7.2 Formale Spezifikationen im Ökosystem der Verwaltungsschale

Formale Spezifikationen werden als Teil der Spezifikation der Verwaltungsschale veröffentlicht. Dabei handelt es sich zum einen um das JSON-Schema zur Beschreibung des Metamodells, zum anderen um die OpenAPI-Beschreibung für die Operationen der reaktiven Verwaltungsschale. Kapitel 2 erläutert diese Formate näher. Die beiden Formalisierungen finden im Lebenszyklus vieler Softwarebibliotheken und -anwendungen aus dem Ökosystem der Verwaltungsschale Verwendung.

In der **Spezifikationsphase** werden die oben genannten, formalen Beschreibungen durch Mitglieder der Arbeitsgruppe *WG Open Technology* der IDTA erstellt und gepflegt [61]. In dieser Phase bieten formale Spezifikationen diverse Vorteile: Zunächst sind sie strikter und im Gegensatz zu natürlicher Sprache eindeutig. Dies verhindert Missverständnisse. So wurde z.B. während des Spezifikationsprozesses der Verwaltungsschale das Wort „oder“ von einigen Beteiligten als logisches XOR, von anderen als logisches OR interpretiert. Dies resultierte schließlich zu Fehlern in einer Implementierung der Verwaltungsschale [105]. Außerdem können formale Spezifikationen darauf überprüft werden, dass sie widerspruchsfrei sind. Wenn Widersprüche erst nach Veröffentlichung einer menschenlesbaren Spezifikation auffallen, führt dies zu unnötigen Zeitverzögerungen und erfordert ggf. sogar eine Neuveröffentlichung der Spezifikation.

Die veröffentlichten, formalen Spezifikationen werden bei der **Entwicklung** diverser Software aus dem Ökosystem der Verwaltungsschale eingesetzt. Diese Spezifikationen können z.B. bei der automatisierte Generierung von Code oder Codegerüsten verwendet werden, was für die Verwaltungsschale in [67] erprobt wurde. Dabei wurden Verwaltungsschalenimplementierungen in mehreren Programmiersprachen generiert. Auch die Entwickler des AASX Servers nutzen die OpenAPI-Beschreibung, um den Code für den Server zu generieren [106]. Weiterhin helfen Beispielwerte und -instanzen die Spezifikation verständlich zu machen. Diese können mithilfe der formalen Spezifikation direkt auf Korrektheit überprüft werden.

Vor der finalen Verwendung der Software muss diese gründlichen **Tests** unterzogen werden, um Fehler aufzudecken und zu beheben. Auch hierfür können formale Spezifikationen herangezogen werden. So können z.B. basierend auf der formalen Spezifikation Testeingaben generiert werden, die dann von der Software verarbeitet werden. Außerdem können die Ausgaben der Software gegen die formale Spezifikation validiert werden. Die automatische Erstellung von Testfällen aus den formalen Beschreibungen der Verwaltungsschale ist zentraler Gegenstand dieser Arbeit und wurden in den Kapiteln 4 bis 6 bereits ausführlich erläutert. Die Konformitätstests der Verwaltungsschale werden mithilfe der AAS Test Engines [78] durchgeführt.

Nach Auslieferung, während der **Verwendung** der Software, können formale Spezifikationen weitere Prozesse automatisieren. Beispielsweise können formale Spezifikationen zur Dokumentation verwendet werden, was auch im Rahmen der Verwaltungsschale erprobt wird [107]. Dabei wird aus einer formalen Spezifikation des Metamodells eine UML-Darstellung von Klassen und eine tabellarische Auflistung der Attribute erzeugt. Die Auflistung wird mit manuell geschriebenen Erläuterungen angereichert, um somit die finale, menschenlesbare Spezifikation zu erhalten.

7.3 Softwarelebenszyklus der AAS Test Engines

Die *AAS Test Engines* sind das von der IDTA bereitgestellte Tool zum Ausführen von Konformitätstests für die Verwaltungsschale. Das Tool nutzt die in dieser Arbeit vorgestellten Methoden, um automatisiert Konformitätstests aus den formalen Spezifikationen abzuleiten. Auch die AAS Test Engines durchlaufen den Softwarelebenszyklus. Die konkreten Schritte sind in Abbildung 7.2 dargestellt. Zunächst veröffentlichen die Arbeitsgruppen der IDTA die Spezifikation in menschenlesbarer Form (im PDF Format). Ergänzend werden die formalen Spezifikationen, also das JSON-Schema und die OpenAPI-Beschreibung, veröffentlicht. Basierend auf den Inhalten der menschenlesbaren Form der Spezifikation implementiert die *Test Engine Developer Group* die Constraints. Zusammen mit dem JSON-Schema ermöglichen die implementierten Constraints das Überprüfen von Verwaltungsschalen auf Konformität zur Spezifikation, wie in Kapitel 4 beschrieben. Außerdem ermöglichen die Constraints das Erzeugen von Testdaten zum Testen von SDKs auf Konformität, wie in Kapitel 5 beschrieben. Ferner nutzt die *Test Engine Developer Group* die Spezifikation, um positive Testfälle für reaktive Verwaltungsschalen zu implementieren. Aus der OpenAPI-Beschreibung werden die negativen Testfälle erzeugt und mit den positiven Testfällen kombiniert, um reaktive Verwaltungsschalen auf Konformität zu überprüfen, wie in Kapitel 6 beschrieben. Zuletzt werden die drei Grundbausteine kombiniert, und in Form der AAS Test Engines bereitgestellt.

Die AAS Test Engines müssen eine Reihe von Akzeptanztests vor der finalen Veröffentlichung bestehen. Das hat zwei wesentliche Gründe: Zum einen werden dadurch Fehler in den AAS Test Engines aufgedeckt. Zum anderen ist sichergestellt, dass die Testfälle auch von realen Implementierungen erfüllt werden können und nicht z.B. unerfüllbare Testfälle existieren. Für die Akzeptanztests werden (Referenz-) Implementierungen aus der Verwaltungsschalen-Community gewählt. Tabelle 7.1 zeigt die verwendeten Referenzen für Version 3.0 der Spezifikation der Verwaltungsschale.

Eines der Ziele bei der Verwendung von formalen Spezifikationen ist die Reduktion des Aufwands bei der Implementierung. In diesem Abschnitt soll die Reduktion qualitativ untersucht werden. Dafür werden zunächst die Schritte bei der Implementierung der AAS Test Engines betrachtet:

- Die Überprüfung von passiven Verwaltungsschalen umfasst zwei wesentliche Schritte, die implementiert werden müssen: Zunächst muss das Metamodell der Verwaltungsschale implementiert werden. Dies geschieht üblicherweise in Form von Klassen der verwendeten Programmiersprachen. Darüber hinaus müssen die Constraints implementiert werden.

- Zur Überprüfung von Verwaltungsschalen-SDKs müssen Testinstanzen erzeugt wer-

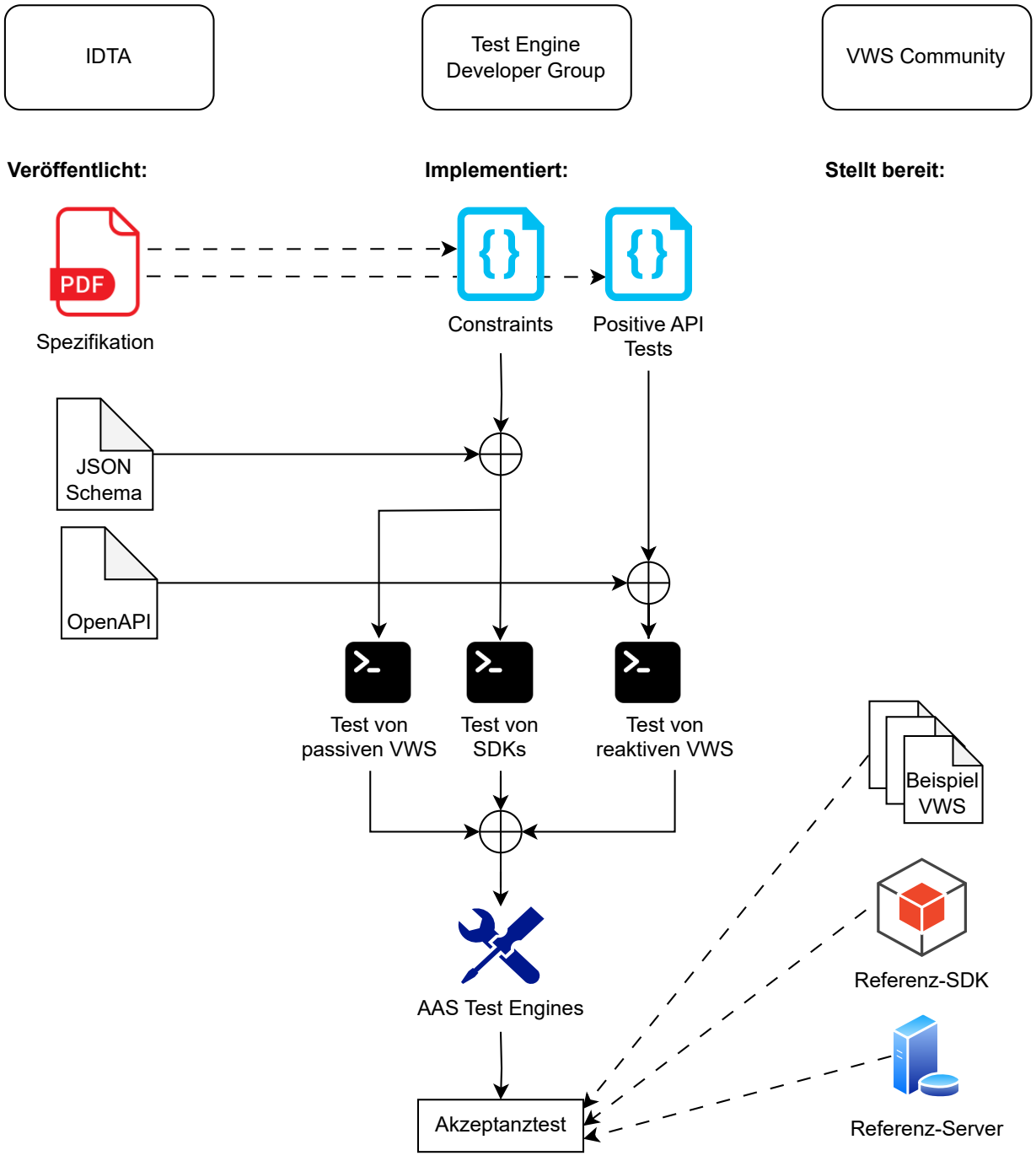


Abbildung 7.2: Softwarelebenszyklus der *AAS Test Engines*

den. Dabei sind sowohl valide als auch invalide Testinstanzen nötig, die idealerweise alle Aspekte des Metamodells und der Constraints abdecken.

- Für den Test von reaktiven Verwaltungsschalen ist die Implementierung von positiven und negativen API Tests nötig. Diese decken idealerweise alle Operationen aller Profile ab.
- Neben den eigentlichen Testfunktionen müssen noch weitere Aspekte implementiert werden. Ein wesentliches Feature ist dabei der Testreport, der gleichermaßen nutzer-

Tabelle 7.1: Verwendete Referenzen aus der VWS-Community für die Akzeptanztests der AAS Test Engines zur Überprüfung von Konformität zur Version 3.0 der Spezifikation der Verwaltungsschale

Baustein	Akzeptanzkriterium	Referenz
Test von passiven VWS	Alle validen VWS akzeptieren, alle invaliden abweisen	Beispielinstanzen des Projektes *AAS Core Works*
Test von VWS-SDKs	SDK besteht alle Testfälle	AAS Core Python SDK
Test von reaktiven VWS	Server besteht alle Testfälle für alle Profile	AASX Server

freundlich als auch aussagekräftig sein soll.

Es ist möglich, alle diese Schritte ohne die Verwendung von formalen Spezifikationen zu implementieren. Tatsächlich wurden einige der Schritte bei der Implementierung der AAS Test Engines durch die Nutzung formaler Spezifikationen beschleunigt. Tabelle 7.2 zeigt den Vergleich der geschätzten Aufwände. Dabei wird davon ausgegangen, dass die Tools zum automatisierten Erzeugen von Testfällen aus formalen Spezifikationen, wie sie im Rahmen dieser Arbeit entwickelt wurden, bereits existieren und nicht Teil der Aufwände sind.

Tabelle 7.2: Geschätzte Aufwände bei der Implementierung der AAS Test Engines in Relation zum Gesamtaufwand: Vergleich des Aufwandes einer manuellen Implementierung mit der Implementierung mit formalen Spezifikationen

Aufgabe	Aufwand manuell	form. Spezifikationen
Implementierung Metamodell	10%	0%
Implementierung Constraints	15%	15%
Generierung von Testinstanzen	15%	0%
Positive API Tests	30%	30%
Negative API Tests	20%	0%
Testreport u. weitere Features	10%	20%
Gesamt	100%	60%

Die Implementierung des Metamodells entfällt komplett, da es durch eine Überprüfung gegen das JSON-Schema der Verwaltungsschale ersetzt werden kann. Die Constraints hingegen müssen weiterhin in beiden Varianten manuell implementiert werden. Das liegt im Wesentlichen daran, dass das offizielle JSON-Schema der Verwaltungsschale die Constraints nicht formalisiert, sondern lediglich das Metamodell. Außerdem lässt sich nur ein Teil der Constraints (bedingt und wertbezogen) formalisieren, die übrigen müssen ohnehin separat implementiert werden. Die manuelle Generierung von Testinstanzen kann komplett durch die automatische Generierung aus dem JSON-Schema ersetzt werden. Neben der Zeitersparnis stellt dies auch eine systematische Abdeckung aller Freiheitsgrade des Metamodells sicher. Beim Testen von reaktiven Verwaltungsschalen müssen die positiven Testfälle auch

bei der Verwendung der OpenAPI-Beschreibung der Verwaltungsschale manuell definiert werden. Eine Zeitersparnis findet sich daher nur bei den negativen Tests, die vollständig automatisiert aus der OpenAPI-Beschreibung abgeleitet werden können. Abschließend muss *mehr* Zeit in den Testreport investiert werden, da Fehlermeldungen aus der Validierung mithilfe von formalen Spezifikationen oft generisch sind. Sie müssen daher auf besser lesbare Fehlermeldungen abgebildet werden und in die Termini aus der Verwaltungsschale übersetzt werden. Insgesamt ergibt sich dadurch eine geschätzte Aufwandsersparnis um 40% gegenüber einer komplett manuellen Implementierung ohne formale Spezifikationen.

Auch beim Update auf neuere Versionen der Spezifikation reduzieren formale Spezifikationen den Aufwand. Beispielsweise wurde im Juni 2024 die Version 3.0.2 der Verwaltungsschale veröffentlicht [8]. Eine der großen Änderungen dieser Version ist das Entfernen des `level` Parameters aus allen `/$metadata`-Endpunkten. Dieser Parameter wurde entsprechend aus der OpenAPI-Beschreibung entfernt, sodass in den AAS Test Engines lediglich die positiven Testfälle angepasst werden mussten [9].

7.4 Fazit

In diesem Kapitel wurde die Verwendung von formalen Spezifikationen entlang des Softwarelebenszyklus anhand einiger Beispiele betrachtet. Dabei werden einige Erkenntnisse aus der Entwicklung der *AAS Test Engines*, dem Testsystem für Verwaltungsschalen, beschrieben. Durch die Verwendung von formalen Spezifikationen (OpenAPI, JSON-Schema) können einige Schritte der Implementierung beschleunigt werden. Wesentliches Qualitätsmerkmal bildet ein abschließender Akzeptanztest mit unabhängigen Referenzen aus der Verwaltungsschalen-Community.

FF5: Welche Vor- und Nachteile hat die automatische Ableitung von Konformitätstests aus formalen Spezifikationen?

Eine Analyse des Aufwands bei der Implementierung der *AAS Test Engines* zeigt, dass formale Spezifikationen die Implementierung um bis zu 40% beschleunigen können. Auch in anderen Phasen der Softwareentwicklung bieten formale Spezifikationen Vorteile: Sie sind eindeutig und zeigen so bereits während der Spezifikation Ungenauigkeiten auf und ermöglichen die Generierung von Code und Dokumentation. Allerdings erfordern sie eine gewisse Einarbeitungszeit. Auch beim Testreport erhöht sich der Aufwand, weil Fehlermeldungen zusätzlich in die Anwendungsdomäne übersetzt werden müssen.

8 Zusammenfassung

Interoperabilität ist eine von drei Kernzielen, die im „Leitbild 2030 für Industrie 4.0“ [16] definiert sind. Ein wesentliche Voraussetzung für Interoperabilität ist Konformität zur Spezifikation. Um Konformität sicherzustellen, haben sich Konformitätstests etabliert, d.h. das zu überprüfende System wird mit einem Satz an vordefinierten Testfällen untersucht. Wenn all diese Testfälle positiv ausfallen, wird das System als konform angesehen.

Für die Verwaltungsschale fehlten bisher solche Konformitätstests [3, 4, 8, 17]. Im Rahmen dieser Arbeit wurde daher ein Testsystem vorgestellt, das in der Lage ist, Konformitätstests für Implementierungen der Verwaltungsschale durchzuführen. Parallel zur Definition der eigentlichen Konformitätstests wurde die Frage untersucht, inwieweit formale Spezifikationen das Erstellen von Konformitätstests unterstützen und beschleunigen können. Wesentlich hierfür war das Konzept, dass die Konformitätstests nicht *manuell* definiert werden, sondern *automatisiert* aus formalen Spezifikationen generiert werden. Als formale Spezifikationen wurden dabei JSON-Schema [18] (zur Beschreibung des Metamodells der Verwaltungsschale) und Open API [19] (zur Beschreibung von reaktiven Verwaltungsschalen) gewählt. Diese werden als Teil der menschenlesbaren Spezifikation der Verwaltungsschale veröffentlicht.

Anschließend wurde zunächst das Ökosystem rund um die Verwaltungsschale analysiert. Dadurch wurden die zentralen Softwarekomponenten identifiziert, die wesentlich verantwortlich sind für die Interoperabilität im Ökosystem der Verwaltungsschale. Basierend auf dieser Analyse wurden drei dieser Komponenten ausgewählt, deren Konformität im Folgenden durch Konformitätstests sichergestellt werden sollte. Dabei handelt es sich um passive Verwaltungsschalen, Verwaltungsschalen-SDKs sowie reaktive Verwaltungsschalen.

Zur Überprüfung der Konformität von passiven Verwaltungsschalen werden diese zunächst gegen das JSON-Schema der Verwaltungsschale validiert. Es zeigt sich, dass die meisten Constraints formalisiert werden konnten. Die übrigen wurden programmatisch überprüft. Zur Überprüfung der Konformität von Verwaltungsschalen-SDKs wird ebenfalls das JSON-Schema herangezogen. Das Schema dient dabei als Grundlage zur Erzeugung von Testdaten. Diese Testdaten enthalten valide Instanzen von Verwaltungsschalen, die vom zu überprüfenden SDK akzeptiert werden müssen, sowie invalide Verwaltungsschalen, die vom SDK abgewiesen werden müssen. Die Korrektheit des Verfahrens wurde experimentell nachgewiesen. Es zeigt sich, dass neben der Korrektheit auch die *Vollständigkeit* der Testdaten eine wesentliche Rolle spielt, damit alle Aspekte des Metamodells abgedeckt werden. Abschließend wurde ein Verfahren vorgestellt, um reaktive Verwaltungsschalen basierend auf der OpenAPI-Beschreibung auf Konformität zu überprüfen. Dabei werden negative Testfälle automatisch erzeugt mithilfe desselben Verfahrens wie für SDKs. Positive Testfälle werden semi-automatisch erzeugt, da valide Parameterwerte manuell ausgelesen werden müssen und nicht generiert werden können

Durch die Verwendung von formalen Spezifikationen bei allen drei Verfahren konnte der manuelle Aufwand um etwa 40% gesenkt werden. Darüber hinaus stellt die automatische Generierung von Testfällen sicher, dass die Testfälle vollständig sind, in dem Sinne, dass alle

Aspekte des Metamodells bzw. der API abgeprüft werden. Es zeigen sich allerdings auch Nachteile wie die teils erschwerte Lesbarkeit der Testergebnisse sowie nötige Anpassungen, um z.B. die Constraints des Metamodells abbilden zu können.

Insgesamt wurden im Rahmen dieser Arbeit 3 SDKs und 4 Serverimplementierungen auf Konformität überprüft. Bei den SDKs ist die Konformität bereits recht hoch ($78\% \leq A_b \leq 99\%$). Bei den Serverimplementierungen bestehen allerdings teilweise noch hohe Defizite bei der Konformität ($49\% \leq A_b \leq 100\%$). Die im Rahmen dieser Arbeit vorgestellten Konformitätstests werden den Entwicklern im Softwareökosystem der Verwaltungsschale einen hilfreichen Leitfaden bieten, um die Konformität in Zukunft weiter zu erhöhen und schlussendlich die geforderte Interoperabilität zu erreichen [9].

8.1 Beantwortung der Forschungsfragen

8.1.1 Forschungsfrage 1

FF1: Wie ist das Softwareökosystem rund um die Verwaltungsschale aufgebaut? Welche Abhängigkeiten gibt es und welche grundlegenden Komponenten ergeben sich daraus, deren Konformität sicherzustellen ist?

Grundlage des Ökosystems der Verwaltungsschale bilden passive Verwaltungsschalen, die als Datei serialisiert werden. Somit bilden diese die erste Komponente, deren Konformität sicherzustellen ist. Weiterhin zeigt sich, dass nahezu jede Softwarekomponente im Ökosystem auf ein SDK zurückgreift. Diese implementieren Grundfunktionen, insbesondere das Serialisieren und Deserialisieren von passiven Verwaltungsschalen und bilden somit die zweite zu überprüfende Komponente. Ferner stellen Server reaktive Verwaltungsschalen bereit. Diese bilden folglich die dritte Komponente im Ökosystem, deren Konformität sicherzustellen ist. Zusätzlich wurden im Rahmen dieser Arbeit Serverframeworks und Clientbibliotheken als weitere Komponenten identifiziert. Allerdings sind diese bisher sehr wenig verbreitet, sodass im Rahmen dieser Arbeit Server direkt getestet wurden. Auch sollten Verwaltungsschalenapplikationen auf Konformität geprüft werden. Dies ist aus zwei Gründen außerhalb des Rahmens dieser Arbeit: Zunächst verwenden Applikationen in der Regel intern ein SDK, die bereits als zu testenden Komponente identifiziert wurden. Außerdem ist der Testaufbau für Applikationen stark von der Art der Applikation abhängig (z.B. grafische oder Konsolenanwendung). Daher ist es nicht möglich, dies in einem generischen Testaufbau abzubilden und muss von den Entwicklern der Anwendung selbst übernommen werden.

8.1.2 Forschungsfrage 2

FF2: Welche Besonderheiten zeichnen die Verwaltungsschale aus softwaretechnischer Sicht aus? Welche speziellen Anforderungen an ein System zum Überprüfen der Konformität der bzgl. Interoperabilität relevanten Teile des Verwaltungsschalen-Ökosystems ergeben sich daraus?

Grundlage der Spezifikation der Verwaltungsschale bildet das Metamodell und die Constraints [3]. Das Metamodell ist verglichen mit anderen Softwarekomponenten sehr komplex. Es besteht aus diversen Klassen, die zudem rekursive Abhängigkeiten aufweisen. Darüber hinaus müssen die Constraints betrachtet werden. Im Rahmen dieser Arbeit wurden die Constraints in vier Kategorien gruppiert. *Wertbezogenen* und *bedingte Constraints* lassen sich gut formalisieren, *komplexe Constraints* dagegen erfordern die Implementierung mithilfe einer vollwertigen Programmiersprache, *nicht prüfbare Constraints* lassen sich nicht implementieren. Weiter definiert die Spezifikation nicht nur *ein* Serialisierungsformat - wie üblich - sondern drei: XML, JSON sowie AASX [4]. Diese müssen alle vom Testsystem berücksichtigt werden.

Während die oben genannten Besonderheiten für passive Verwaltungsschalen gelten, werden bei reaktiven Verwaltungsschalen noch zusätzliche Besonderheiten deutlich: Die API für die Verwaltungsschale ist sehr umfangreich. Sie enthält diverse Operationen und Parameter. Darüber hinaus ist sie komplex, weil viele Parameter, insbesondere die Serialisierungsmodifier, in Wechselwirkung zueinander stehen. Ein Testsystem muss diese Komplexität durch die sinnvolle Auswahl von Testfällen und Parameterwerten berücksichtigen, damit aussagekräftige Testfälle entstehen.

Während sich obige Besonderheiten durch die inhaltliche Natur der Verwaltungsschale ergeben, kommen noch weitere Besonderheiten aufgrund des Spezifikationsprozesses hinzu: So muss ein Testsystem adaptierbar sein, um mit den häufigen Releases der Spezifikation mitzuhalten. Dabei müssen dennoch Nutzerfreundlichkeit und Qualität gewahrt bleiben.

8.1.3 Forschungsfrage 3

FF3: Wie kann ein Testsystem für Softwarekomponenten der Verwaltungsschale umgesetzt werden?

Im Rahmen dieser Arbeit wurden Methoden zur Überprüfung der Konformität von drei Softwarekomponenten vorgestellt: passive Verwaltungsschalen, SDKs und reaktive Verwaltungsschalen. Dabei stand die Verwendung von formalen Spezifikationen im Vordergrund, nämlich von JSON-Schema [18, 32] und OpenAPI [19]. Basierend auf diesen formalen Spezifikationen wurden drei grundlegende Bausteine vorgestellt (Abbildung 8.1).

Der erste Baustein überprüft ein JSON-Dokument auf Konformität zu einem JSON-Schema. Dieser Baustein wird im Kontext von Tests passiver Verwaltungsschalen verwendet, um Dateien auf Konformität zur Spezifikation zu überprüfen. In gleicher Weise wird

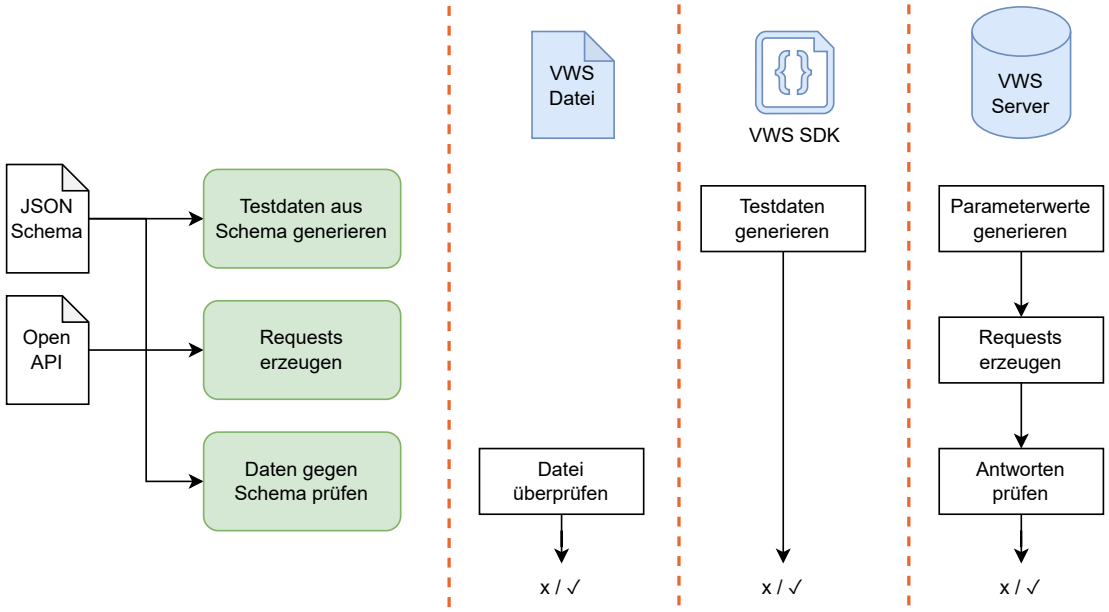


Abbildung 8.1: Softwarebausteine zur Überprüfung von Konformität

dieser Baustein verwendet, um die Antworten vom einem Server für reaktive Verwaltungsschalen auf Konformität zu überprüfen.

Der zweite Baustein dient dem Erzeugen von Testdaten basierend auf einem JSON-Schema. Im Rahmen dieser Arbeit wurde dafür ein neuartiger Ansatz vorgestellt. Dieser basiert auf der Idee, alle möglichen Entscheidungen bei der Instanziierung von Verwaltungsschalen in einem Flussgraph abzubilden. Durch systematisches Durchlaufen dieses Flussgraphen werden dann Testdaten erzeugt. Der zweite Baustein wird maßgeblich beim Test von Verwaltungsschalen-SDKs verwendet, um negative und positive Testdaten zu erzeugen, die dann vom SDK angenommen bzw. abgelehnt werden müssen.

Beim dritten Baustein handelt es sich um das Erzeugen von Testrequests aus einer OpenAPI-Beschreibung. Hierbei werden negative Testrequests vollautomatisch erzeugt. Für positive Testrequests sind teilweise manuelle Erweiterungen nötig. Dieser dritte Baustein wird maßgeblich beim Testen von reaktiven Verwaltungsschalen eingesetzt.

8.1.4 Forschungsfrage 4

FF4: Welche Fehler können mit dem Testsystem in realen Softwarekomponenten für die Verwaltungsschale aufgedeckt werden? Welche Grenzen gibt es?

Der vorgestellte Ansatz kann jegliche Abweichung vom Metamodell erkennen. Dafür muss Metamodell vollständig und eindeutig beschrieben sein. Mögliche Abweichungen umfassen beispielsweise fehlende Attribute oder fehlerhaft instanziierte Kindklassen. Da die Formalisierung auf Ebene der Serialisierung (JSON) und nicht auf Ebene der Klassenhierarchie erfolgt, können Fehler in der Serialisierung besser erkannt werden. Dies umfasst beispielsweise das Attribut `modelType`, das bei Ansätzen auf Ebene der Klassenhierarchie ggf. nicht überprüft wird. Darüber hinaus erkennt das Verfahren, wenn Constraints

nicht eingehalten werden. Einige Constraints können allerdings generell nicht geprüft werden, was jedoch auch mit anderen Verfahren nicht möglich wäre. Dazu zählt beispielsweise Constraint AASd-012, der die inhaltliche Gleichheit von mehrsprachigen Einträgen in einer MultiLanguageProperty fordert. Dies lässt sich nicht deterministisch überprüfen.

Das Verfahren kann grundsätzlich eine Vielzahl von syntaktischen Fehlern in SDKs finden. Dazu zählen z.B. fehlende oder falsch gesetzte Attribute. Fehler, die auf semantischer Ebene anzusiedeln sind, können nicht gefunden werden. Dazu zählen z.B. nicht-auflösbare Referenzen.

Mithilfe des vorgestellten Verfahrens lassen sich diverse Fehler in realen Implementierungen der Verwaltungsschale aufdecken. Dazu zählt die fehlende Überprüfung von Eingabewerten sowie die fehlende oder fehlerhafte Implementierung von Operationen. Es zeigt sich, dass viele Testfälle das Vorhandensein bestimmter Daten im Server erfordern. Dies kann dazu führen, dass bei lesenden Profilen Testfälle nicht ausgeführt werden können.

8.1.5 Forschungsfrage 5

FF5: Welche Vor- und Nachteile hat die automatische Ableitung von Konformitätstests aus formalen Spezifikationen?

Bei der Überprüfung von passiven Verwaltungsschalen ermöglicht die Nutzung des bereitgestellten JSON-Schemas eine sehr schnelle Implementierung eines Validierungstools. Allerdings müssen die Constraints separat formalisiert werden. Ferner kann das JSON-Schema nur auf die JSON Serialisierung der Verwaltungsschale angewandt werden. Für XML und AASX Serialisierungen sind zusätzliche Konvertierungen nötig. Ein Nachteil bei der Nutzung von JSON-Schema sind teilweise schwer verständliche Fehlermeldungen, insbesondere im Rahmen von Constraints. Bei großen Instanzen kann zudem die Performance der Validierung zu gering sein.

Im Rahmen der Überprüfung von Verwaltungsschalen-SDKs hat die Verwendung von JSON-Schema für die Generierung von Testdaten einige Vorteile: So können sehr viele und aussagekräftige Beispielinstanzen in kurzer Zeit erzeugt werden. Mithilfe des in dieser Arbeit vorgestellten Abdeckungsmaßes konnte sogar die Vollständigkeit dieser Testdaten gezeigt werden. Ein Nachteil der erzeugten Testdaten ist, dass sie nicht realitätsnah wirken, sondern rein syntaktische Aspekte abdecken. Dies kann die Verständlichkeit für menschliche Tester erschweren.

Im Rahmen dieser Arbeit wurde die weit verbreitete OpenAPI-Beschreibung als formale Spezifikation gewählt. Für negative Tests, also Tests, die invalide Parameterwerte an den Server übergeben, haben OpenAPI-Beschreibungen einige Vorteile bei der automatischen Testfallerzeugung. Zunächst reduzieren sie den benötigten manuellen Aufwand drastisch, da alle Requests automatisch abgeleitet werden können. Darüber hinaus ermöglichen sie ein *systematisches* Testen aller möglichen Klassen von invaliden Parameterwerten. Dies wäre beim manuellen Testen schwer sicherzustellen. Außerdem können die Antworten des Servers direkt auf syntaktische Korrektheit geprüft werden, indem gegen die hinterlegten JSON-Schemata geprüft wird.

Für positive Tests sind OpenAPI-Beschreibungen nur eingeschränkt einsetzbar. Das liegt insbesondere daran, dass OpenAPI es nur eingeschränkt ermöglicht, Beziehungen *zwischen*

den einzelnen Operationen zu beschreiben. Dadurch lassen sich schlecht valide Parameterwerte automatisch ableiten. Diese müssen oft vom Server zunächst ausgelesen werden, weil eine rein syntaktisch korrekte Erzeugung nicht ausreichen würde. Darüber hinaus bietet OpenAPI nur eine syntaktische Überprüfung der Korrektheit der Antwort. Eine Überprüfung der Seiteneffekte einer Operation muss manuell definiert werden. Daher verwendet der vorgestellte Ansatz einen semi-automatischen Ansatz zur Erzeugung der positiven Tests.

8.2 Ausblick

8.2.1 Verbesserung der Verfahren

An einigen Stellen der vorgestellten Methoden fallen noch manuelle Aufwände an, die weiter reduziert werden können. Bei den vorgestellten Konformitätstests für passive Verwaltungsschalen müssen die Formate XML und AASX zunächst nach JSON transformiert werden. Dieser Schritt ist manuell zu implementieren. Hierfür könnten Methoden der Modelltransformation [108] eingesetzt werden.

Im Kontext der vorgestellten Konformitätstests für Server fallen bei positiven Testfällen manuelle Aufwände an. Dies resultiert zum einen in zeitlichen Verzögerungen bei der Entwicklung der Testfälle. Zum anderen können auch Fehler in der Art auftreten, dass der Parameterraum nicht systematisch abgeprüft wird und Testfälle vergessen werden. Daher sollten auch hier mehr positive Testfälle automatisch generiert werden. Ein Ansatz dafür wird in [24] erprobt: Es werden Zusammenhänge zwischen den einzelnen Operationen mithilfe von OpenAPI Links [109] beschrieben. Ein Link definiert, wie Werte aus den Antworten einer Operation in Requests für eine andere Operation wiederverwendet werden können. Z.B. kann mit `GetAllAssetAdministrationShells` eine Liste aller Verwaltungsschalen ausgelesen werden. Jede Verwaltungsschale aus dieser Liste besitzt eine Asset-ID, die dann zum Auslesen von Submodellen dieser Verwaltungsschale verwendet werden. kann.

Ferner hat sich gezeigt, dass die Validierung gegen formale Spezifikationen weniger performant ist als eine manuell implementierte Variante. Ein zweiteiliger Ansatz kann die Vorteile aus beiden Welten vereinen: Der Code wird automatisch aus der formalen Spezifikation generiert und im Anschluss manuell verbessert. Dies vereinfacht auch spezielle Anpassungen wie Fehlermeldungen und kann somit helfen, die Verständlichkeit der Testberichte zu erhöhen.

8.2.2 Test von weiteren Komponenten

Im Rahmen dieser Arbeit wurden alle Interfaces der Verwaltungsschale betrachtet, bis auf das wenig verbreitete AASX File Server Interface. Dabei wurden bisher lediglich die Operationen betrachtet, die *lesend* auf einen Verwaltungsschalen-Server zugreifen. In Zukunft ist es sinnvoll, auch die *schreibenden* Operationen zu testen. Dafür kann das vorgestellte Verfahren im Wesentlichen unverändert weiter verwendet werden. Lediglich die positiven Testfälle müssen manuell definiert werden. Ferner können Methoden entwickelt werden, um Applikationen für Endnutzer wie den AASX Package Explorer [10] zu überprüfen. Mit den Konformitätstests, die im Rahmen dieser Arbeit entwickelt wurden, wurde bereits eine

Grundlage hierfür geschaffen. Im wesentlichen muss das Konzept des Testadapter erweitert werden, um die Testinstanzen in einer Applikation über das GUI direkt zu laden.

Abschließend hat sich diese Arbeit auf *syntaktische* Interoperabilität beschränkt. Wenn diese etabliert ist, dann folgt die Sicherstellung die *semantischen* Interoperabilität, die insbesondere durch die Verwendung von Teilmodellen [64] hergestellt werden soll. In dieser Arbeit wurde ein neuartiges Verfahren vorstellt, dass basierend auf einem JSON-Schema Testdaten erzeugt. Dieses Verfahren wurde auch auf Teilmodelle angewandt, indem die Templates mithilfe von JSON-Schema formalisiert wurden. Im Rahmen dieser Arbeit wurde dieser Ansatz anhand der Teilmodelle *Contact Information* und *Digital Nameplate* erfolgreich erprobt. Die Anwendung für weitere Teilmodelle ist der nächste logische Schritt.

8.2.3 Anwendung der Verfahren in anderen Domänen

Die vorgestellten Verfahren können auch für andere Anwendungsfälle als die Verwaltungsschale verwendet werden. Das Generieren von Testdaten aus Schemata wird auch in anderen Domänen erprobt [110, 111]. So ist die im Rahmen dieser Arbeit entwickelte Softwarebibliothek `Fences` in der Lage, neben JSON-Schemata auch XML Instanzen für XSD Schemata und Zeichenketten für Grammatiken zu erstellen [112]. Dies kann in diversen Anwendungsfällen verwendet werden. Ferner hat sich OpenAPI für die Beschreibung von REST APIs etabliert [38]. Somit lässt sich das vorgestellte Verfahren prinzipiell auf jede API anwenden, für die auch eine OpenAPI-Beschreibung existiert.

Diese Arbeit leistet einen entscheidenen Beitrag zur Sicherstellung der Konformität von Verwaltungsschalen. Dafür wurden neuartige Methoden entwickelt, in deren Zentrum formale Spezifikationen genutzt werden, um automatisch Konformitätstests abzuleiten. Dies ermöglicht es, mit geringem manuellen Aufwand qualitativ hochwertige Konformitätstests zu erzeugen und mit der hohen Geschwindigkeit der Entwicklungen im Umfeld der Verwaltungsschale mitzuhalten. Begleitend zu dieser Arbeit wurden die AAS Test Engines entwickelt, die als offizielles Konformitätstestsystem für die Verwaltungsschale eine entscheidende Basis für Interoperabilität liefern [9]. Die AAS Test Engines wurden im Rahmen dieser Arbeit eingesetzt, um die Konformität von real existierenden Softwarekomponenten zu evaluieren. Die Ergebnisse geben den Entwicklern wertvolle Hinweise, um die Konformität ihrer Software zu steigern und damit die Interoperabilität im Ökosystem der Verwaltungsschale zu erhöhen. Interoperabilität ist eines der zentralen Forderungen der Vision 2030 und eines der zentralen Versprechen der Verwaltungsschale. Somit liefern die im Rahmen dieser Arbeit vorgestellten Methoden und Tools einen wichtigen Beitrag zur Verbreitung und zum Erfolg der Verwaltungsschale.

Literaturverzeichnis

[1] H. Kubicek, A. Breiter, und J. Jarke, ,,Daten, Metadaten, Interoperabilität," *Handbuch Digitalisierung in Staat und Verwaltung*, S. 1–13, 2019.

[2] I. Graessler und J. Hentze, ,,The new V-Model of VDI 2206 and its validation," *at-Automatisierungstechnik*, Vol. 68, Nr. 5, S. 312–324, 2020.

[3] Industrial Digital Twin Association, *Specification of the Asset Administration Shell, Part 1: Metamodell*, März 2023.

[4] Industrial Digital Twin Association, *Specification of the Asset Administration Shell, Part 5: Package File Format (AASX)*, März 2023.

[5] B. Otto und T. Kleinert, ,,Test-Framework zur Qualitätssicherung von Verwaltungsschalen-SDKs," *Automation Kongress*, 2023.

[6] Industrial Digital Twin Association, *Submodel for Contact Information (Version 1.0)*, 2022.

[7] Industrial Digital Twin Association, *Digital Nameplate for industrial equipment (Version 2.0)*, 2022.

[8] Industrial Digital Twin Association, *Specification of the Asset Administration Shell, Part 2: Application Programming Interfaces*, Juni 2023.

[9] B. Otto und T. Kleinert, ,,Test Engines for the Asset Administration Shell," *at-Automatisierungstechnik*, Vol. 73, Nr. 2, S. 145–158, 2025.

[10] Eclipse AASX Package Explorer. https://github.com/eclipse-aaspe/package-explorer. Abgerufen am 14.01.2025.

[11] Igor Garmaev et. al. AAS Manager. https://github.com/rwth-iat/aas_manager. Abgerufen am 24.01.2025.

[12] 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)*. IEEE, 2022, S. 269–274.

[13] B. Otto und T. Kleinert, ,,Fences: Systematic Sample Generation for JSON Schemas using Boolean Algebra and Flow Graphs," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, 2024, S. 66–75.

[14] AAS Submodel Templates. https://industrialdigitaltwin.org/content-hub/teilmodelle. Abgerufen am 26.01.2025.

[15] B. Otto und T. Kleinert, ,,Automatisierte Generierung von Beispielinstanzen für Verwaltungsschalen Submodel Templates," in *Automation Kongress*, 2024.

[16] Bundesministerium für Wirtschaft und Energie, *Leitbild 2030 für Industrie 4.0 - Digitale Ökosysteme global gestalten*, Mai 2019.

[17] Industrial Digital Twin Association, *Specification of the Asset Administration Shell, Part 3a: Data Specification - IEC 61360*, März 2023.

[18] A JSON Media Type for Describing the Structure and Meaning of JSON Documents. https://datatracker.ietf.org/doc/html/draft-zyp-json-schema-01. Abgerufen am 16.01.2024.

[19] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, und R. Ratovsky. (2021) OpenAPI Specification v3.1.0.

[20] DIN SPEC 91345, *Reference Architecture Model Industrie 4.0 (RAMI4.0)*, 2016.

[21] International Organization for Standardization, ,,ISO/IEC 29119: Software Testing," 2013.

[22] 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, Nr. 8, S. 1978–2001, 2013.

[23] B. Otto und 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*. IEEE, 2022.

[24] B. Otto, K. Meinecke, und T. Kleinert, ,,Model-Based Test Case Generation for Compliance Checking of Reactive Asset Administration Shells," *KommA 2022 - 13. Jahreskolloquium Kommunikation in der Automation*, 2022.

[25] *Space systems — Definition of the Technology Readiness Levels (TRLs) and their criteria description*, International Organization for Standardization Standard ISO 16 290:2013, 2013, https://www.iso.org/standard/56064.html.

[26] A. v. Lamsweerde, ,,Formal specification: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, S. 147–159.

[27] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause *et al.*, ,,Using formal specifications to support testing," *ACM Computing Surveys (CSUR)*, Vol. 41, Nr. 2, S. 1–76, 2009.

[28] J. P. Bowen, ,,Z: A formal specification notation," in *Software Specification Methods: An Overview Using a Case Study*. Springer, 2001, S. 3–19.

[29] M. Zhu und R. R. Brooks, ,,Comparison of petri net and finite state machine discrete event control of distributed surveillance network," *International Journal of Distributed Sensor Networks*, Vol. 5, Nr. 5, S. 480–501, 2009.

[30] S. S. Gao, C. M. Sperberg-McQueen, und H. Thompson, ,,W3C XML schema definition language (XSD) 1.1 part 1: Structures," 2012.

[31] ECMA-404, ,,The JSON data interchange syntax," 2017.

[32] JSON Schema: A Media Type for Describing JSON Documents. https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01. Abgerufen am 16.01.2024.

[33] The Official YAML Website. https://yaml.org. Abgerufen am 26.01.2025.

[34] M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, und S. Scherzinger, ,,An empirical study on the "usage of not" in real-world json schema documents," in *Conceptual Modeling: 40th International Conference, ER 2021, Virtual Event, October 18–21, 2021, Proceedings 40*. Springer, 2021, S. 102–112.

[35] P. Bryan, K. Zyp, und M. Nottingham, ,,JavaScript object notation (JSON) pointer," Tech. Rep., 2013.

[36] Understanding JSON Schema: Recursion. https://json-schema.org/understanding-json-schema/structuring#recursion. Abgerufen am 02.03.2024.

[37] R. T. Fielding und J. Reschke, ,,Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," RFC 7231, Juni 2014, Abgerufen am 01.04.2025. https://www.rfc-editor.org/info/rfc7231

[38] Swagger Hub. https://app.swaggerhub.com. Abgerufen am 14.01.2025.

[39] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, und T. Berners-Lee, ,,Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Juni 1999, Abgerufen am 01.04.2025. https://www.rfc-editor.org/info/rfc2616

[40] R. T. Fielding und J. Reschke, ,,HTTP Semantics," RFC 9110, Juni 2022, Abgerufen am 01.04.2025. https://www.rfc-editor.org/rfc/rfc9110

[41] R. T. Fielding und J. Reschke, ,,HTTP/1.1," RFC 9112, Juni 2022. https://www.rfc-editor.org/rfc/rfc9112

[42] A. Arcuri, M. Z. Iqbal, und L. Briand, ,,Random testing: Theoretical results and practical implications," *IEEE transactions on Software Engineering*, Vol. 38, Nr. 2, S. 258–277, 2011.

[43] P. Godefroid, N. Klarlund, und K. Sen, ,,DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, S. 213–223.

[44] A. Fioraldi, D. Maier, H. Eißfeldt, und M. Heuse, ,,{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[45] M. Harman und P. McMinn, ,,A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, Vol. 36, Nr. 2, S. 226–247, 2009.

[46] W. Afzal, R. Torkar, und R. Feldt, ,,A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, Vol. 51, Nr. 6, S. 957–976, 2009.

[47] C. Nie und H. Leung, ,,A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, Vol. 43, Nr. 2, S. 1–29, 2011.

[48] R. Tzoref-Brill, ,,Advances in combinatorial testing," *Advances in Computers*, Vol. 112, S. 79–134, 2019.

[49] X. Li, R. Gao, W. E. Wong, C. Yang, und D. Li, ,,Applying combinatorial testing in industrial settings," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2016, S. 53–60.

[50] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, und B. M. Horowitz, ,,Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, 1999, S. 285–294.

[51] A. C. Dias Neto, R. Subramanyan, M. Vieira, und G. H. Travassos, ,,A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, 2007, S. 31–36.

[52] G. Fink und M. Bishop, ,,Property-based testing: A new approach to testing for assurance," *ACM SIGSOFT Software Engineering Notes*, Vol. 22, Nr. 4, S. 74–80, 1997.

[53] D. R. MacIver und Hatfield-Dodds, ,,Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, Vol. 4, Nr. 43, S. 1891, 2019.

[54] D. M. Powers, ,,Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[55] International Electrotechnical Commission, ,,Asset Administration Shell for Industrial Applications - Part 1: Asset Administration Shell Structure," https://webstore.iec.ch/publication/65628, 2023, IEC 63278-1:2023, Edition 1.0.

[56] IDTA - Der Standard für den Digitalen Zwilling. https://industrialdigitaltwin.org/. Abgerufen am 25.05.2024.

[57] Vokabular, Nachrichtenstruktur und semantische Interaktionsprotokolle der I4.0-Sprache. https://www.plattform-i40.de/IP/Redaktion/DE/Downloads/Publikation/hm-2018-sprache.html. Abgerufen am 06.01.2025.

[58] World Wide Web Consortium (W3C). (1999) Resource Description Framework (RDF). https://www.w3.org/RDF/. Abgerufen am 31.12.2024.

[59] International Electrotechnical Commission, *IEC 62714-1:2018 Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language - Part 1: Architecture and general requirements*. International Electrotechnical Commission, 2018, Abgerufen am 01.04.2025. https://webstore.iec.ch/publication/62714-1

[60] International Electrotechnical Commission, *IEC 62541-1: OPC Unified Architecture - Part 1: Overview and concepts*, Nov. 2000, Vol. 2020.

[61] IDTA-01001 - Metamodel of the Asset Administration Shell. https://github.com/admin-shell-io/aas-specs. Abgerufen am 06.01.2025.

[62] ECMA-376, ,,Open Packaging Conventions."

[63] PKWARE Inc., *.ZIP File Format Specification*, 2022.

[64] Process Description: Registration of AAS Submodel Templates for Digital Twins. https://industrialdigitaltwin.org/wp-content/uploads/2023/06/IDTA_Process-Description-Registration-of-AAS-Submodel-Templates-for-Digital-Twins-.pdf. Abgerufen am 14.01.2025.

[65] Plattform Industrie 4.0. (2019) Verwaltungsschale in der Praxis. Wie definiere ich Teilmodelle, beispielhafte Teilmodelle und Interaktion zwischen Verwaltungsschalen. https://www.plattform-i40.de/PI40/Redaktion/DE/Downloads/Publikation/2019-verwaltungsschale-in-der-praxis.html.

[66] N. Braunisch, R. Lehmann, M. Wollschlaeger, M. Ristin, H. W. van de Venn, B. Otto, und T. Kleinert, ,,Maturity Evaluation of SDKs for I4. 0 Digital Twins," in *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2023, S. 1–8.

[67] N. Braunisch, M. Ristin-Kaufmann, R. Lehmann, und 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)*. IEEE, 2021, S. 1–4.

[68] Eclipse BaSyx. https://github.com/eclipse-basyx. Abgerufen am 30.01.2025.

[69] AAS Compliance Tool. https://github.com/rwth-iat/aas-compliance-tool. Abgerufen am 06.01.2025.

[70] International Organization for Standardization (ISO). (2005) Comma-Separated Values (CSV) Format. https://www.iso.org/standard/32508.html. Abgerufen am 31.12.2024.

[71] CSV Schema Language 1.1. https://digital-preservation.github.io/csv-schema/csv-schema-1.1.html. Abgerufen am 06.01.2025.

[72] Object Management Group (OMG). (1997) Unified Modeling Language (UML). https://www.omg.org/spec/UML/. Abgerufen am 31.12.2024.

[73] Object Management Group (OMG). (1997) Object Constraint Language (OCL). https://www.omg.org/spec/OCL/. Abgerufen am 31.12.2024.

[74] World Wide Web Consortium (W3C). (2004) Web Ontology Language (OWL). https://www.w3.org/TR/owl2-overview/. Abgerufen am 31.12.2024.

[75] Google. (2008) Protocol Buffers. https://protobuf.dev. Abgerufen am 31.12.2024.

[76] International Telecommunication Union (ITU). (1984) Abstract Syntax Notation One (ASN.1). https://www.itu.int/en/ITU-T/asn1/Pages/default.aspx. Abgerufen am 31.12.2024.

[77] I. Garmaev, T. Miny, und T. Kleinert, ,,Automatic Generation of Submodel-Specific Classes with Predefined Meta-information Based on Submodel Templates," in *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2023, S. 1–7.

[78] AAS Test Engines. https://github.com/admin-shell-io/aas-test-engines. Abgerufen am 14.01.2025.

[79] Wrong Relationship Type in *.rels. https://github.com/admin-shell-io/aas-test-engines/issues/2. Abgerufen am 01.04.2025.

[80] Check of XML in check_aasx_file is not working. https://github.com/admin-shell-io/aas-test-engines/issues/32. Abgerufen am 01.04.2025.

[81] Allow deprecated relationship in AASX Files. https://github.com/admin-shell-io/aas-test-engines/issues/30. Abgerufen am 01.04.2025.

[82] Optional but empty arrays are considered an error. https://github.com/admin-shell-io/aas-test-engines/issues/15. Abgerufen am 01.04.2025.

[83] Instance deserialized although modelType is missing. https://github.com/aas-core-works/aas-core3.0-python/issues/32. Abgerufen am 01.04.2025.

[84] N. Braunisch, M. Ristin, B. Otto, P. Schanely, H. W. Van De Venn, M. Wollschlaeger, und T. Kleinert, ,,Automatic Fuzzing of Asset Administration Shells as Digital Twins for Industry 4.0," in *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2024, S. 3525–3530.

[85] G. Grieco, M. Ceresa, und P. Buiras, ,,QuickFuzz: An automatic random fuzzer for common file formats," *ACM SIGPLAN Notices*, Vol. 51, Nr. 12, S. 13–20, 2016.

[86] P. Godefroid, B.-Y. Huang, und M. Polishchuk, ,,Intelligent REST API data fuzzing," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, S. 725–736.

[87] L. Attouche, M.-A. Baazizi, D. Colazzo, F. Falleni, G. Ghelli, C. Landi, C. Sartiani, und S. Scherzinger, ,,A tool for JSON schema witness generation," in *24th International Conference on Extending Database Technology*. OpenProceedings. org, 2021, S. 694–697.

[88] S. Jan, A. Panichella, A. Arcuri, und L. Briand, ,,Automatic generation of tests to exploit XML injection vulnerabilities in web applications," *IEEE Transactions on Software Engineering*, Vol. 45, Nr. 4, S. 335–362, 2017.

[89] P. M. Maurer und J. Carbone, ,,Generating Random XML Files Using DGL," in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2022, S. 690–694.

[90] F. A. d. Santos, H. A. C. Cardoso, J. d. C. Costa, V. F. P. Carvalho, und J. C. Ramalho, ,,Datagen: JSON/XML dataset generator," *10th Symposium on Languages, Applications and Technologies*, 2021.

[91] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001.

[92] JSON Schema Test Suite. https://github.com/json-schema-org/JSON-Schema-Test-Suite. Abgerufen am 17.07.2024.

[93] Upgrade of the Validator to Metamodel Version 3.0. https://github.com/eclipse-aas4j/aas4j/issues/9. Abgerufen am 01.04.2025.

[94] T. Miny, S. Heppner, I. Garmaev, M. Ristin, B. Otto, N. Braunisch, M. Jacoby, T. Kleinert, H. W. Van De Venn, und M. Wollschlaeger, ,,Towards a Test Framework for Reactive (Type 2) Asset Administration Shell Implementations," in *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2024, S. 01–08.

[95] M. Jacoby, M. Baumann, T. Bischoff, H. Mees, J. Müller, L. Stojanovic, und F. Volz, ,,Open-source implementations of the reactive Asset Administration Shell: A survey," *Sensors*, Vol. 23, Nr. 11, S. 5229, 2023.

[96] V. Atlidakis, P. Godefroid, und M. Polishchuk, ,,Restler: Stateful REST API fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, S. 748–758.

[97] A. Arcuri, J. P. Galeotti, B. Marculescu, und M. Zhang, ,,EvoMaster: A search-based system test generation tool," 2021.

[98] J. Gosling, B. Joy, G. L. S. Jr., und G. Bracha, *The Java™ Programming Language*, 4. Aufl. Addison-Wesley Professional, 2005.

[99] H. Wu, L. Xu, X. Niu, und C. Nie, ,,Combinatorial testing of restful APIs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, S. 426–437.

[100] S. Karlsson, A. Čaušević, und D. Sundmark, ,,QuickREST: Property-based test generation of OpenAPI-described RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, S. 131–141.

[101] E. Viglianisi, M. Dallago, und M. Ceccato, ,,Resttestgen: Automated black-box testing of restful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, S. 142–152.

[102] Docker, Inc. Docker Documentation. https://docs.docker.com/. Abgerufen am 01.04.2025.

[103] I. Sommerville, *Software Engineering*, 10. Aufl. Pearson, 2015.

[104] M. Cohn, *Agile Estimating and Planning*. Prentice Hall, 2005.

[105] Ambiguity of Constraint AASd-131 related to XOR vs OR. https://github.com/aas-core-works/aas-core-meta/issues/278. Abgerufen am 06.01.2025.

[106] Eclipse Foundation. Eclipse AASX Server. https://github.com/eclipse-aaspe/server. Abgerufen am 01.04.2025.

[107] Antora Meta-Repository - Asset Administration Shell (AAS). https://github.com/admin-shell-io/aas-specs-antora. Abgerufen am 06.01.2025.

[108] T. H. Miny, *Konzept für die semantische Interoperabilität zwischen Informationsmodellen*. Lehrstuhl für Informations- und Automatisierungssysteme für die Prozess- und Werkstofftechnik, 2022, Dissertation.

[109] OpenAPI: Links. https://swagger.io/docs/specification/v3_0/links. Abgerufen am 14.01.2025.

[110] J. Edvardsson, ,,A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, S. 21–28.

[111] R. Romli, S. Sulaiman, und K. Z. Zamli, ,,Automatic programming assessment and test data generation a review on its approaches," in *2010 International symposium on information technology*, Vol. 3. IEEE, 2010, S. 1186–1192.

[112] Fences. https://github.com/ifak/fences. Abgerufen am 14.01.2025.