Universität Bremen

University of Bremen

Faculty 3 – Mathematics and Computer Science

Design and Implementation of an IoT Device Description Converter between SDF and WoT TD

by

Jan Romann

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science (B. Sc.)

First Supervisor: Dr. Olaf Bergmann

Second Supervisor: Prof. Dr. Carsten Bormann

Bremen, July 1, 2022

Abstract

The W3C Web of Things Thing Description (WoT TD) and the Semantic Definition Format (SDF) are two device description specifications for the Internet of Things (IoT). They both aim at solving interoperability issues in the IoT: A WoT TD describes metadata and interfaces of Things, while SDF provides a universal format for data and interaction model definition and conversion. However, although both specifications attempt to solve similar problems, there is not yet a canonical mapping between the two description formats.

This bachelor thesis proposes such a mapping and provides a flexible converter written in Python as an implementation, which can be used from the command line, as a library, and as a web application for converting SDF models to WoT documents and vice versa.

Based on our mappings and our implementation, we were able to identify gaps in both specifications which previously prevented a comprehensive conversion between the two formats. Using new concepts for adding instance- and ecosystem-specific information to SDF and describing nested WoT data structures in a single document, we have been able to bridge the gap between the two specifications, making contributions to the standardization process in both cases. In this context, we have found that WoT Thing Models (TMs), a variant of TDs similar to SDF models used to describe device *classes*, can act very well as intermediaries when converting between SDF models and TDs.

Zusammenfassung

Die W3C Web of Things Thing Description (WoT TD) und das Semantic Definition Format (SDF) sind zwei Gerätebeschreibungsspezifikationen für das Internet of Things (IoT). Beide zielen auf die Lösung von Interoperabilitätsproblemen im IoT ab: Eine WoT TD beschreibt Metadaten und Schnittstellen von IoT-Geräten, während SDF ein universelles Format für die Definition und Konvertierung von Daten- und Interaktionsmodellen bietet. Obwohl beide Spezifikationen versuchen, ähnliche Probleme zu lösen, gibt es noch keine kanonische Abbildung zwischen den beiden Beschreibungsformaten.

Diese Bachelorarbeit schlägt eine solche Abbildung vor und stellt einen flexiblen, in Python geschriebenen Konverter als Implementierung zur Verfügung, der von der Kommandozeile, als Bibliothek und als Webanwendung zur Konvertierung von SDF-Modellen in WoT-Dokumente und umgekehrt verwendet werden kann.

Basierend auf unserer Abbildung und unserer Implementierung konnten wir Lücken in beiden Spezifikationen identifizieren, die bisher eine umfassende Konvertierung zwischen den beiden Formaten verhinderten. Mit neuen Konzepten für das Hinzufügen von instanz- und ökosystemspezifischen Informationen zu SDF und die Beschreibung von verschachtelten WoT-Datenstrukturen in einem einzigen Dokument konnten wir die Lücke zwischen den beiden Spezifikationen schließen und in beiden Fällen zum Standardisierungsprozess beitragen. In diesem Zusammenhang haben wir festgestellt, dass WoT Thing Models (TMs), eine Variante von TDs, die ähnlich wie SDF-Modelle der Beschreibung von Geräte*klassen* dienen, sehr gut als Vermittler bei der Konvertierung zwischen SDF-Modellen und TDs fungieren können.

Contents

Abstract iii

Zusammenfassung v

List of Acronyms xi

List of Tables xiii

List of Figures xv

List of Listings xvii

1 Introduction 1

2 Foundations 5

2.1 Standards and Specifications 5

2.1.1 Web of Things 5

2.1.2 Semantic Definition Format 10

2.2 Related Work 13

3 Mappings between WoT TD and SDF 15

3.1 General Considerations 15

3.2 Mapping between SDF Models and WoT TMs 16

3.2.1 Atomic Units and Nesting 19

3.2.2 Context Extensions and Namespaces 22

3.2.3 Data Schemas and Data Qualities 24

3.2.4 schemaDefinitions and sdfData 25

3.2.5 Interaction Affordances 26

3.2.6 References 28

3.2.7 Mapping of Additional Properties 28

3.2.8 WoT-specific Mappings 28
3.2.9 SDF-specific Mappings 31
3.3 Mappings between WoT TMs and WoT TDs 33

4 Implementation Requirements 35
4.1 Functional Requirements 35
4.2 Non-Functional Requirements 37

5 Design 39
5.1 Structure 39
5.2 Components 40
5.2.1 Library 40
5.2.2 Command Line Interface 42
5.2.3 Web Application 45

6 Implementation 47
6.1 Technologies Used 47
6.2 Library Implementation 49
6.2.1 Internal Conversion Functions 49
6.2.2 Testing 55
6.3 CLI Tool 56
6.4 Web Application 57

7 Evaluation 59
7.1 Requirements Evaluation 59
7.1.1 Comprehensive Mapping 60
7.1.2 Roundtripping 61
7.1.3 Conversion between TDs and TMs 62
7.1.4 Deployment Capabilities 62
7.1.5 Other Requirements 63
7.2 Quantitative Criteria 63
7.2.1 Code Metrics 63
7.2.2 Performance Comparison 64
7.3 Possible Specification Improvements 66

8 Conclusion 69

A Mapping Examples **73**
A.1 SDF Data Qualities and WoT Data Schemas 73
A.2 References and Required Elements 75

B Evaluation Results **77**
B.1 Code Metrics 77
B.2 Performance Comparison 81

Bibliography **89**
Technical Specifications 89
Additional References 92

List of Acronyms

API Application Programming Interface. 13, 14, 39, 41, 49, 50, 63, 69

CDDL Concise Data Definition Language. 11, 48, 65

CI Continuous Integration. 56, 64, 65

CLI Command Line Interface. 36, 37, 39–44, 55–57, 62, 63, 69, 71

CSS Cascading Style Sheets. 57

CURIE Compact URI. 22

DTDL Digital Twin Definition Language. 13, 14

GUI Graphical User Interface. 69

HTML Hypertext Markup Language. 48, 57

HTTP Hypertext Transfer Protocol. 29, 36, 63

HTTPS Hypertext Transfer Protocol Secure. 63

IETF Internet Engineering Taskforce. 2, 6, 89–91

IoT Internet of Things. iii, v, 1, 2, 5, 13, 67, 69

IP Internet Protocol. xi, 14

IPSO IP for Smart Objects. 14

IRI Internationalized Resource Identifier. 22, 23, 28

ISO International Organization for Standardization. 37

JSON JavaScript Object Notation. xii, 6–12, 14, 17, 20, 24–26, 28–30, 33, 36, 37, 39–43, 45, 47–52, 54, 60, 61, 63, 67, 69

JSON-LD JSON Linked Data. 6, 7, 12, 14, 15, 19, 22, 23, 28, 33, 55, 60, 67, 70

OCF Open Connectivity Foundation. 2, 13, 14

OMA Open Mobile Alliance. 2, 13, 14

OneDM One Data Model. 2, 13, 43, 63, 65, 81

RDF Resource Description Framework. 6, 23, 61, 70

REST Representational State Transfer. 49, 63, 69

RFC Request for Comments. 6, 70

SBC Single-board computer. 62

SDF Semantic Definition Format. iii, v, xvii, 2, 3, 5, 7, 8, 10–37, 39–43, 45, 47, 48, 51–53, 57, 59–63, 65, 66, 69, 70, 73, 75, 81

SDO Standards Developing Organization. 1, 2, 13

SPDX Software Package Data Exchange. 31

TD Thing Description. iii, v, xvii, 1–3, 5–17, 22–24, 27, 29, 30, 32–36, 39–44, 47, 48, 54, 55, 59–62, 66, 67, 69, 70

TM Thing Model. iii, v, xvii, 3, 7–11, 13, 15–25, 27–34, 36, 39–44, 51–55, 59–62, 64–67, 69, 70

URI Uniform Resource Identifier. xi, 12, 20, 22, 31, 36, 49

URL Uniform Resource Locator. 17, 33, 43, 44, 60, 64

W3C World Wide Web Consortium. iii, v, 1, 5, 89–92

WoT Web of Things. iii, v, xvii, 1–3, 5–37, 39–44, 47, 48, 52–54, 59–63, 65–67, 69, 70

WWW World Wide Web. 1, 5

YANG Yet Another Next Generation. 14, 65, 70

List of Tables

Table 3.1: Overview of mappings of the most important SDF keywords to WoT. . 17
Table 3.2: Overview of mappings of the most important WoT classes and keywords to SDF. 18
Table 3.3: Directly convertible data schema/data quality fields. 25

Table 5.1: Internal module structure of our converter library. 42
Table 5.2: Available Parameters for the Sub-Commands of the CLI 44

Table B.1: Performance comparison between our converter and the SDF-YANG-Converter. 81

List of Figures

Figure 3.1: High-level view on our conversion process 16

Figure 6.1: Screenshot of our Converter's Web Interface. 58

List of Listings

Listing 2.1: Example of a WoT TD. 7
Listing 2.2: Example of a WoT TM. 8
Listing 2.3: Example of a WoT TM using import, extension, and composition. . . 9
Listing 2.4: Example of an SDF model. 11
Listing 2.5: Example of an SDF mapping file. 12

Listing 3.1: Example for a Thing Model Collection. 21
Listing 3.2: SDF model of the Thing Model Collection Example in Listing 3.1. . . 21
Listing 3.3: SDF Namespaces Block example. 23
Listing 3.4: Mapped TM created from the SDF example in Listing 3.3. 23
Listing 3.5: SDF Information Block example (without URIs). 31
Listing 3.6: Mapped TM created from the SDF example in Listing 3.5. 32

Listing 6.1: Code of our main conversion function `map_field`. 50

Listing 7.1: Example for a valid but unsatisfiable schema in [I-D.-jso-draft-7] . . 67

Listing A.1: SDF model for illustrating the conversion of dataqualities. 73
Listing A.2: Mapped TM created from the SDF example in Listing A.1. 74
Listing A.3: SDF example for the mapping sdfRef and sdfRequired. 75
Listing A.4: Mapped TM created from the SDF example in Listing A.3. 76

Listing B.1: Cyclomatic Complexity values for all elements of our library implementation. 77

1 Introduction

The term Internet of Things (IoT) describes the integration of a wide variety of devices such as sensors, smart lamps or industrial machines into the internet. The interoperability of such IoT devices is currently quite limited as many manufacturers tend to create their own ecosystems and standards which are not necessarily compatible with those of other manufacturers. Besides not being able to interoperate due to differing or even proprietary communication protocols, devices from different manufacturers often rely on different description frameworks, preventing them from identifying the features and capabilities their peers offer. Furthermore, manufacturers and Standards Developing Organizations (SDOs) tend to use their own data modeling approaches for IoT devices, creating further interoperability and re-usability problems.

Over the last couple of years, there have been increasing efforts to tackle both the problem of missing device interoperability and incompatible data models with open standards. Analogous to how the World Wide Web (WWW) operates on top of the Internet at the application layer, giving users a simple yet secure way to interact with web resources via a web browser, the World Wide Web Consortium (W3C) tries to establish a similar relationship between the Internet of Things and the Web of Things (WoT).

Just as browsers commonly access websites using an index.html[1] file on a web server as an entry point, Things in the WoT architecture [wot-architecture] are supposed to be accessed using a so-called Thing Description (TD) [wot-td]. The TD serves as the entry point to the Thing, exposing both device and communication metadata. Consumers can use the information contained in a TD to interact with the Thing based on their application logic or use the human-readable information in the TD for rendering user interfaces. While IoT devices can host their own TDs, already existing or constrained

1 Note that this a convention, not a standard.

[RFC7228] devices which cannot offer a TD themselves can also be integrated into the Web of Things by letting an intermediary provide a TD instead.

Another open standard that deals with the second described problem of incompatible data models is the Semantic Definition Format (SDF) [I-D.-asdf-sdf]. Developed by the Internet Engineering Taskforce (IETF), SDF has the goal of defining a common language for data and interaction models which can be used as a translation medium for models from different ecosystems.

The development of SDF was initiated by the Liason Group One Data Model (OneDM) which is endorsed by the Open Connectivity Foundation (OCF) and Open Mobile Alliance (OMA) SpecWorks, two important SDOs not only in the area of IoT data modelling. SDF aims at providing a universal format for describing IoT devices, thus being able to serve as an intermediary between different ecosystems and data modeling frameworks. Similar to WoT, SDF's primary goal is not to create a new standard to make existing solutions obsolete, but rather to *describe* them in order to bridge the gap between the different ecosystems. Bridging this gap requires the specification of mappings between SDF and the data model framework of interest as well as the implementation of a corresponding converter.

As both SDF and WoT TD try to provide solutions for the interoperability problem in the IoT a conversion between these two formats is particularly interesting. This way, vendor or SDO-specific formats could be easily integrated into the Web of Things, given that a converter between SDF and the format in question exists. This way, WoT TD could serve as a medium for realizing interoperability at the consumer level, while SDF can act as a mediator between different ecosystems, thus providing a broader sense of interoperability.

However, there are still a number of obstacles that have to be cleared before this goal can be achieved. The most important problem that has to be solved is the general mapping between SDF and WoT TD, i.e., defining how an SDF model should be converted into a Thing Description and vice versa. While there have been activities in creating converters between the two formats, there is still no canonically defined mapping. How such a mapping can look like is therefore the main question this thesis deals with, making a concrete proposal in the process. As both SDF and WoT TD are still in development at the time of writing of this thesis, another related question is which gaps need to be closed in order to actually produce a complete mapping between the two formats. A third problem arises from the different purposes of SDF and WoT TD: While SDF mostly

deals with *classes* of devices, the main purpose of WoT TD is the description of device *instances*. As WoT TD defines its own format for device classes called Thing Models (TMs), these can be used as a direct equivalent of SDF models. The question of how to also cover the instance information of TDs in SDF (using, for example, the newly defined concept of mapping files [I-D.-sdf-mapping]) is, therefore, another aspect we deal with in this thesis. Lastly, we will deal with the question of how well conversions between WoT TD and SDF can be reversed or *roundtripped*, and under which circumstances the same document can or cannot be obtained when transforming a conversion result back to its original format.

In the remainder of this thesis, we will first lay the groundwork for the rest of this thesis by presenting the relevant specifications and related work (chapter 2). Then, we will specify the mappings between SDF and WoT TD (chapter 3) before deriving the requirements for our implementation (chapter 4), which will inform the design of our converter (chapter 5). After describing our actual converter implementation (chapter 6), we will then evaluate it in chapter 7. Finally, in chapter 8 we will draw a conclusion and outline potential avenues for future work.

2 Foundations

This chapter gives an overview of the standards and specifications (section 2.1) relevant for this thesis, as well as relevant related work (section 2.2).

2.1 Standards and Specifications

Despite the fact that both WoT and SDF focus on providing data and interaction models for the Internet of Things, both specifications follow different approaches and philosophies. In this section, we explore the two specifications in more detail while outlining their differences, which will serve as the foundation for the development of mappings between them in chapter 3.

2.1.1 Web of Things

The WoT family of standards tries to apply principles of the World Wide Web (WWW) to the Internet of Things (IoT). Similar to how the Web enables users to easily use the internet and access resources hosted on a server, the Web of Things tries to make IoT devices more accessible and increase their interoperability. Within the Web of Things (WoT) architecture [wot-architecture], the Thing Description (TD) is the most important building block. Published as a World Wide Web Consortium (W3C) Recommendation [wot-td], the first version of the TD specification is an official Web Standard since 2020. The WoT working group, however, is currently working on improving the standard, which will be published as version 1.1 in the upcoming months. The latest working draft [wot-td11] of this new version also serves as the basis for this thesis and the presented mappings.

Being compared to the (commonly used) index.html file of a website [wot-architecture], the TD is supposed to serve as an “entry point” of a Thing, by exposing its so-called interaction affordances (properties, actions, and events) as well as metadata. In order to be able to communicate with a Thing, TDs must provide protocol bindings linking an affordance to a concrete resource provided by the Thing as well as security information, such as the requirement for basic authentication [RFC7617] or the use of OAuth 2.0 [RFC6749].[1]

Serialized as JSON [RFC8259], a TD is a JSON-LD document [json-ld], which allows for the linking to other documents for importing additional vocabulary as well as the inclusion of machine-readable, semantic annotations which are compatible with the Resource Description Framework (RDF) [rdf].

Using vocabulary from the popular, yet not formally standardized[2] JSON Schema (Draft 7, specified in [I-D.-jso-draft-7]), TDs allow for defining data schemas, which can be used to validate input and output data. A WoT consumer that wants to interact with a Thing, for example, can use schema information to both know beforehand which kind of input data a Thing expects and to make sure that it received a valid payload in a response from a Thing.

Listing 2.1 shows an example of a TD. It contains the most important TD elements, namely the JSON-LD `@context`, metadata (security definitions, a human-readable title, and an ID), and interaction affordances (a property, an action, and an event) with protocol bindings (in this case all three affordances use HTTPS) in the `forms` member. The `type` member in the status property is one of the vocabulary terms borrowed from [I-D.-jso-draft-7] and can be used to prescribe a range of different mostly JSON-inspired data types.[3] Additional vocabulary can be used to further constrain the set of valid data, indicating, for example, the minimum or maximum of numeric values, a certain pattern for the formatting of a string, or the structure of complex data types (for instance, the

1 The security schemes defined in the TD specification are currently limited to a number of HTTP-specific mechanisms, some of which (such as basic authentication) can be adapted to other protocols like MQTT (requiring a username and password for the communication with a broker). For TD 2.0, it is planned to overhaul this part of the specification by making it easier to extend and less HTTP-specific.

2 While the JSON Schema authors have submitted the different versions of their specification as IETF Internet-Drafts, none of these drafts have reached the status of a Request for Comments (RFC) yet. Furthermore, all the drafts intend to only have an *informational* status, that is, they do not intend to become Internet Standards. This makes them unfit for being referenced in a *normative* context by a technical specification.

3 Namely null, boolean, string, integer, number, array, and object.

Listing 2.1: Example of a WoT TD describing a simple smart light. The TD contains one interaction affordance of each kind (one action, property, and event) as well as basic metadata (a title, an identifier, and a security scheme specifying basic authentication [RFC7617]).

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in": "header"}
    },
    "security": "basic_sc",
    "properties": {
        "status": {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"}]
        }
    },
    "actions": {
        "toggle": {
            "forms": [{"href": "https://mylamp.example.com/toggle"}]
        }
    },
    "events":{
        "overheating":{
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

maximum length of an array). Just as TDs can be validated using a (non-normative) JSON Schema definition provided alongside the TD specification, the data schema definitions inside a TD can be used as inputs for JSON Schema validators for validating input and output data.

Version 1.1 of the WoT TD specification adds a number of new features, of which the so-called Thing Model (TM) is probably the most important one for this thesis and the design of the SDF WoT converter. TMs provide reusable templates for TDs. Similar to SDF models, TMs can be used to describe *classes* of Things, which can be instantiated by converting the respective TM to a TD. TMs are near supersets of TDs[4] and allow omitting instance-specific information such as protocol bindings and security information.

4 TMs do require an additional JSON-LD type annotation, which is the main reason why not every Thing Description is also Thing Model. In the other direction, however, each Thing Model that provides the required instance-specific definitions and does not use the import or extension mechanisms qualifies as a Thing Description.

Listing 2.2: Example of a WoT TM, based on the TD shown in Listing 2.1, with all instance-specific information omitted and with a placeholder for the `id` field.

```
1  {
2      "@context": "https://www.w3.org/2022/wot/td/v1.1",
3      "id": "urn:dev:{{IDENTIFIER}}",
4      "title": "MyLampThing",
5      "properties": {
6          "status": {
7              "type": "string"
8          }
9      },
10     "actions": {
11         "toggle": {}
12     },
13     "events":{
14         "overheating":{
15             "data": {"type": "string"}
16         }
17     }
18 }
```

An example for a TM can be seen in Listing 2.2. Here, we omit all instance-specific information (i.e., the affordances' forms and the security definitions) and include a so-called placeholder (denoted by a double-pair of curly braces) in the `id` field. These placeholders are supposed to be replaced during the conversion process from a TM to a TD and allow for any data type to be used as a replacement value in a so-called placeholder map [wot-td11, section 10.3.3]. Therefore, TM fields which normally do not allow for using strings allow for them in this special case if the string only consists of the placeholder itself.

Similar to SDF models (as we will see in the next section), TMs feature a vocabulary for importing from other Thing Models, using JSON Pointers [RFC6901] and the keyword `tm:ref`. Furthermore, they allow for extending other TMs using a special relation-type called `tm:extends` in a link object. The WoT TD specification describes a process for deriving Thing Descriptions from Thing Models, resolving all references and extensions and therefore instantiating it. Using the `tm:required` keyword, interaction affordances can be defined as mandatory, making it obligatory to take them over into the resulting TD during the derivation process.

WoT also includes a mechanism for nesting TMs and TDs by using links. In the case of TMs, this is achieved with a special link-relation `tm:submodel`, which also allows for reusing the same TM with a different `instanceName`, which is applied during the conversion from TMs to TDs, a process well-defined in the WoT TD specification [wot-

Listing 2.3: Example of a WoT TM using an import (with the `tm:ref` keyword), the extension mechanism (with the link-relation `tm:extends`), and the composition mechanism (using `tm:submodel`).

```
1  {
2      "@context": "https://www.w3.org/2022/wot/td/v1.1",
3      "title": "MyLampThing",
4      "property": {
5          "status": {
6              "tm:ref": "https://example.org/reference-tm#status"
7          }
8      },
9      "links": [
10         {
11             "href": "https//example.org/sub-tm",
12             "rel": "tm:submodel",
13             "instanceName": "Sub-Lamp"
14         },
15         {
16             "href": "https//example.org/parent-tm",
17             "rel": "tm:extends"
18         }
19     ]
20 }
```

td11, section 10.4]. In the case of TDs, the relation-type `item` is supposed to be used to indicate that a TD has subordinates. The semantic difference between `tm:submodel` and `item` is not entirely clear, which becomes apparent when considering the conversion of a (nested) TD to a TM, where we could imply the same kind of relationship between the resulting TMs using `item` instead of `tm:submodel` as relation-type. This is one aspect where future versions of the WoT TD specification need a bit of clarification.

An example for a TM that uses all three available mechanisms for re-using definitions from other models can be seen in Listing 2.3. Here, we "import" the content of the status property from another TM, while inheriting definitions from another TM as a whole using a link with `tm:extends` relation-type. For the resolution of imports, the use of the JSON Merge Patch algorithm [RFC7396] is prescribed by the specification for updating the current definition with the contents of the referenced one. For the extension mechanism, however, the specification only defines a number of assertions but no formal algorithm, which leaves room for how the extension mechanism should be handled in practice. Lastly, in our example, we also use a link with the `tm:submodel` relation-type, referencing another TM as a submodel and indicating a hierarchical relationship between the two. The `instanceName` indicates here that this is other model is a "Sub-Lamp", but could be used in other scenarios, where we reference one TM multiple times, to

differentiate between different instances of the underlying class.[5]

During the description of our mapping between WoT documents and SDF in chapter 3, we will explore and describe the defined TD vocabulary (which is listed and described in detail in the specification itself) in more depth. First, however, we will turn to the other major specification relevant for this thesis.

2.1.2 Semantic Definition Format

The Semantic Definition Format (SDF, [I-D.-asdf-sdf]) strives to be a universal format for describing IoT data models, thus trying to solve the interoperability issue of IoT modelling languages and their corresponding ecosystems. Its current focus lies on the description of classes of components and devices. Therefore, SDF models are supposed to only contain abstract information that is independent of protocols or manufacturers, in order to maximize their reusability. Ecosystem, protocol or instance-specific information is supposed to be included in companion documents, such as mapping files [I-D.-sdf-mapping], which augment the definitions of an SDF model specified by JSON pointers. An SDF model and one or more corresponding mapping files can be merged to create a consolidated model, which can serve as the basis of ecosystem-specific conversions.

Listing 2.4 shows a simple example of an SDF model that describes a switch. Similar to a WoT TD or TM, the SDF model contains interaction affordances—an action, a property, and an event—which indicate the possibilities for the outside world to interact with the device. However, a key difference between SDF and WoT lies in the fact that the affordances are part of an `sdfObject` or `sdfThing`, which serve as their container.[6] The `sdfObject` class is the main reusable component in SDF and forms the "leaf nodes" of SDF models. In order to create hierarchies in SDF models, the `sdfThing` can be used, which is syntactically equivalent to the `sdfObject` class but can also contain

5 One example here could be a traffic light consisting of three individual lamps, which are derived from the same class of device differing with regard to their (instance-specific) color. In this case, the instance-name can be used to differentiate between the three lamps. Note, however, that it is not possible to pass "arguments" to the other model this way. The concrete information for the color values has to be provided during the TD derivation process. This is another aspect that could be improved in the next major version of the specification.

6 While it is also possible to define "global" interaction affordances as well as `sdfData` definitions at the top of an SDF model, these are only supposed to be referenced/imported from inside `sdfObjects` and `sdfThings`.

Listing 2.4: Example of an SDF model.

```
{
  "info": {
    "title": "Example file for OneDM Semantic Definition Format",
    "version": "2019-04-24",
    "copyright": "Copyright 2019 Example Corp. All rights reserved.",
    "license": "https://example.com/license"
  },
  "namespace": {
    "cap": "https://example.com/capability/cap"
  },
  "defaultNamespace": "cap",
  "sdfObject": {
    "Switch": {
      "sdfProperty": {
        "value": {
          "description": "The state of the switch; false for off and true for on.",
          "type": "boolean"
        }
      },
      "sdfAction": {
        "on": {
          "description": "Turn the switch on; equivalent to setting value to true."
        },
        "off": {
          "description": "Turn the switch off; equivalent to setting value to false."
        },
        "toggle": {
          "description": "Toggle the switch; equivalent to setting value to its complement."
        }
      }
    }
  }
}
```

`sdfObjects` and `sdfThings` as children. Both the `sdfThing` and the `sdfObject` class can be arranged as arrays with the `minItems` and `maxItems` members, which can be used to describe devices such as outlet strips which consist of a number of identical elements (in this case sockets). This is a major difference to WoT documents, which only for linking but no comparable nesting within the same TD or TM.

Similar to WoT TD, SDF uses JSON Schema [I-D.-jso-draft-7] inspired qualities for the definition of data schemas used as properties or interaction inputs or outputs. However, for validation of models, SDF relies primarily on the standardized Concise Data Definition Language (CDDL) [RFC8610], while also offering an equivalent JSON Schema in its appendix. Besides the JSON Schema inspired terms, both SDF and WoT add their own vocabulary for data qualities, which does not have a direct equivalent in the other specification. While WoT allows for extending the allowed vocabulary in a TM

Listing 2.5: Example of an SDF mapping file, augmenting the SDF model in Listing 2.4 with @type annotations from JSON-LD.

```
{
  "info": {
    "title": "Example file for OneDM Semantic Definition Format",
    "version": "2019-04-24",
    "copyright": "Copyright 2019 Example Corp. All rights reserved.",
    "license": "https://example.com/license"
  },
  "namespace": {
    "cap": "https://example.com/capability/cap",
    "saref": "https://w3id.org/saref#"
  },
  "defaultNamespace": "cap",
  "map": {
    "#/sdfObject/Switch": {
      "@type": "saref:LightSwitch"
    },
    "#/sdfObject/Switch/sdfProperty/value": {
      "@type": "saref:OnOffState"
    },
    "#/sdfObject/Switch/sdfAction/toggle": {
      "@type": "saref:ToggleCommand"
    }
  }
}
```

or TD by using a context extension, SDF does require an additional document like the aforementioned mapping files to augment a model using JSON pointers, which indicate where the additional qualities belong. An example for a mapping file augmenting the SDF model in Listing 2.4 is shown in Listing 2.5. Here, we let the mapping file add JSON-LD-specific vocabulary (the semantic `@type` annotations) to the `sdfObject` itself, the property, and the toggle action.

Just like the augmented SDF model itself, the mapping file contains top-level metadata (a title, licensing and copyright information, and a version indicator) in an `info` block. Similar to the JSON-LD context, we can refer to namespaces in both document types, mapping prefixes like `cap` or `saref` to a URI. SDF also allows setting a default namespace, indicating where the definitions of a model belong to. However, SDF does not define the exact semantics of namespaces yet, which is why qualities prefixed with a colon in their quality name or "given name" (e.g., `cap:foo`, forming a so-called [CURIE], which can be expanded to a URI) are forbidden in SDF models at the moment, but are supposed to be added as a feature at a later point in time. The next major version of SDF will probably provide more clarity in this regard. SDF mapping files can provide a workaround here, as they do not forbid the use of prefixed qualities, which we will rely on later when converting WoT definitions to SDF.

SDOs like OMA SpecWorks, OCF or the Zigbee Alliance have contributed a number of (non-official) SDF models in a “playground” under the aegis of OneDM, hosted on GitHub.[7] We used these models both for developing and refining the mappings between SDF and WoT, and as a benchmark for how well the converter performs as part of our evaluation in chapter 7.

2.2 Related Work

There are a number of pre-existing projects which also deal with the conversion between SDF and other formats. The one most closely related to this thesis is a converter by Roman Kravtsov[8] written in JavaScript (for the Node.js ecosystem), which is able to convert SDF models to WoT Thing Models. Kravtsov dealt with the conversion between the two formats in the context of his master’s thesis [Kra21], comparing, besides SDF, multiple formats for semantic descriptions of IoT devices with the newly added Thing Model feature. Besides his SDF converter, he also provides implementations for Oracle Device Models[9], Eclipse Vorto models[10], and Microsoft’s Digital Twin Definition Language (DTDL)[11].

While working in general, Kravtsov’s converter between SDF and WoT has a number of limitations: It does not support the backwards conversion of WoT documents to SDF, can only convert SDF models to Thing Models (there is no support for Thing Descriptions), and only accepts a single `sdfObject` within a model as an input. This also means that Kravtsov’s converter does not support roundtripping, i.e., translating a conversion result back into its original format. Furthermore, there is no validation of both inputs and outputs, while the actual mapping of SDF affordances is a simple copy operation, potentially resulting in WoT TMs containing definitions only specified for SDF (e.g., the `sdfChoice` quality, a more expressive enumeration type based on key-value pairs). Besides these limitations for the actual conversion, the converter has a number of usability issues, neither providing a library API nor a command line interface

7 https://github.com/one-data-model/playground (retrieved: May 5, 2022).
8 https://github.com/roman-kravtsov/sdf-object-converter (retrieved: May 3, 2022).
9 https://github.com/roman-kravtsov/oracle-device-model-converter (retrieved: June 23, 2022).
10 https://github.com/roman-kravtsov/vorto-model-converter (retrieved: June 23, 2022).
11 https://github.com/roman-kravtsov/digital-twins-converter (retrieved: June 23, 2022).

that allows for specifying input and output file names.

Another relevant SDF converter[12] covers the conversion between SDF and the data modelling language YANG (Yet Another Next Generation, [RFC7950]). Written by Jana Kiesewalter in C++, her converter can be used as a standalone command line application, which also allows the integration in web applications.[13] Her converter resembles a more sophisticated approach to SDF conversion than Kravtsov's, also supporting bidirectional conversion and roundtripping. Kiesewalter's converter, which is part of her Master's thesis [Kie21], also features a comprehensive mapping between the two data modelling approaches, which she also codified in an Internet-Draft [I-D.-yang-sdf]. However, there are a number of small issues with the converter which are related to the flexible nature of the JSON Schema inspired vocabulary and are going to be discussed in more detail when formulating our requirements in chapter 4.

Furthermore, there are two converters by Ericsson Research: The first one[14] allows the conversion between SDF and IPSO[15] data models, which is the data modelling approach of OMA SpecWorks. The second converter[16] also allows for a conversion between SDF and DTDL, which is used for the company's Azure Digital Twins models and is, like WoT TD, based on JSON-LD. Finally, OCF provides tools[17] to convert between SDF and OpenAPI, a popular format for describing web APIs.

Through a growing number of converter implementations, we can observe an integration between different ecosystems and data modelling approaches fostered by SDF as a common description framework. However, both comprehensive support for the WoT document formats and the description of instance-specific information have been underdeveloped so far.

On the basis of the foundations outlined in this chapter, we now first define the mappings between SDF and WoT necessary for our converter before describing the requirements for its design and implementation, informed by the existing conversion approaches.

12 https://github.com/jkiesewalter/sdf-yang-converter (retrieved: May 4, 2022).

13 See http://sdf-yang-converter.org/ (retrieved: May 3, 2022).

14 https://github.com/EricssonResearch/ipso-odm (retrieved: May 5, 2022).

15 IP for Smart Objects

16 Although the converter is accessible through a web interface hosted by Ericsson http://wishi.nomadiclab.com/sdf-converter/ (retrieved: June 16, 2022), alongside a number of other SDF converters, its source code and/or project description seems unavailable to the public at the moment.

17 https://github.com/openconnectivityfoundation/SDFtooling (retrieved: June 16, 2022).

3 Mappings between WoT TD and SDF

For the design and implementation of a converter between WoT TD and SDF it is necessary to lay out the mapping between the definitions of both specifications. This chapter proposes such a mapping, which will also serve as the foundation for the requirements outlined in the next chapter, as well as our converter's design (chapter 5).

As we discussed in the last chapter, this mapping does not only include WoT TDs but also TMs, which will serve as an important intermediary between Thing Descriptions and SDF models.

3.1 General Considerations

WoT Thing Descriptions and SDF models have different purposes: TDs describe concrete instances of Things, while SDF models describe classes of Things. The fact that TDs can contain instance-specific information which cannot be expressed in SDF is a challenge for creating a mapping between the two formats, as this information cannot be included directly in SDF models. Therefore, the concept of mapping files has to be used, where the additional information from TDs can be stored. This means that the conversion of a TD creates two documents: An SDF model and an additional mapping file. For the other direction, we also need both kinds of SDF documents as an input as otherwise the resulting TDs will not be valid since some fields in a TD are mandatory.

Due to these limitations, we decided to use Thing Models (TMs), the second kind of WoT document, as an intermediary for the conversion between Thing Descriptions and SDF models. As Thing Models do not require instance-specific information and have a limited number of mandatory fields—a JSON-LD `@context` importing the WoT vocabulary and an `@type` of `tm:ThingModel`—, they can be mapped to a single SDF model if they do not contain fields which have no equivalent in SDF. The fact that WoT TMs are very close

to being a superset of TDs further qualifies them as an intermediary, as a conversion between TDs and TMs can be achieved easily, as we will see below.

Thus, the conversion between the three formats only requires the specification of two concrete mappings: The one between SDF and WoT TMs, and the one between WoT TMs and WoT TDs. The conversion between SDF and WoT TDs can be derived from these two mappings as a corollary. A high-level view on our conversion process can be seen in Figure 3.1. Notice how TMs are serving as an intermediary between TDs on the one hand and SDF models and mapping files on the other here. When using TMs as an input, they can be augmented with additional protocol bindings or metadata, as well as replacement values for potential placeholders.

In the next two sections, we will first outline our mapping between SDF and WoT TMs (section 3.2) before we describe additional mappings between WoT TDs and WoT TMs (section 3.3), building upon the conversion process already described in the WoT TD specification.

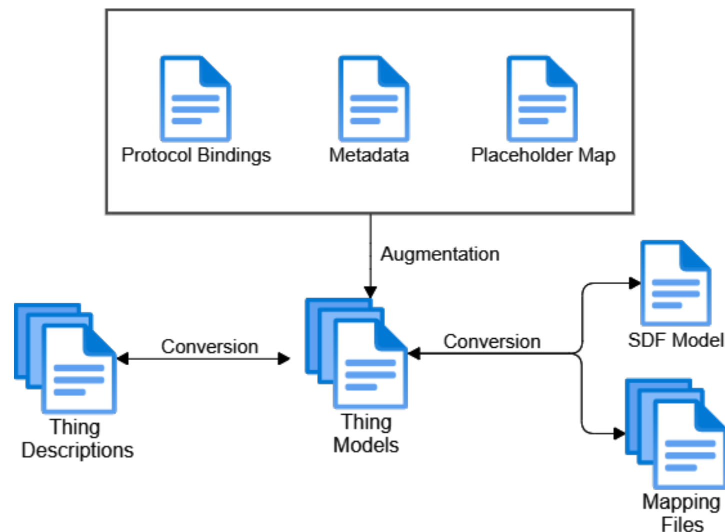

Figure 3.1: High-level view on the conversion process for WoT TMs, WoT TDs, and Thing Models.

3.2 Mapping between SDF Models and WoT TMs

As already mentioned in the last chapter, SDF and WoT documents have a lot of conceptual similarities, while following a different purpose and philosophy in a number

of aspects. Table 3.1 shows an overview of the most important SDF concepts and their equivalent in WoT Thing Models, underlining how similar some parts of the terminology are. The most striking similarity is of course the interaction affordances (actions, properties, events, also see section 3.2.5), but the keywords for referencing (sdfRef, tm:ref) and for indicating required definitions (sdfRequired, tm:required) also show not only the parallels between the two specifications, but also hints at how they influenced each other during the standardization process. However, as Table 3.2 gives an overview of the mapping from WoT TDs to SDF, we can see that there are quite a few concepts with no equivalent in SDF.

Table 3.1: Overview of mappings of the most important SDF keywords to WoT.

SDF Keyword	WoT Class/Keyword
sdfThing	TM with `tm:submodel` links
sdfObject	TM without `tm:submodel` links
sdfProperty	PropertyAffordance
writable	readOnly (negated)
readable	writeOnly (negated)
sdfAction	ActionAffordance
sdfOutputData	output
sdfInputData	input
sdfEvent	EventAffordance
sdfOutputData	output
sdfData	schemaDefinitions (at the TM level)
sdfRef	tm:ref
sdfChoice	Enum of JSON objects with `sdf:choiceName`
sdfRequired	tm:required
namespaces	@context
defaultNamespace	sdf:defaultNamespace
info	*Multiple targets:*
version	`model` field in Version class
title	sdf:title
copryright	sdf:copyright
license	If URL: link with relation-type Else: sdf:license

While the mapping of many definitions is relatively straightforward due to these parallels (SDF actions become WoT actions, SDF properties become WoT properties, WoT data

1 This is the base class of the three affordance types.

Table 3.2: Overview of mappings of the most important WoT classes and keywords [wot-td11, section 5] to SDF.

WoT Class/Keyword	SDF Keyword
Thing	sdfThing (TM has `tm:submodel` links), sdfObject
title	label
description	description
schemaDefinitions	sdfData
@context	namespaces of the SDF model (with exceptions)
DataSchema	dataqualities
readOnly	Mapping File
writeOnly	Mapping File
InteractionAffordance[1]	—
title	label
description	description
PropertyAffordance	sdfProperty
readOnly	writable (negated)
writeOnly	readable (negated)
observable	observable
ActionAffordance	sdfAction
input	sdfInputData
output	sdfOutputData
EventAffordance	sdfEvent
tm:ref	sdfRef
tm:required	sdfRequired
Link	Mapping File, except for special link types (e.g., `license`, `tm:extends`, `tm:submodel`)

schemas become SDF data qualities, and so on), the mapping becomes more complex when taking into account hierarchical models as well as the mechanisms for referencing and extending present in both specifications.

In the following, we will first focus on the mapping of concepts which have equivalents in both specifications, before discussing aspects that are either WoT- or SDF-specific and therefore require a specific treatment before being able to map them to the other format. In general, we will observe that TM fields with no equivalent will become part of an SDF mapping file, while additional SDF fields become part of a resulting TM directly, with an added `sdf:` prefix, indicating that the field is part of an SDF vocabulary extension.[2] This not only ensures that we translate as much of the original document into the other format, but also enables roundtripping, as we will discuss throughout this chapter.

3.2.1 Atomic Units and Nesting

As we realized during the writing of this thesis, the most crucial difference between the two specifications lies in the way both specifications approach hierarchical data models, describing Things which consist of multiple components integrated by a parent component. A simple example for such a Thing also used in the SDF specification would be a combination of a freezer and fridge, forming a fridge-freezer combination with global properties such as a status.

SDF and WoT would describe such a device completely differently. In SDF, it is possible to describe this kind of Thing with a single model, using an `sdfThing` for describing the device as a whole and two `sdfObjects` for the fridge and the freezer, respectively. In WoT, the three components of the device need to be described with their own TM, creating the hierarchy with the linking approach described in the last chapter.

Both specifications designate affordances as part of a container, which resembles the associated Thing or a subordinate part of it. In WoT TMs, these containers are the Thing Models themselves, which can link to "submodels" to create a hierarchy of TMs. A direct inclusion of subordinate TMs in a WoT document is not intended by the specification.

SDF follows a different philosophy, specifying two containers affordances can be associated with: The `sdfObject` serves as a "leaf node" for SDF models which cannot

2 This extension has yet to be formally defined in a JSON-LD compatible format.

contain any subordinate containers (i.e., other `sdfObjects`). The `sdfThing`, however, has nesting capabilities and can contain both other `sdfThings` and/or `sdfObjects`, which can be used to create SDF models of arbitrary depth.

Although being vastly different, the two approaches can be mapped to each other: When mapping from WoT to SDF, each link can be dereferenced and converted into an individual TM. Should a TM link to at least one submodel, we decided that it becomes an `sdfThing` during the conversion process; otherwise, it becomes an `sdfObject`. Finally, all resulting `sdfThings` and `sdfObjects` are collected in one SDF model, in accordance with their position in the hierarchy of links.

A special challenge for the conversion between SDF and WoT TMs lies in the fact that an SDF model does not need to have a clear hierarchy with a single "entry point". Instead, there can be multiple "top-level" sdfThings or sdfObjects, which result in more than one TM with no parent TM. A comprehensive mapping with support for roundtripping needs to take this into account by allowing more than one TM as an input for the conversion process. Furthermore, we saw the need to be able to collect all conversion output TMs in a single document, in order to have a self-contained result, usable for further processing. These considerations led us to the introduction of a concept which we call *Thing Model Collections*. These JSON documents are essentially a map where each value is a TM. The map keys can be used to convert or export each contained TM to a single JSON file.

An example for a Thing Model Collection can be seen in Listing 3.1. Here, we demonstrate how we can map a hierarchical relationship between two Thing Models within the same collection, having the Lamp TM point to the Switch TM using a corresponding JSON pointer in a link with the relation-type `tm:submodel`. In the link, we also indicate an `instanceName`, which is a special addition to the `Link` class for naming submodels, making it possible, for example, to differentiate multiple instances of the same TM with different purposes.[3] Note that the contained TMs can of course also have submodels which are located externally, i.e., under a URI or a relative path on the file system.

While in theory, an `sdfRef` could also be used to reference a definition that is not part of an SDF model (this is not explicitly forbidden in the SDF specification), we chose to specify that all submodels must be dereferenced before the conversion process. This way, the resulting SDF model can be processed directly, without creating a consolidated

3 One example could be a traffic light with three lamps—red, yellow, and green—as submodels. These could share the same TM, but could be differentiated by their `instanceName`.

Listing 3.1: Example for a Thing Model Collection.

```
{
  "Lamp": {
    "@context": ["https://www.w3.org/2022/wot/td/v1.1"],
    "@type" : "tm:ThingModel",
    "links": [
      {
        "rel": "tm:submodel",
        "href": "#/Switch",
        "instanceName": "SubmodelSwitch"
      }
    ]
  },
  "Switch": {
    "@context": ["https://www.w3.org/2022/wot/td/v1.1"],
    "@type" : "tm:ThingModel"
  }
}
```

Listing 3.2: SDF model of the Thing Model Collection Example in Listing 3.1.

```
{
  "sdfThing": {
    "Lamp": {
      "sdfObject": {
        "SubmodelSwitch": {}
      }
    }
  }
}
```

SDF model (that might contain WoT-specific vocabulary) first. Using the newly defined SDF relations extension [I-D.-sdf-relations], however, it might be possible to provide information about the origin of, for example, an SDF object, preserving the `tm:submodel` link with an altered relation-type in the process.[4]

If there is more than one TM provided as input to the TM-to-SDF conversion process (e.g., as part of a Thing Model Collection), one must either explicitly state which TMs are the “top-level” documents or a reference-counting algorithm has to be applied in order to determine which documents are the top-level ones. This algorithm needs to go over each TM and check if any other TM is linking to it. If not, then the TM in question will be moved to the top of the hierarchy and will be placed in the top-level `sdfObject` or `sdfThing` member after the conversion process has finished. If a TM contains definitions which do not have a direct equivalent in SDF, we include these fields in a mapping file.

4 [I-D.-sdf-relations] was published too late in the process of writing this thesis to be considered as part of the actual mappings.

In the SDF-to-WoT direction, we first apply all SDF mapping files (if any) provided as input to SDF model, creating a consolidated SDF model as a result. We then convert all top-level `sdfObjects` within the model directly to TMs. Afterwards, we iterate recursively over the `sdfThings` in the model and convert each one to a TM, doing the same for its child `sdfThings` and `sdfObjects`. If an `sdfThing` contains `sdfObjects` or other `sdfThings`, we reestablish the hierarchical relationship in the conversation results by adding a link in the resulting parent TM, pointing to the resulting child TM. In order to make roundtripping possible, we add a special member field to the resulting TMs or TDs which either has a key of `sdf:objectKey` or `sdf:thingKey`, depending on the kind of SDF definition the TM or TD originated from, containing the key of the original `sdfObject` or `sdfThing`. If the SDF-to-WoT conversion creates more than one TM, we create a Thing Model Collection from the results. Otherwise, the result is a single TM.

3.2.2 Context Extensions and Namespaces

Both WoT TMs and SDF documents feature definitions for namespaces and additional vocabularies, using a (restricted) JSON-LD `@context` or a `namespaces` map, respectively. In both approaches, the document maps terms to either IRIs (WoT) or URIs (SDF), which are then supposed to be used in compact IRIs or Compact URIs (CURIEs) [CURIE], making it possible to import vocabulary from another source, using the terms as prefixes.

In contrast to WoT, however, SDF's namespaces are not as well-defined yet and do not allow for vocabulary extensions in the current version of the draft [I-D.-asdf-sdf, section 2.3.3]. Instead, currently the main purpose of namespaces is for using them in `sdfRef` definitions when referring to external documents, as SDF forbids the use of absolute URIs here [I-D.-asdf-sdf, section 4.3]. Another important difference is that WoT's `@context` not only allows for mapping terms to IRIs, but also for including special keywords like `@language` (which sets the document's default language) and also singular IRIs with no mapping from a term. These IRIs can be used to import one or more namespaces into the document, when resolving the IRI with a JSON-LD processor[5]

5 For example, https://www.w3.org/2022/wot/td/v1.1 is the IRI used to identify WoT TD/TM version 1.1 documents and contains namespaces for all the vocabulary terms defined in the specification, including the ones for TMs.

and are somewhat comparable to SDF's `defaultNamespace` vocabulary term, as they also set the default context of the document and import the WoT TD vocabulary to be used as such. However, as SDF's `defaultNamespace` is rather used to express that an SDF model adds *additional* definitions to that namespace [c. f. I-D.-asdf-sdf, section 4.2], there is an important semantic difference, making it impossible to map these two concepts to one another.

Therefore, while translating those entries in WoT's `@context` that map terms to IRIs to SDF namespace members and vice versa, SDF's `defaultNamespace` becomes a prefixed `sdf:defaultNamespace` in the resulting WoT TM in order to make roundtripping possible. As there is no JSON-LD document containing an SDF vocabulary yet, we let the `sdf` context extension point to `https://example.com/sdf`, indicating that this IRI is supposed to be replaced once an RDF or JSON-LD document for SDF vocabulary is available.

Listing 3.3: Example for the Namespaces Block of an SDF model which also defines a default namespace it contributes to.

```
1  {
2    "namespace": {
3       "cap": "https://example.com/capability/cap",
4       "zcl": "https://zcl.example.com/sdf"
5    },
6    "defaultNamespace": "cap",
7    "sdfObject": {
8      "Example": {
9        "label": "Example Object"
10     }
11   }
12 }
```

Listing 3.4: Mapped TM created from the SDF example in Listing 3.3.

```
1  {
2    "@context": [
3      "https://www.w3.org/2022/wot/td/v1.1",
4      {
5        "cap": "https://example.com/capability/cap",
6        "zcl": "https://zcl.example.com/sdf",
7        "sdf": "https://example.com/sdf"
8      }
9    ],
10   "@type": "tm:ThingModel",
11   "sdf:objectKey": "Example",
12   "sdf:defaultNamespace": "cap",
13   "title": "Example Object"
14 }
```

An example for the mapping between SDF namespaces and a WoT `@context` can be seen in listings 3.3 and 3.4. However, as we noted above, not every WoT `@context`

entry can be included in an SDF model. Therefore, when mapping from WoT to SDF we also include the original `@context` in a mapping file, making it possible to restore it when converting back from SDF to WoT.

3.2.3 Data Schemas and Data Qualities

Both SDF and WoT TMs use vocabulary from [I-D.-jso-draft-7] for defining schemas for properties and for validating input and output data. This is a key reason for a great amount of overlap between WoT's `DataSchema` class and SDF's dataqualities (c.f. Table 3.3), allowing us to map most of the vocabulary directly between SDF and WoT (see Appendix A.1 for an example mapping between SDF data qualities and WoT `DataSchemas`). There are a number of cases, though, which require special attention and are specific to the two specifications.

SDF fields which have no direct equivalent in WoT TMs are `nullable`, `sdfType`, `uniqueItems`, and `sdfChoice`. `nullable`, `uniqueItems`[6], and `sdfType` are mapped directly to a WoT `DataSchema`, prepending an `sdf:` prefix to the key of the mapping result. We need a different strategy, however, for the `sdfChoice` field, which can actually be integrated into the `enum` field of a WoT data schema. To do so, we convert each entry in the `sdfChoice` object field into an individual `DataSchema` instance, where we add the entry's key to an additional `sdf:choiceName` field in order to enable roundtripping. Should the SDF definition also contain an enum, the conversion process is supposed to be aborted with a validation error in order to prevent inconsistencies. In future mapping versions, this aspect should be revisited once the relationship between `sdfChoice` and `enum`, in the case both are present alongside each other, has been clarified.

One key difference between WoT and SDF is the use of JSON Schema's `readOnly` and `writeOnly` qualities in WoT data schemas. In the latest version of the SDF draft, the SDF equivalents `writable` and readable have been limited to properties, which inherit from the WoT data schema class and SDF data qualities, respectively, making it necessary to include `readOnly` and `writeOnly` in a mapping file when used in a data schema which is not also a property. The problems with mapping the semantics of these two qualities to SDF are discussed in more detail in section 3.2.5.1.

6 SDF borrows the `uniqueItems` term from the newer JSON Schema draft 2019 [I-D.-jso-draft-2019-09], which might also be supported in future versions of WoT TD.

7 Might also contain values from `sdfChoice` when converting from SDF to WoT.

Table 3.3: Directly convertible data schema/data quality fields.

Field Name
pattern
format
required
maximum
minimum
exclusiveMaximum
exclusiveMinimum
maxItems
minItems
maxLength
minLength
multipleOf
default
const
enum[7]
unit
type

3.2.4 schemaDefinitions and sdfData

Besides the usage in properties and for defining schemas of input and output data, both WoT and SDF also define special qualities—`sdfData` and `schemaDefinitions`—which primarily serve for being referenced from other definitions using `sdfRef` or `tm:ref`, respectively.[8] While `sdfData` can be mapped to WoT's `schemaDefinitions`, a major difference for the use of the qualities is that `sdfData` can not only be used on the `sdfThing` or `sdfObject` level, but also in SDF's actions and events. Mapping these `sdfData` fields to a WoT TM `schemaDefinitions` field poses the problem that there are potential naming conflicts if there is a definition with the same key in another `sdfData` object. Furthermore, roundtripping problems appear as it is not clear where the mapped `sdfData` originated from without providing an additional JSON pointer to the affordance or the original definition in the SDF model. Therefore, we map the entries of affordance-level `sdfData` fields `schemaDefinitions` entries, with a key composed

8 WoT's `schemaDefinitions` also serve the purpose of being referenced from a special `AdditionalExpectedResponse` class, not using a JSON pointer but the key of the schema definition that is supposed to be referenced. WoT also uses this pattern for its security schemes (described in section 3.2.8.1).

of an escaped JSON pointer [c.f., RFC6901, section 3], replacing / characters with ~1. That is, an sdfData entry called foo in an sdfAction called bar would become sdfAction~1bar~1sdfData~1foo.

While this approach allows for roundtripping due to the encoded JSON pointer, it is not a very elegant solution. As an alternative, we considered using an additional field sdf:sdfData in order to replicate the original structure of the sdfData placements, which has the drawback of requiring an additional vocabulary term. This aspect of our mapping probably requires more discussion in the future.

3.2.5 Interaction Affordances

Both SDF and WoT use a concept called affordances to describe how consumers can interact with a Thing. Both specification divide them into properties, actions, and events, although SDF uses slightly different object keys in its models (sdfProperty, sdfAction, and sdfEvent). Therefore, a direct mapping of affordances is generally possible, with only minor differences in semantics, which will be outlined below for the three affordance types.

3.2.5.1 Properties

In both SDF and WoT, properties can be considered subclasses of the Data Qualities and Data Schema definitions, respectively, adding a number of fields which are relevant for interactions. In both cases, this includes an observable field which indicates that consumers are able to signal to the Thing that they wish to be informed about state changes of the property in question.[9] This field is directly mappable, as it is both syntactically and semantically equivalent. Observable has a different default value in SDF (true) and WoT (false), though, which has to be taken into account when the value is unset. This has minor implications for roundtripping, as the initial implicit default will become set explicitly during the conversion process, reproducing a document which is not strictly equivalent to the original.

9 In practice, this could be implemented, for example, using CoAP's observe option, as specified in [RFC7641].

As mentioned in section 3.2.3, a more important difference lies in the semantics of WoT's `readOnly` and `writeOnly` compared to SDF's `writable` and `readable`. While the use case of data qualities which are neither readable nor writable might not be very relevant in practice, in SDF it can still be expressed by setting both fields to `false`. The `writable` (`readable`) field set to false and `readOnly` (`writeOnly`) set to true are equivalent on their own. Semantically, however, `readOnly` and `writeOnly` cannot both be true at the same time, while they can be syntactically. Therefore, this aspect of WoT properties is not directly mappable to SDF. However, there is an ongoing discussion in the WoT TD taskforce regarding this topic[10], which might lead to an alignment with SDF in the next major version of the WoT TD specification.

3.2.5.2 Actions

The `sdfInputData` and `sdfOutputData` fields of SDF's `actionqualities` can be mapped to WoT TD's `input` and `output` `ActionAffordance` fields, respectively. In the process, the associated SDF data qualities are WoT data schemas. As there is no direct equivalent for `sdfData` in `ActionAffordances`, we map this field to the `schemaDefinitions` of the resulting TM, using the action name as a prefix in order to avoid name collisions if multiple `sdfData` definitions should be present.

Three WoT-specific fields with no equivalent in SDF are `safe`, `idempotent`, and `synchronous`, which need to be mapped to a mapping file when converting actions to SDF.

3.2.5.3 Events

An `sdfEvent` can contain an `sdfOutputData` field, which we map to the `data` field of WoT TD's `EventAffordance` class, once more converting SDF data qualities to a WoT data schema. However, WoT's events provide additional fields for providing data schemas for message formats, namely for `subscription` and `cancellation` messages as well as for `dataResponses` given by the consumer. If present, these fields need to be mapped to a mapping file to be used in SDF. For the next major version of SDF, it is worth discussing if definitions like these should also be included in the SDF vocabulary.

10 https://github.com/w3c/wot-thing-description/issues/1541 (retrieved: June 16, 2022).

3.2.6 References

Mapping becomes a bit more complex when it comes to the remaining common qualities: All `sdfRequired` fields have to be mapped the top-level `tm:required` field of a Thing Model [wot-td, section 10.3.4]. For `sdfRef`, another distinction can be made: If the reference points to the same SDF model, the JSON pointer itself can be converted so that it points to the converted target (see Appendix A.2 for a simple conversion example). If an `sdfRef` is referencing another model, then the pointer cannot be properly converted and has to resolved before the conversion. The same holds for referenced top-level `sdfProperty`, `sdfAction`, and `sdfEvent` definitions as they do not belong to an `sdfObject`.

3.2.7 Mapping of Additional Properties

Using JSON-LD `@context` extensions, TMs are quite flexible when it comes to adding additional vocabulary. The additional terms, however, should be prefixed to refer to a `@context` entry, forming so-called compact IRIs in the process. WoT TMs allow the inclusion of additional properties in their definitions via context extensions.

While namespaces exist in SDF which could also be used for vocabulary extensions, there is currently no specified way of integrating additional properties in SDF qualities. Instead, mapping files have to be used, where additional properties from WoT documents can be mapped to. In the other direction, additional definitions in mapping files can be mapped to prefixed definitions in the resulting WoT document.

3.2.8 WoT-specific Mappings

In this section, we will describe how we handle the features supported by WoT TMs which currently do not have an equivalent in SDF and therefore either need to be resolved before the conversion or become part of an SDF mapping file.

3.2.8.1 Security Schemes

In WoT, there are a number of pre-defined security schemes, which can be included to indicate which kind of security mode shall be used in order to interact with a

Thing.[11] While the actual credentials used are instance-specific, security schemes can be defined for classes of devices and, therefore, at the Thing Model level using the `securityDefinitions` key with a map of security definitions as the value. In a `security` member, only the keys used in the `securityDefinitions` map can appear, applying the corresponding security definition either at the top level as the Thing's default or the Form-level for a specific affordance.

SDF currently does not allow for the definition of security information at all. Therefore, we need to include this information in a mapping file. Once such a vocabulary is available for SDF, the mapping needs to be updated accordingly.

3.2.8.2 TM Extension Mechanism

WoT Thing Models support an extension mechanism which is comparable to class inheritance in object-oriented programming languages. Similar to the reference mechanism (using `tm:ref`, see section 3.2.6), TMs can indicate that they extend another TM by including a link object with the relation type `tm:extends` in their `links` container, pointing to the extended TM's location in the `href` field. In contrast to references, this URI cannot contain a JSON pointer [RFC6901] as a fragment identifier, but has to point to the document itself.

While the extension mechanism can be reproduced for individual SDF containers using an `sdfRef` in an `sdfObject` or `sdfThing` pointing to another definition of the same kind, the semantics are not exactly the same, given that WoT TMs also apply the extension to metadata which is located at the model level in SDF. Another challenge for the mapping of this mechanism lies in the fact that the extended TM itself cannot be referenced by an SDF model without prior conversion, as this will otherwise lead to an invalid consolidated SDF model.[12] Therefore, in our mapping, extensions in WoT TMs must be resolved prior to the conversion, as we described in the case of external references in section 3.2.6. This prevents TMs with an extension link from being roundtrippable, which is undesirable, but unavoidable with the current SDF vocabulary. Alternatively,

11 The WoT TD specification has borrowed these schemes in part from OpenAPI (https://swagger.io/specification/ (retrieved: June 16, 2022)). They will probably be reworked in the next major version of the WoT TD specification, as they are in part incorrectly specified and too HTTP-specific in general.

12 Referencing documents from other ecosystems out of SDF models might be an interesting discussion point for future work on SDF.

the extension link could not be resolved but added to a mapping file instead, which would preserve the possibility to regenerate the original TM. However, this would mean losing the ability to actually use the definitions from the extended TM in the resulting SDF model. For these reasons, we decided to resolve TM extensions in our mapping by default, while allowing for the inclusion of the original `tm:extends` link in a mapping file as an alternative. A more direct mapping of the extension mechanism to SDF is probably not possible since it would also require a translation of TM semantics.

3.2.8.3 Links

In contrast to WoT, the current SDF draft does not specify a way for modelling links (i.e. web links as defined in [RFC2288]) in its vocabulary. Except for a few special cases like composition (section 3.2.1), the extension mechanism (section 3.2.8.2), and licenses (section 3.2.9.1), we treated links as generally not directly mappable to SDF, which is why we included them in a mapping file instead. However, with relations, the aforementioned [I-D.-sdf-relations] draft defines a possible SDF equivalent for WoT links, which should be integrated in future versions of mappings between WoT and SDF.

3.2.8.4 Placeholder Maps

WoT TMs allow defining placeholders which can be replaced during a conversion process using a so-called placeholder map. Placeholders use the format `{{PLACEHOLDER_NAME}}`, wrapping the name of the placeholder inside curly braces. During conversion, the placeholder as a whole (including the braces) is replaced with the value specified in the placeholder map. As the WoT TD specification defines placeholders to only be usable within string values, fields which are supposed to have a numeric or complex value need to be temporarily typed as string.[13] As there is no equivalent for placeholder maps in SDF, yet, they are currently not supposed to converted but to be replaced before the actual conversion process.

13 An example would a data schema like `{"maximum": "{{MAX_VALUE}}"}`, where `maximum` must have a numeric value after the placeholder replacement. Due to the limitations of the JSON format, the placeholder cannot be used directly as a value without wrapping it in a string. As the converter knows the target data type from the value inside the placeholder map, however, we can avoid adding strings in places where we expect a non-string-based value, by using the placeholder value directly if the string containing the placeholder name does not have additional padding whitespace.

3.2.9 SDF-specific Mappings

Both SDF and WoT allow for certain metadata in definitions. This subsection deals with the mapping of global (i.e., model-related) and definition-level metadata.

3.2.9.1 Info Block

The optional information block of an SDF model can contain metadata (namely a title, a version, copyright information, and/or a license). These fields are optional, so an information block can be empty or, for instance, only contain a title. As an info block's title applies to the SDF model as a whole, we map it to a prefixed field labelled `sdf:title` in each resulting TM, as the `title` of individual TMs corresponds to the `label` of the original `sdfObject` or `sdfThing` (see the next section). Similarly, we map the copyright and license fields to their own `sdf:`-prefixed fields. However, if the license should be a URL, we include it as a link with a relation-type of `license` [RFC4946] in the resulting TM instead. Finally, the `version` field can be mapped to the `model` field of the existing `VersionInfo` class [wot-td11, section 5.3.1.6]. Examples can be seen in Listings 3.5 and 3.6.

As it is unclear which information should be used for the info block when converting multiple TMs to SDF at once, we decided for our converter implementation that users should be able to explicitly set the info block if they desire to do so.

Listing 3.5: Example for the Information Block of an SDF model. This example does not use URIs for the copyright and license fields. Instead, a regular copyright statement and an SPDX license identifier are being used.

```
1  {
2    "info": {
3      "copyright": "Copyright 2019 Example Corp. All rights reserved.",
4      "license": "BSD-3-Clause"
5    }
6    "sdfObject": {
7      "Example": {
8        "label": "Example Object"
9      }
10   }
11 }
```

Listing 3.6: Mapped TM created from the SDF example in Listing 3.5.

```
{
  "@context": {
    "http://www.w3.org/ns/td",
    {
      "sdf": "https://example.com/sdf"
    }
  },
  "version": {
    "model": "2019-04-24"
  },
  "sdf:title": "Example file for OneDM Semantic Definition Format",
  "sdf:copyright": "Copyright 2019 Example Corp. All rights reserved.",
  "sdf:license": "BSD-3-Clause",
  "title": "Example Object"
}
```

3.2.9.2 Common Qualities

SDF defines a number of common qualities which can be used in each of the seven main SDF classes. Of these qualities, the `description` and `label` qualities are the ones which can be mapped in the most direct way (with the only difference that the equivalent of `label` is `title` in the WoT TD specification). There is no explicit field for comments in WoT TMs, however, which is why SDF's `$comment` has to be mapped to a prefixed `sdf:comment`. Finally, the common qualities also include the `sdfRef` vocabulary term, whose mapping we already described in section 3.2.6

3.2.9.3 Top-Level Affordances and sdfData

SDF models can contain affordances (`sdfProperty`, `sdfAction`, and `sdfEvent`) as well as `sdfData` definitions at their top-level. As these definitions only have the purpose of being referenced by other definitions within `sdfObjects` or `sdfThings`, we cannot map them directly to a TM. Instead, we dereference all of these top-level definitions and then convert the resolved results.

This has negative impacts when it comes to roundtripping, as it is not possible to reconstruct the original SDF model this way. An alternative to the current approach would be converting all top-level definitions to their own TM, which we could then reference from the other resulting TMs. Using a special keyword, we could then identify the TM as one containing top-level definitions and process it accordingly. Due to the fact that we discussed this mapping approach too late in the process of writing this thesis, we could not incorporate it into the actual mappings chosen for our converter.

However, both future mappings and converter versions should consider using it instead of the current approach for a more precise mapping with comprehensive roundtripping support.

3.3 Mappings between WoT TMs and WoT TDs

In order to achieve a mapping between SDF and WoT Thing Descriptions, we now need to define a mapping between TDs and TMs. As described above, we can then first convert a TD to a TM and then to an SDF model and vice versa, achieving a complete mapping between all three kinds of documents in the process, with WoT TMs serving as an intermediary between TDs and SDF models. As TDs contain mandatory instance-specific information which does not have an equivalent in SDF (most notably forms at the affordance level as well as security definitions at the top-level), the conversion process between SDF and WoT TD will always result in an additional mapping file and will take at least one as an input, respectively.

For the conversion from TMs to TDs, we can rely on the pre-defined algorithm in the TD specification [wot-td11, section 10.4]. However, for the Thing Model Collections introduced above, we need to slightly alter the conversion process, converting each contained TM to a TD in the process, resolving extensions and references. If a TM links to a submodel, we also need to convert the linked TM and all of its submodels (et cetera) to TDs. All the results become part of a Thing Description Collection, maintaining the relationship between the original Thing Models after the TD conversion using JSON pointers with the corresponding TDs in the collection as their target location. In order to be able to recreate the original links, we add a link with relation-type `type` [wot-td11, section 10.4] to the resulting TMs, pointing to the original URL (if available).

The other direction, converting TDs to TMs, is not as well-defined, yet. However, due to the fact that TMs are very close to being supersets of TDs, we can turn every TD that does not have an `item` or a `collection` link (which can be used for nesting in TDs) into a TM by simply adding the string `tm:ThingModel` to the top-level JSON-LD `@type` field. In the case of nested TDs (containing `item` links that point to another TD), we also need to convert the linked TDs to TMs and either create a collection output, linking them with JSON pointers, or export them to the file system or a different source,

creating a file or web link in the process.[14]

One open question in this regard for future versions of the WoT TD specification is if the link relation-type actually needs to be changed from `item` to `tm:submodel` as both are quite similar in their semantic meaning. For now, we have followed the strategy of converting the relation-type; we could imagine as an alternative, however, that the `instanceName` field simply becomes an additional vocabulary term for links with the `item` relation-type in Thing Models.

With a complete mapping between SDF and both WoT TMs and WoT TDs, we started formulating the requirements for our converter, which can be found in the next chapter.

14 Note, however, that even with the original location linked in the resulting TD recreating the original structure of TMs might not be possible in the case of external links if a TM is not available anymore as a resource. This is a potential drawback of the linking approach that is not present in SDF when it comes to nesting. However, in the case of imports using `sdfRef` and `tm:ref` both specifications face the same problems.

4 Implementation Requirements

While the mappings outlined in the last chapter are generally enough for carrying out a conversion between two data modelling formats manually, we want to be able to perform conversions automatically and on scale, using a dedicated software—a converter—to do so. In this chapter, we will outline the functional and non-functional requirements for our converter, which we will translate into a concrete converter design in chapter 5. These requirements do not only take our mappings into account, but also the previously discussed foundations and the possible deployment scenarios.

4.1 Functional Requirements

The most important requirement for our converter is to provide a comprehensive mapping between SDF and WoT TD. That means that our converter should be able to translate every field that is contained in an SDF model or a WoT definition into an equivalent definition in the other format. This should also include definitions which are either not defined in one of the two specifications or originate from vocabulary extensions. In the case of SDF, these additional definitions should be included in so-called Mapping Files, as defined in [I-D.-sdf-mapping].

An important criterion for evaluating both the comprehensiveness and the accuracy of a mapping is the ability to roundtrip the conversion process, i.e., to convert a document from either of the two specifications into the other one and receive the same document when applying a conversion into the other direction. The roundtrip potential should therefore be considered both during the development of the mappings themselves and as additional test cases for the actual converter implementation, ensuring that as many mappings as possible are reversible.

Besides the conversions between SDF and WoT, the converter should also support the conversion of WoT Thing Models into WoT Thing Descriptions, and vice versa. This way, users are able to first create reusable WoT documents from SDF models, which they can then instantiate on demand. Conversely, they can create generalizations in the form of Thing Models from Thing Descriptions, although the resulting TMs will contain instance specific information that might have to be removed manually by the user after the conversion process.

Both specifications have mechanisms for referencing and (in the case of WoT TMs) extending models, which should also be able to be resolved in order to create consolidated documents before conversion, as documents conforming to the other specification cannot be directly referenced. However, this might be at odds with the roundtripping mentioned above, as resolved references cannot be converted back into a reference without making it explicit, which definitions originate from another document.

Moreover, the converter should be able to validate both conversion inputs and outputs on the basis of the pre-defined schemas (using either CDDL [RFC8610] or JSON Schema [I-D.-jso-draft-7]) that are included in both specifications.

The second most important requirement is that the converter can be used in as many deployment scenarios as possible. Similar to Kiesewalter's converter, our implementation should be usable both as a CLI tool and as a web application. The CLI tool should enable users to both call the converter from a terminal and integrate it in scripts, while the web application should offer a simple interface for converting between SDF and WoT. However, the converter should also expose its conversion logic as a library for the ecosystem/programming language of our choice, making it possible to reuse it in other implementations of the same or a neighboring ecosystem, such as code generators.

Another aspect that increases the number of scenarios users can use the converter for is to allow for multiple sources for input files. The library and the CLI tool should allow for importing documents both from the local file system and from an external URL, supporting at least the Hypertext Transfer Protocol (HTTP). Furthermore, the library should also allow for converting models both in serialized form (i.e., a JSON text) and as a deserialized data structure. The minimal requirement for the web application is to accept inputs via HTML forms (using an HTTP POST request in the process), a slightly more advanced optional requirement is the definition of a REST API, which exposes the conversion functionality as explicit HTTP resources, enabling the selection of the desired conversion via URI template variables [RFC6570].

Finally, the converter as a whole should work in a wide variety of environments, supporting as many operating systems (e.g., Linux, macOS, and Windows) and types of devices as possible. This should include single-board computers like Raspberry Pis, but not necessarily constrained devices, as JSON—the serialization format of both SDF and WoT—is not very well suited for constrained environments.

4.2 Non-Functional Requirements

In general, our converter is not supposed to be explicitly designed for performance critical scenarios, requiring a high number of conversions per second, but rather as a tool operated semi-automatically by humans. Therefore, we do not require our converter to operate at high speed, while the conversions should also not be performed noticeably slowly.

A similar requirement can be formulated for safety and security: While the converter should avoid security and safety issues wherever possible, we do not require it to explicitly fulfill ISO or comparable standards. However, if both WoT and SDF become more mature specifications and therefore more widely used, e.g., in industry scenarios, it might be desirable for a converter to be certified by authorities like ISO in the future. As more concrete security requirements, we can demand the support of HTTPS when fetching referenced or extended documents, as well as when running our converter as a web application. The converter should also avoid well-known security risks and support (only) versions of the chosen programming languages with long-term support, ensuring that its deployment does not pose a potential security risk.

Another important non-functional requirement concerns the converter's usability. Both the CLI tool and the web application should provide easy-to-use interfaces, offering help messages and comprehensible error messages. The web application should also be designed in a manner that is reasonably appealing, while also supporting smaller screen sizes. The library itself should be well documented, making it easier to use for third-party users, describing the functionality of all outwards facing functions documentation. Optionally, the internal functions should also be documented to make it easier for outside contributors to become familiar with the code base.

Finally, the converter should be easy to install and added to third party projects, e.g., by

being included in a package repository associated with the programming language/ecosystem chosen.

5 Design

In this chapter, we describe how we designed the converter to fulfill the requirements defined in chapter 4. This includes an implementation of the mappings defined in chapter 3.

5.1 Structure

Because our converter as is supposed to be usable both as a library, a CLI tool and a web application, we need to expose three different kinds of APIs.

First, we need a "library" module which exposes interfaces to the actual conversion logic in order to be used by other applications or libraries written in the same or a compatible language. The library API allows the conversion from one of the three possible formats (SDF, WoT TM, and WoT TD) to one of the other formats. Submodules of the library contain the underlying converter logic, as well as functions for validating inputs and outputs.

The CLI tool and the web application use the library as any other application would, building upon its public API. They also perform input and output operations, as well as JSON serializing and deserializing. However, the CLI tool is supposed to be an additional part of the library which is installed alongside it. Running the library as a program from the command line calls the CLI tool as an executable.

The web application is a separate component or library, which exposes a graphical user interface via a web page, running in the browser. In the backend, the web application calls the library's functions on user submission, transforming the given input with the conversion logic, and displaying both the original input and the result. HTML forms contain both the input and output, enabling subsequent editing and converting of input models, as well as experimenting with roundtripping.

After this high-level view on the structure of our converter, we take a closer look at different aspects of our design in the following sections.

5.2 Components

In this section, we describe the components that make up our converter as a whole in more detail.

5.2.1 Library

We divide our library module into an outwards facing API and internal submodules which implement the actual conversion logic, provide utility functions, and contain schemas for validation. Furthermore, we define another module that lives besides the actual library contains unit and integration tests, asserting that the converter meets both the requirements and conforms to the mappings defined in chapter 3

As we have three kinds of data formats, we need one type of function for each conversion direction for our external API. That means that we need six functions in total, converting from

- SDF to WoT TM,
- SDF to WoT TD,
- WoT TM to SDF,
- WoT TD to SDF,
- WoT TD to WoT TM and
- WoT TM to WoT TD.

Despite the fact that the CLI and the web application will provide flexible input methods (i.e., JSON texts, which might originate from the file system or the internet in the case of the CLI) these exposed library functions only need to accept and return deserialized formats. This simplifies the implementation, leaving input and output operations, as well as serialization and deserialization to the library users (including our CLI and the web application).

As we have seen in chapter 3, the different conversion directions have different input and outputs, which also influences the parameters of our conversion functions. When it comes to SDF, both SDF to WoT functions require an SDF model as a mandatory argument. Furthermore, they accept an arbitrary number of SDF mapping files, which will be used to augment the SDF model during the conversion process. Depending on the complexity of the input model, the output is either a single TM or TD, or a collection of TMs or TDs.

5.2.1.1 Internal Modules

In general, the internal structure of the library can be chosen freely by implementors as they see fit, as it should not affect its external API. In order to achieve a separation of concerns, however, it can be advised to divide the library into at least two modules:

1. a Converter module, containing the logic for mapping between the three formats,
2. a Validation module, which contains validation functions using the JSON Schema or CDDL definitions for WoT TD s and TMs as well as SDF models and mapping files, and
3. optionally, the CLI logic, which will only be exposed when calling the converter as an executable.

The converter module itself can further be split into at least four submodules, as we need only to cover the conversion between SDF and WoT TM as well as WoT TM and WoT TD. As WoT TMs can be used as an input for two of these submodules, a fifth submodule can be used to factor out functions usable for both TM to SDF and TM to TD conversion. Furthermore, common utility functions as well as mappings of the JSON schema definitions used in both specifications can optionally be factored into their own submodules.

Based on these considerations, we created the internal module structure depicted in Table 5.1, which should be adaptable for most implementation scenarios. While not a part of the library API, we also included the CLI in this overview, as its internal logic

1 The SDF Framework schema is a more liberal version of the SDF Schema, which also allows for extension points, e.g., augmented SDF models with one or more mapping files applied to them.

Table 5.1: Internal module structure of our converter library.

Module	Submodules
Converters	SDF to TM TM to SDF TM to TD TD to TM WoT Common JSON Schema Qualities Utility
Validation	Validation Functions TD Schema TM Schema SDF Validation Schema SDF Framework Schema[1] SDF Mapping File Schema
(Command Line Interface)	—

will be *part* of the library, even though it should only be accessible when users call the converter from the command line.

5.2.1.2 Testing Module

Besides the library itself, we define a module for performing unit and integration tests. This testing module ensures that the mappings defined in chapter 3 are implemented correctly and that the implementations meets all (testable) requirements. Besides testing the mappings defined in chapter 3, the module is supposed to also assert the correct implementation of the Command Line Interface, which will be described in the next section.

5.2.2 Command Line Interface

We designed our CLI tool to build upon and live alongside our library, making it possible to install both, for example, with the same package manager or dependency management system. In general, the CLI exposes a set of sub-commands after being called with the

command `sdf-wot-converter`. The sub-commands correspond directly with the six conversion functions defined by our library API:

- sdf-to-tm
- sdf-to-td
- tm-to-sdf
- tm-to-td
- td-to-sdf
- td-to-tm

Using parameters and flags, we designed the sub-commands to accept the same set of arguments as the conversion functions themselves. The inputs (which must be serialized as JSON) can originate both from the file system by providing a corresponding path or from a web server, using a URL with the schemes `http://` or `https://`. Providing output paths, however, is optional—leaving them out is supposed to cause the CLI to write the results to the standard output. Otherwise, it serializes the results to JSON and writes them to the provided file paths.

Our design offers a number of additional explicit parameters, which are displayed in Table 5.2. On the one hand, they include general options, namely the indentation depth for the formatting of the resulting JSON documents (which defaults to four spaces) and an option for suppressing additional roundtripping definitions. On the other hand, we define a number of format-specific options.

When converting from SDF, users can pass an arbitrary number of mapping files alongside SDF models and an optional origin URL which points to the original document, when converting a model to a WoT TD or a WoT TM.[2] When using a TM as an input, users can specify placeholder maps and/or JSON documents with additional metadata or bindings information.

2 We added this parameter after receiving feedback on the initial version of our converter and the conversion results from the OneDM playground. See chapter 7 for more details.

Table 5.2: Overview of the available parameters for the sub-commands of our CLI.

Parameter	Arguments	Sub-Commands	Mandatory	Default
`--input, -i`	File path(s) or URL(s)[a]	all	✓	—
`--output, -o`	File path	all		—
`--suppress-roundtripping`	—	all		False
`--indent`	Natural number	all		4
`--origin-url`	URL	sdf-to-tm, sdf-to-td		—
`--mapping-files`	Zero or more file paths	sdf-to-tm, sdf-to-td		—
`--title`	String	sdf-to-tm, sdf-to-td		—
`--version`	String	sdf-to-tm, sdf-to-td		—
`--copyright`	String	sdf-to-tm, sdf-to-td		—
`--license`	String	sdf-to-tm, sdf-to-td		—
`--meta-data`	File path or URL	tm-to-sdf, tm-to-td		—
`--bindings`	File path or URL	tm-to-sdf, tm-to-td		—
`--placeholder-map`	File path or URL	tm-to-sdf, tm-to-td		—
`--mapping-file-output`	File path	tm-to-sdf, td-to-sdf		—
`--remove-not-required-affordances`	—	tm-to-td		False

[a] Can be multiple paths/URLs when converting from WoT TM/TD – the imported TMs/TDs are then treated as Collections.

5.2.3 Web Application

Following the design of Jana Kiesewalter's approach for her sdf-yang-converter[3], we chose to provide two input fields as part of an HTML form in our web application. Depending on the conversion direction, the web page displays the output in one of the two fields, while also keeping the original input in the other. This makes it possible to experiment with multiple inputs in a row, as well as with roundtripping.

However, as we do not have two but three possible input and output formats, we designed our web application to have a dropdown menu to be able to select the two formats for the actual conversion. Similarly to Kiesewalter's web application, we included the possibility to load at least one example for each format in our design. Furthermore, to make editing the input—which must be formatted as JSON—easier, we chose to include buttons for auto-formatting and clearing the contents of the input fields. The inclusion of additional roundtripping fields as well as SDF mapping documents in the respective outputs can be toggled via two checkboxes.

Finally, we also designed our web application to provide a single REST API resource, which allows for sending HTTP POST requests with a conversion input encoded as JSON, which is then converted and returned as a JSON-encoded payload. To do so, the web application exposes the resource `/convert/<command>`, where the legal values for the URI template variable `command` correspond with the six available conversion functions.[4]

3 http://sdf-yang-converter.org/ (retrieved: June 9, 2022).

4 That is `sdf-to-tm`, `sdf-to-td`, `tm-to-sdf`, `tm-to-td`, `td-to-sdf`, and `td-to-tm`.

6 Implementation

This chapter describes how we implemented our converter on the basis of our design decisions made in the last chapter.

6.1 Technologies Used

For the implementation of our converter, we had a number of different programming languages to choose from, each having different trade-offs that needed to be considered. Due to the potential use of the converter in constrained environments, we first considered a strongly typed systems programming language like C++ or Rust. As they are compiled to machine code, they have a performance advantage over interpreted languages like Python or JavaScript, but also enforce a stricter type system onto the implementation, which might be counterproductive for fast development and the handling of JSON-based and therefore dynamically typed data formats.[1] Due to Python's flexibility, its

1 A prior approach to realizing the converter relied on the Rust programming language, defining data schemas and qualities in accordance with their specified data type. This led to the problem that the [I-D.-jso-draft-7] inspired schemas in both formats allow for unsatisfiable definitions (like a numeric constant in a string-based schema) which could not be easily covered in a strongly typed language like Rust. For this reason, we chose Python as an alternative, which makes it easier to deal with dynamic data structures. In hindsight, we could have solved this problem by switching from a strongly typed data structure to one relying entirely on more or less dynamically typed values provided by the serialization framework `serde` and its JSON implementation `serde_json`. At the time, however, we felt that using Python would have advantages for increased development speed, while relying on `serde_json` would run counter to the benefits we had hoped to gain from choosing Rust. On the basis of our Python implementation, it should be fairly easy, though, to reimplement the current state of our SDF WoT converter in Rust, which should bring a number of performance benefits and higher type safety.

In any case, the problems we faced with regard to the JSON Schema based vocabulary do indicate that future versions of both specifications should evaluate how flexible their type system should be. A stricter type system would probably make it a lot easier to implement both WoT TD and SDF in programming languages like C or Rust, which are the main candidates for the use in embedded and constrained environments.

library ecosystem, and also because of its high portability, providing support for all major operating systems as well as a wide variety of devices, including, for example, Raspberry PIs, we eventually chose Python as the implementation language, supporting the versions 3.7 through 3.10.

Another advantage of Python (that might also be true for other programming languages) is that its ecosystem already features libraries for all standards relevant for the processing of both SDF and WoT TD, namely JSON pointers[2] [RFC6901] and the JSON Merge Patch algorithm[3] [RFC7396]. There is also a library[4] for validating JSON documents based on a JSON Schema [I-D.-jso-draft-7], which is relevant for validating the inputs and outputs of the conversion, as both specifications provide informative JSON schema definitions corresponding to the normative contents. In the case of SDF, the JSON Schema originates from a CDDL [RFC8610] schema, which is—in contrast to [I-D.-jso-draft-7]—a standardized schema language that surpassed the draft status that still applies to JSON Schema. At the time of writing of this thesis, the library support for CDDL in the Python ecosystem was still in its early stages of development, which is why we made the decision to rely on a single, JSON Schema based solution for validation. Future versions of the converter will mostly likely be able to rely on a CDDL library like zcbor[5] for validation, making it possible to directly use the original schema format, increasing the reliability of the validation process.

Python also features frameworks for building tools for the command line as well as web applications. Due to prior experience and the fact that it comes included with the Python standard library, we chose argparse[6] for creating our command line tool, which has the benefit of not adding another dependency to our library. As our requirements for our web application were fairly minimal, considering the fact that we, for example, did not need to persist any kind of state in a database, we decided to use Flask[7] for this part of the implementation, which is a mature framework with a low overhead, featuring HTTP routing and a template engine, which makes it easy to insert conversion results in the resulting HTML documents. Flask also allows for creating resources with

2 https://pypi.org/project/jsonpointer/ (retrieved: July 1, 2022).
3 https://pypi.org/project/json-merge-patch/ (retrieved: July 1, 2022).
4 https://pypi.org/project/jsonschema/ (retrieved: May 5, 2022).
5 https://pypi.org/project/zcbor/ (retrieved: May 5, 2022).
6 https://docs.python.org/3/library/argparse.html (retrieved: June 9, 2022).
7 https://pypi.org/project/Flask/ (retrieved: June 9, 2022).

URI template variables, accepting and outputting JSON, which we could use to create a simple REST API for calling one of the converter functions.

6.2 Library Implementation

In order to be able to publish our library to the official Python package repository PyPI[8], we applied the wheel packaging standard[9] to our library using the setuptools[10] and wheel[11] packages. Our package setup resulted in a setup.cfg file which contains package metadata and defines the needed dependencies, so that they can be installed together with our library using Python's package manager pip.[12]

Our package structure follows our design outlined in the last chapter (see section 5.2.1): The top-level __init__.py file exposes the conversion functions defined in a submodule called converters[13], which we divided in turn into single Python files implementing specific aspects of the conversion process, also corresponding with our design (i.e., tm_to_sdf.py, tm_to_td.py, sdf_to_tm, and td_to_td).

6.2.1 Internal Conversion Functions

In general, our converter follows a functional approach, modelling each step of the conversion process as an individual function, which might in turn call other functions. As Python is a multi-paradigm programming languages, which also supports object-oriented programming, we could have also created one or multiple converter classes for handling the conversion. This would have allowed us to perform the conversion statefully, keeping track of, for example, retrieved submodels or additional fields (which

8 https://pypi.org/ (retrieved: May 19, 2022).

9 https://peps.python.org/pep-0427/ (retrieved: May 19 2022).

10 https://pypi.org/project/setuptools/ (retrieved: May 19, 2022).

11 https://pypi.org/project/wheel/ (retrieved: May 19, 2022).

12 Having pip installed, you can install the latest version of our converter by typing pip install sdf-wot-converter in a terminal of your choice. The converter version this thesis is referring to is 1.9.0, which is corresponding with the commit hash acc1b52e4834b84a7193f5de8e26f72bd3cae0ce in the Git repository hosted under https://github.com/JKRhb/sdf-wot-converter-py (retrieved: July 1, 2022).

13 The function names follow the pattern convert_<input-format>_to_<output-format>, for example convert_wot_tm_to_sdf.

Listing 6.1: Code of our main conversion function `map_field`.

```
def map_field(
    source_definition: Dict,
    target_definition: Dict,
    source_key: str,
    target_key: str,
    conversion_function=None,
    mapped_fields=None,
):
    """Maps a field from a source definition to a target definition, applying a
    conversion function if given."""
    source_value = source_definition.get(source_key)

    if source_value is None:
        return

    if conversion_function is not None:
        source_value = conversion_function(source_value)

    if mapped_fields is not None:
        mapped_fields.append(source_key)

    target_definition[target_key] = source_value
```

need to go into a mapping file) within the instantiated class object. We decided against an object-oriented approach in favor of a functional one, as it simplified our libraries external and internal APIs, and improved the testability of our implementation due to the omitted global state. However, this decision came at the disadvantage of carrying over the state of the conversion in our functions' parameters, which also lead to long argument lists in some places.

In general, we use Python *dictionaries*, which are hash maps containing key-value-pairs, for modelling input and output data. Dictionaries are very similar to JSON objects (with one major difference being that the keys of dictionaries can not only be strings, but any *immutable* datatype), which allowed us to perform direct serialization and deserialization with Python's `json` module, which is also part of the standard library. Dictionaries also have integrated methods which return iterators, allowing to conveniently iterate over a dictionary's keys and values using `for` loops. This way, we can perform mapping actions for all properties of an `sdfObject`, for example.

As a general mapping strategy, we first validate the inputs to the respective conversion function with the corresponding JSON Schema definition. We then resolve all import-s/references as well as composition and extension links, also validating all retrieved documents in the process. Besides making sure that we only process valid inputs, we can also make assumptions about the given structure of the dictionary (e.g., the presence of mandatory fields), further outweighing possible disadvantages of not using an explicit

data model for deserializing the input data.

We carry out most of our conversions using a helper function called `map_field` (see Listing 6.1), which generalizes the conversion from a `source_definition` to a `target_definition`. The function first checks if the specified `source_key` is present in the `source_definition`. If not, we abort the conversion with a `return` statement, leaving the function. If the key is present, we access the corresponding value.

As a next step, the `make_field` function can call a custom `conversion_function` (if present), which can also be passed as an argument, to further transform the input. At the moment, we only use this feature to negate boolean inputs[14], future versions of the converter, however, might make use of this argument for more complex conversions as well.

Before writing the resulting `value` to the `target_definition` at the `target_key`, we add the `source_key` to an optional list of `mapped_fields`, which can also be passed to the function. This list keeps track of all field names that are ignored when mapping additional fields (either directly to a TM or to an SDF mapping file). This is one simple example of how we maintain the state of our conversion process through additional function arguments, as mentioned above. We perform the mapping of additional fields after we have converted all "well-known" fields, iterating over the current definition and skipping all fields names which are present in the list of `mapped_fields`.

The overall process for all conversions is a recursive one: We start at the top-level of the document which want to convert (e.g., a Thing Model). We then access all well-known definitions at that level by value and call the respective conversion function, which adds the key to the list of mapped fields. If the value of a definition is a JSON object (e.g., an action) itself, then we repeat the process with the conversion functions defined for this type of definition. After the pre-defined conversions have finished, we iterate over all key-value pairs of the current definition and treat all of those whose key is not present in the list of already mapped fields as additional definitions, mapping them to, for example, to an SDF mapping file.

In the following subsections, we will discuss the details of the four concrete conversion functions and their underlying implementation logic in more detail.

14 For the conversion of `readOnly` and `writeOnly` to `writable` and `readable`, to be precise.

6.2.1.1 TM to SDF

The file tm_to_sdf.py contains all the code for the implementation of the WoT TM to SDF conversion and exposes its logic using a function called convert_wot_tm_to_sdf, accepting both single TMs and TM collections as inputs. Depending on the actual input, the function converts single TMs directly to an SDF model, after resolving all references and extensions as well as placeholders (if there are any defined within the TM). In the case of TM collections, we first determine which TMs are supposed to be treated as the top-level TMs, which in turn are going to become the top-level sdfThings and sdfObjects. For this purpose, the library user can either specify the top-level TMs' keys in the collection or—by default—let the converter determine the top-level models.

For determining the top-level models, we implemented a function which iterates through a TM collection and adds the keys of all TMs which are being referenced by at least one other TM within the collection (using links with tm:submodel or item) to a set. All TMs whose key does not end up in the set (and are therefore not being referenced by any other model) are considered top-level TMs.

All vocabulary terms with a defined mapping to SDF (see chapter 3) are mapped directly to the resulting SDF model. We insert any additional vocabulary terms, as determined with the mapped_fields parameter mentioned above, into a mapping document. To be able to do so, we keep track of the current SDF path as we iterate through the structure of (potentially) nested TMs: When we enter the map_properties function, for example, and iterate through all properties present, we append a sdfProperty/<propertyKey> string to the path of the parent sdfThing or sdfObject, where <propertyKey> is the property's key in its containing JSON object. The resulting paths become a JSON pointers and serve as the keys for those definitions with additional properties in the resulting mapping document. If the resulting mapping document contains at least one entry, then we return a tuple of both the SDF model and the mapping document. Otherwise, the function only returns the SDF model.

Note that during all steps of the conversion process, validation of both inputs and outputs using the JSON Schema takes place: We validate both single TMs and all entries of TM collections, as well as every TM retrieved for resolving imports or extensions. After the conversion, we validate the resulting SDF model in order to ensure that we only return a valid result. We currently do not validate SDF mapping documents, as the Internet-Draft [I-D.-sdf-mapping] that specifies them does not provide a JSON Schema

definition yet. However, once such a Schema becomes available, the addition of an additional validation step will be relatively simple.

6.2.1.2 SDF to TM

For the conversion from SDF to WoT TM (in the file `sdf_to_tm.py`), we follow the same general strategy as outlined above and first validate all input documents using the validation functions defined for SDF. We then apply all given mapping files to the SDF model, creating a "consolidated" model, which will serve as the sole input for the rest of the conversion process. As this augmented SDF model might contain additional properties, we validate it with the SDF Framework Schema, for which we also defined its own validation function.

In contrast to the conversion from WoT TM, we do not resolve all sdfRef references before the conversion, as SDF defines no additional extension mechanism as it is the case for WoT TMs. Therefore, we can resolve `sdfRef` imports as we iterate through the SDF model, reducing the complexity of the respective `resolve_sdf_ref` function. Similarly to our implementation of the resolution of WoT references, we need to keep track of references we have already encountered, in order to avoid infinite loops when following subsequent `sdfRefs`. We do so by defining a list which is passed continuously to every recursive call of our `resolve_sdf_ref` function. If a reference is already present in this list, we raise an exception, aborting the conversion process.

In general, our conversion logic follows our mapping defined in chapter 3. One of the most important distinctions we need to make here is the one between `sdfObject` and `sdfThing` definitions. While both are mapped to TMs, we need to replicate the (potentially) nested structure of an `sdfThing` by also converting every child `sdfObject` and `sdfThing` to a TM and reestablishing the hierarchical relationship via a `tm:submodel` link. We implemented this aspect by using the current definition path within the SDF model (e.g., `sdfThing/foo/sdfObject/bar`) as a key in the resulting Thing Model Collection.

This way, we can make sure that there are no conflicts in the resulting Thing Model Collection, in case any two `sdfObjects/sdfThings` have the same key within the SDF model. If the resulting Thing Model Collection only contains one entry, we return the single TM instead of the whole collection.

6.2.1.3 TM to TD

When converting from WoT TM to WoT TD, we have to distinguish once again between single TMs and TM Collections.

When it comes to single TMs, we first perform an input validation using the TM JSON Schema definition and then follow the algorithm detailed in the WoT TD specification, first resolving extensions and imports, and then applying additional metadata and binding information. Since the format of this additional information is not well-defined yet[15], we decided to simply apply the inputs for these two parameters with the JSON Merge-Path algorithm to the input TM using the `json_merge_patch` package. Therefore, both arguments also need to follow the general structure of the TM at the moment to be applied correctly. We then remove the value `tm:ThingModel` from the TM's `@type` field, and replace all placeholder entries if a placeholder map is present as a function argument. Finally, we adjust the `version` object of the TM (if defined), copying the value of its `model` field to the `instance` field, as the latter is mandatory in TD instances. The resulting TD is then validated using the TD JSON Schema definition after making sure that all required affordances are actually present in the resulting TD. Using the optional parameter `remove_not_required_affordances`, users can let the conversion function remove all affordances which are not required from the resulting TD.

If a single TM should contain any `tm:submodel` links, we resolve them, creating a TM collection from the results. As detailed in chapter 3, this is a necessary step due to the fact that we cannot reference a TM from a TD. As required by the WoT TD specification, we replace the submodel links' relation-type with `item` during the conversion process.

When it comes to external submodels, we had to realize during the implementation that this process is a bit more complicated and poses challenges for ensuring that no name clashes are possible. Therefore, we weren't able to fully implement a resolution algorithm, which iterates over all key-value pairs, resolve the submodels of every TM in the collection and add the results to the collection. If the TMs which are subject to the conversion should resemble a nested structure of more than two levels, we would repeat this process through recursive function calls. However, adding this feature to the

15 We started a discussion (https://github.com/w3c/wot-thing-description/issues/1215, retrieved: June 28, 2022) on either explicitly defining or recommending such a format, but it has not become part of the specification yet.

converter implementation should not be an unbearable challenge to add to the next minor version of the converter, as we will discuss in section 7.1.3.

6.2.1.4 TD to TM

As described in chapter 3, the process of converting a single TD to a TM is rather trivial, as we only need to add a string with the value of `tm:ThingModel` to the TD's JSON-LD `@context` to receive a TM. If the single TD should not contain any `item` links, we only perform input and output validation besides this step,

However, if the input is a TD collection or the single TD should contain `item` links, we perform a potentially recursive procedure similar to the one mentioned in the previous subsection, collecting all subordinate TDs in the TD collection. For every TD in the collection, we then perform the conversion to a TM mentioned above, changing the relation-type of every `item` link to `tm:submodel` and performing both input and output validation. When it comes to this feature, we faced a similar problem as described in the previous section and could not finish the recursive resolution of external references before submitting this thesis. See section 7.1.3 for more details.

6.2.2 Testing

We defined our tests in a separate `tests` module, structured internally in accordance with the converter submodule of our library module. As our goal was to assert that the conversion logic follows the mappings we defined in chapter 3, we mostly wrote black-box tests, ensuring that the implementation produces results as we specified. Additionally, we defined tests for the CLI which verified that the converter can be used as intended from the command line.

For performing the tests, we used the `pytest`[16] package, which is especially useful for asserting that the contents of two dictionaries are equal, making it possible to compare a conversion result with the expected output. We performed the actual testing mostly on the basis of the mappings and the examples given in chapter 3, making sure that the mappings we are properly translated into the converter's actual logic.

16 https://pypi.org/project/pytest/ (retrieved: May 19, 2022).

During the development of our converter, we used a Continuous Integration (CI) pipeline which ran the tests we defined on every incremental change of our implementation, ensuring that our main development branch was always in a usable state. Using a built-in feature of `pytest`, we also constantly determined the test coverage of our library, making sure that the tests we defined completely covered the implementation code we had written. This was enforced by making a code coverage of 100 % a requirement for our CI pipeline to pass.

6.3 CLI Tool

As mentioned above, we implemented the Command Line Interface using the Python module `argparse`, which is part of the Python standard library and offers a convenient way to define and parse command line arguments. The library maps inputs that correspond with the pre-defined arguments to a dictionary, which in turn can be passed to other functions. This way, we created a CLI tool offering the API described in the previous chapter, as well as documentation which can be accessed by invoking the CLI tool with the `-help` or `-h` parameter. For each command and parameter, we defined a description, which the help page displays when called by a user.

Just as third-party libraries and applications, the CLI tool accesses the internal logic of the converter through the library API and the top-level functions defined for each of the six conversion modes. Depending on the sub-command passed by the user, the implementation calls the corresponding conversion function after reading the inputs from the file system or via HTTP requests. For example, if a user passes the sub-command `sdf-to-tm`, the converter calls the function `convert_sdf_to_wot_tm`, passing the parsed arguments (see Table 5.2 in the previous chapter) to it.

If no output path for any of the resulting documents should be provided by the user, we write a formatted string-representation to the standard output using the `pprint` module, which is part of the standard library. Otherwise, we write them to the file system at the specified location(s).

6.4 Web Application

We implemented the web application using the Python framework Flask in conjunction with the Jinja2[17] template engine, which allows us to integrate Python data structures and functions into HTML documents. A screenshot of the web interface can be seen in Figure 6.1. A continuously running version of the web application for demonstration purposes is available under https://sdf-wot-converter.herokuapp.com/ (retrieved: July 1, 2022).[18]

Due to its limited scope, we were able to create the application with a single page, which accepts input data from one of two text area HTML elements, which are also used for rendering the output data by inserting the conversion results into the text area which did not contain the input data. Using two dropdown menus on both sides of the converter, users are able to select which data model format should be used for the two text areas, while an additional settings section on the page allows for specifying options. At the moment, users are able to indicate whether a mapping file should be emitted during the conversion to SDF and whether additional fields for enabling roundtripping should be part of the conversion result.

Furthermore, users can insert examples for the current document types and let the converter format and clear the content of the input fields. We implemented this client-side logic using three simple JavaScript functions, which are triggered when a user clicks on one of the buttons on either side.

Using the (relatively) modern CSS layout features Flexbox and CSS Grid, we were able to achieve a responsive design, which makes the web version of the converter also usable on smaller screen sizes.

17 https://pypi.org/project/Jinja2/ (retrieved: June 16, 2022).

18 Similarly to the library and CLI tool implementation, the web application's source code is available in a Git repository https://github.com/JKRhb/sdf-wot-converter-py-demo (retrieved: July 1, 2022). The commit hash corresponding with the version presented in this thesis is `9573a07872de1b927c8d10dd43730ce538ff06d2`.

SDF WoT converter

SDF

»

«

WoT TM

SDF or WoT Document

SDF or WoT Document

Clear

Insert Example

Format

Clear

Insert Example

Format

Settings

Output SDF Mapping files

Include additional fields for roundtripping

Figure 6.1: Screenshot of our Converter's Web Interface.

7 Evaluation

With the finished mappings and implementation, we can now make a judgement on whether the requirements defined in chapter 4 are fulfilled. In order to so, we will evaluate the converter both qualitatively, based on these very requirements (section 7.1), and quantitatively (section 7.2). Finally, in section 7.3 we will evaluate how well the two specifications are aligned could be improved in the future to enable a better mapping between them and which improvements we have already contributed in the context of this thesis.

7.1 Requirements Evaluation

Due to the limited scope of this Bachelor's thesis, we haven't been able to perform a comprehensive survey for collecting feedback from both WoT and SDF experts.[1] Therefore, we will focus on criteria derived from our requirements in chapter 4. Future work building upon our thesis, however, should consider taking into account the feedback given from users of both WoT TD and SDF more thoroughly.

In the remainder of this section, we will discuss the requirements defined in chapter 4 and their fulfillment by our implementation.

---

1 Although we asked for feedback through various channels, such as direct email messages and comments in GitHub issues, we unfortunately received very little response to the finished converter implementation.

However, during the earlier stages of the implementation and writing process, we have been able to receive input from both the SDF and the WoT community, which we incorporated both into the mappings themselves and into the implementation. One example for such feedback include the mapping strategy of converting each `sdfObject` and `sdfThing` to a TM. In earlier versions of our thesis and our converter, we followed the approach of "squashing" all affordances, creating only a single TM as an output. Therefore, we can say that we were able to establish a feedback process, which, however, was not as formalized as we would have liked in order to be usable for this chapter.

7.1.1 Comprehensive Mapping

In general, we can conclude that our implementation is able to convert most elements of a WoT or SDF document to the other format. Following our mappings described in chapter 3, we can process all elements contained in a TD/TM and in an `sdfObject`/`sdfThing`, respectively. The fact that the latest version of SDF also allows for defining affordances at the `sdfThing` level helped us very much with the mapping between the two specifications and can be considered an important step for their alignment.

However, we still noticed limitations both in the specifications and our implementation, which required us to add workarounds such as the Thing Model/Description Collections. These enabled us to map a complete SDF model to a single JSON object, modelling hierarchical relationships via JSON pointers with same-document references. However, we also encountered some problems with the resolution of complex TD/TM Collections with external references when converting between TDs and TMs, which we discuss a bit more in section 7.1.3.

While TM/TD Collections also made it possible to implement complex conversions between TDs and SDF, using TM Collections as an intermediary, they are not a part of the WoT TD specification, yet, making this aspect of our mappings a potential extension point.[2]

The current use of SDF mapping files can also be considered a workaround, as there is no standardized way of describing instance-specific information like security metadata or protocol bindings at the moment. While there is ongoing work on adding new features like the `sdfRelation` quality to SDF [I-D.-sdf-relations], which will probably be usable for mapping WoT links to SDF, there are still a number of blank spots for which the addition of equivalents to SDF could be considered. These blank spots include the semantic `@type` annotations provided by JSON-LD, which also allows for defining context entries which are not mappable to SDF's namespaces at the moment, including, e.g., `@language` for setting a default language for the current document or single URLs entries for importing external context definitions.

2 We already discussed our approach with the WoT TD taskforce (https://github.com/w3c/wot-thing-description/issues/1407, retrieved: June 27, 2022). While we received positive feedback and the taskforce acknowledged the need for a document type like TM/TD Collections, the conclusion was to postpone this feature to version 2.0 of the TD specification.

As we can see, there is still potential for alignment in both specifications, which is also important for improving the roundtripping potential of conversion results.

7.1.2 Roundtripping

With the help of mapping files and TM/TD Collections, we have been able to achieve a high degree of roundtripping potential, which we ensured in our converter implementation by providing a number of test cases in which we are able to reproduce the original input. However, we also encountered a number of cases where it is not possible to reproduce an input in its entirety. These appear mostly in the context of both specifications' reference and extension mechanisms (namely `sdfRef`, `tm:ref`, and `tm:extends`) when referring to external models, which have to be dereferenced before being able to convert them to the other format, as we cannot, for example, refer to an SDF model from a WoT TM. As some definitions in an SDF model, especially top-level affordances and `sdfData`, only have the purpose of being referenced from elsewhere in a model, they can also not be converted a roundtrippable manner.

Due to the limitations discussed in the previous subsection, we also need to include the original `@context` in a mapping file when converting from WoT to SDF if we want to be able to reproduce the original format, potentially leading to duplicate entries that also appear in the `namespaces` field. The need for including the `@context` in a mapping file also results from the use of the JSON Merge Patch algorithm which does not allow for merging two JSON arrays, which is required in order to be able to only include the additional `@context` entries in the mapping file. Future and more refined ecosystem-specific mappings might need to take such cases into account and require a more sophisticated approach to creating augmented SDF models.

In the other direction, there are a number of special SDF qualities without an equivalent in WoT, which we decided to add with an `sdf:` prefix, such as the info block fields, the `uniqueItems` or the `$comment` qualities. For being able to properly add these additional fields to a WoT document, an extension of the WoT vocabulary should be considered. However, SDF should also define its own RDF-compatible vocabulary which could be integrated into WoT documents. This vocabulary could also contain auxiliary fields we defined to be able to match an original definition to its spot in the original SDF model, namely `sdf:objectKey` and `sdf:thingKey`.

Future versions of a mapping between WoT TD and SDF should further investigate how much of these additional definitions could be integrated into the core vocabulary and which should be treated as extensions.

7.1.3 Conversion between TDs and TMs

One very important requirement we have been able to mostly fulfill concerns the conversion between TDs and TMs. As we already discussed in chapter 3, TMs can serve as an intermediary between TDs and SDF models, due to their similarity to TDs while having less constraints when it comes to mandatory fields. During our implementation, however, we had to experience that the conversion of TM and TD *collections* with external references to submodels turned out to be more complicated than expected, as we need to resolve every referenced TM or TD before the conversion, adding each resolved TM or TD to the collection.

These complications caused this part of the functional requirements to be only partially fulfilled. However, it also touches the question of how to map TDs to TMs in general, where the WoT TD specification does not make concrete statements regarding the conversion process. This might also lead to possible inconsistencies when converting a TD with a subthing, modelled via an `item` link, to a TM—a discussion which will be continued in the WoT working group for TD 2.0[3].

7.1.4 Deployment Capabilities

Due to the fact that we chose Python as an implementation language, we are able to support all major operating systems, making it possible to use the converter both on a regular personal computer and on Single-board computers like Raspberry PIs. As we will see in section 7.2, the use of Python also seems as if it was not a major performance drawback, although future implementations should still consider using a language like Rust for an even better performance and support for platforms where no Python interpreter is available.

Since we have been able to implement our converter as a library, a CLI tool, and a web application, we have in fact been able to cover a lot of possible deployment scenarios,

3 https://github.com/w3c/wot-thing-description/issues/1508 (retrieved: July 1, 2022).

enabling other users to re-use converter in their projects, e.g., code generators or SDF-based mediators between different ecosystems. A simple web-based example for such a mediator can be seen on the demo page[4] by Ericsson mentioned in chapter 2, which allows the conversion of models defined in a variety of different description languages.

7.1.5 Other Requirements

Using Python's built-in modules, we are able to access WoT and SDF documents both from the file system and via HTTP and HTTPS when using the CLI. The latter enables us to use transport security for the retrieval, mitigating potential security and privacy concerns. The web version actually also provides a simple REST-API, which was an optional requirement defined in chapter 4. Finally, we are able to conduct input and output validation, with one slight drawback as there was no JSON Schema definition available for SDF mapping files at the time of writing. However, this is a feature that should be possible to add in one of the next minor versions of our converter implementation.

In the next section, we will turn to one of our non-functional requirements—efficiency and performance—in more detail.

7.2 Quantitative Criteria

For the quantitative evaluation of our converter, we used both metrics computed from static analysis and a performance benchmark test with models from the OneDM playground (see section 2.1.2).

7.2.1 Code Metrics

As a first metric for evaluating our converter, we calculated the cyclomatic complexity [McC76] using the Python package `radon`[5]. The results of this analysis can be seen in Listing B.1.

4 http://wishi.nomadiclab.com/sdf-converter/ (retrieved: July 1, 2022).

5 https://pypi.org/project/radon/ (retrieved: June 27, 2022).

Unsurprisingly, we achieve a low complexity value for most of the functions mapping single fields from one format to the other, leading to a total average complexity of about 2.292. This reflects the high similarity of the two specifications, making it possible to map many fields one-to-one from format to the other. However, it also sheds a positive light on our implementation, where have been able to factor out the conversion logic into re-usable functions, reducing the complexity of each individual function. However, this is also an aspect which we need to cautious about when interpreting the average complexity, as our decision to use one function which simply calls the generalized `map_field` function for many fields causes a potential distortion of the results.

Taking a closer look at the analysis, we can see that especially those functions which load external models (e.g., `_load_model_or_collection` with a complexity value of 5) or resolve references and extensions (like `resolve_sdf_ref` or `_resolve_tm_ref`) are the ones introducing a higher degree of complexity to the implementation. This is caused by the fact that functions like these need to differentiate between different types of inputs (e.g., file paths and URLs) and—in the case of references and extensions—also need to take nested references into account, requiring these kinds of functions to call themselves recursively if a referenced definition should contain a reference itself. Other functions introducing complexity are those which iterate over field values (e.g., `map_properties` in `tm_to_sdf_py` with a complexity value of 3) and those which need to differentiate between different cases, like `convert_wot_tm_to_sdf`, which has to be able to deal with both single TMs and TM collections. However, the complexity value of 9 of this particular function shows that there might be some room for refactoring in future versions of the converter, reducing the complexity of individual functions even more.

All in all, we can conclude from this analysis that we have been able to keep the overall complexity of our implementation fairly low, which is certainly also related to the similarity of both specifications. In the next section, we will see if this low complexity value also corresponds with an acceptable performance when actually running the converter from the command line.

7.2.2 Performance Comparison

In order to evaluate the performance of our converter, we set up a special Continuous Integration (CI) workflow in our GitHub repository, which automatically generates a

new performance report on every incremental change to the main development branch. As a benchmark, we use the models contained in the OneDM playground repository[6] and measure the time it takes to convert them from SDF to WoT Thing Models. As a reference for the performance, we use Jana Kiesewalter's SDF YANG converter, which we compile and install in the container environment prior to each workflow run in the virtualized CI environment. To perform the performance test, we call both converters from the command line and convert every SDF model to a WoT TM and a YANG model, respectively. Before and after both conversion steps, we create a timestamp, which we use to calculate the expired time during the conversion.

Due to the fact that we used the interpreted language Python for our implementation, we expected to observe a much worse performance of our converter in comparison to Kiesewalter's, which is written in the compiled language C++. However, as Table B.1 shows, our converter actually performs twice as fast for almost every SDF model, converting most models in about 0.18 to 0.23 seconds, while Kiesewalter's converter needs between 0.47 and 0.53 seconds for most of the conversions. This performance difference could arise from the fact that Kiesewalter's implementation performs more operations with the file system during a conversion process, e.g., importing the CDDL schema used for validation from a file instead of using an already deserialized ("hard-coded") version as we do.

While these are very good results for our implementation, especially considering our pessimistic expectations, we must interpret these results with caution: The set of models in the OneDM playground is not representative for the whole feature set of SDF, since all playground models only contain a single `sdfObject`. Furthermore, for example, they do not use `sdfRef` for external references. This omits a lot of potential complexity in SDF models, which prevents us from making a reliable judgement on how both converters would operate on scale. Future evaluations should therefore either try to diversify the set of input models and/or generate inputs procedurally. Furthermore, we cover only one direction of both the conversion processes with this evaluation process, which is also another aspect that should be improved. For now and considering the limited scope of this Bachelor's thesis, however, we can be satisfied with the results regarding our converter's performance.

6 https://github.com/one-data-model/playground (retrieved: June 29, 2022).

7.3 Possible Specification Improvements

Over the course of working on this thesis, we have been able to get an in-depth impression of both SDF and WoT TD, enabling us to make a judgement on possible gaps and incompatibilities of the two specifications. In this context, we have also been able to make contributions to both of their standardization processes. Therefore, we will evaluate how well aligned WoT TD and SDF are by now compared to the beginning of our writing process.

Initially, we saw of a number of hindrances for a comprehensive mapping between the two specifications: SDF did not have a mechanism for including additional or instance-specific fields in a model. Furthermore, the nesting approaches of both specifications were incompatible, as `sdfThing` definitions did not have a real WoT equivalent while affordances in TMs/TDs with submodels/subthings could not be included directly in an `sdfThing`. In the discussions with the SDF working group, we were able to introduce affordances as a feature for `sdfThings` to SDF. This greatly simplifies the mapping between hierarchies of TMs/TDs and SDF models.

Another aspect we have been dealing with is the addition of mapping files [I-D.-sdf-mapping] to SDF. Although this SDF extension is still in an early stage of development, we have been able to successfully use it for capturing information from TMs and TDs that is not yet supported by the core SDF vocabulary. Furthermore, mapping files enabled us to map SDF models with additional augmentation via mapping files to WoT TD, closing the gap between the two kinds of documents, as TDs have a number of mandatory instance-specific fields which cannot be expressed in an SDF model. In this regard, mapping files allow SDF to take a large step towards also being able to describe device instances. However, in the future SDF could also consider allowing the description of instances with a separate kind of document, which could be derived from SDF models augmented with mapping files.

In the case of WoT, we experimented with a new approach for modelling hierarchy in a single WoT document, which we call Thing Model/Description Collections. These allowed us not only to convert complex SDF models to WoT, but also made it possible to resolve submodels/subthings of TMs/TDs and convert all documents in the hierarchy to another format at the same time. This was especially necessary for converting nested SDF models to WoT TD, for which we used Thing Models as an intermediary. At the moment, we are modelling TM/TD collections as simple maps of key-value-pairs, where each

Listing 7.1: Example for a valid but unsatisfiable schema in [I-D.-jso-draft-7]

```
1 {
2   "type": "string",
3   "minimum": 5
4 }
```

value is a TM or TD. Submodels or subthings are referred to with same-document JSON pointers. While this simple approach works for now, in the future it would probably be better if such a collection would also be a valid JSON-LD document with its own top-level `@context` and also formally specified. As mentioned before, we already discussed these ideas with the WoT TD taskforce, and they will hopefully be considered for TD version 2.0.

A problem we noticed in both specification was the fact that their JSON Schema inspired type systems (or rather: the representation in the schema documents both specifications provide) are very loose and allow for defining valid data schemas which are unsatisfiable. An example for such a schema can be seen in Listing 7.1. Here, the schema specifies a "string" data type while also defining a minimum value of 5 at the same time, which in turn implies a numeric data type. Since there is no string that has a numeric value at the same time, this schema is unsatisfiable. As we eventually chose to use Python, which is dynamically typed, for our converter implementation, we were able to handle this flexible type system quite well. However, dealing with possible implementations in other languages, we noticed that it can become an issue for strongly typed languages such as Rust or C, which are a common choice for IoT devices in general and *constrained* devices in particular. Future versions of both specifications should therefore revisit how flexible their type systems should be and make them stricter, if possible.

8 Conclusion

In this thesis, we dealt with the conversion between Semantic Definition Format (SDF) [I-D.-asdf-sdf] and Web of Things (WoT) Thing Description (TD) [wot-td11]—two specifications which aim at improving the interoperability in the Internet of Things (IoT) in terms of data models and device interactions, respectively. On the basis of a detailed mapping between the data models of both specifications and a number of requirements for our implementation, we developed a converter written in Python which can be used as library and a Command Line Interface (CLI). Building upon our library, we also created a web application, which not only allows for easy access to the converter's logic via a Graphical User Interface (GUI), but also via a simple REST-API.

Corresponding with our mappings, our converter supports both WoT Thing Descriptions and Thing Models, the latter being a new addition that allows for describing not only *instances*, but also *classes* of devices, enabling the re-use of definitions via import and extension features. For our mappings, we discovered that Thing Models can act as intermediary for the conversion between SDF and Thing Descriptions, allowing us to only define two types of mappings (TM and SDF on the one hand and TM and TD on the other) in order to cover all three data formats. Besides SDF models, we used the newly defined concept of SDF mapping files [I-D.-sdf-mapping], allowing us to consider instance- and ecosystem-specific information during the conversion process. This aspect closes an important gap between the two specifications, making it possible to convert from SDF to WoT Thing Descriptions, provided that a mapping file is present that contains the necessary mandatory protocol bindings and security definitions. In WoT, we identified the need for being able to include multiple TMs or TDs in a single document in order to be able to work with nested data structures, as SDF allows for describing a hierarchy of Things in a single model, while the WoT TD specification does not, as the preferred way for creating hierarchy is by linking between documents. By introducing an experimental concept called TM/TD collections, we have been able to recreate a nested structure using JSON pointers [RFC6901] with same-document

references. These new concepts also enabled us to support roundtripping—i.e., the reproduction of original inputs from a conversion result—in most cases, with references to external documents (which we also resolve before the conversion) being the greatest exception in this regard.

While we have been able to close a number of gaps between the two specifications and were able to make a few contributions along this way, there is still room for improvement. In particular, there are parts of the WoT vocabulary which we currently need to map to a mapping file that could also be converted to actual SDF vocabulary. The most prominent example for this at the moment is probably the addition of SDF relations [I-D.-sdf-relations], which could be used for mapping generic WoT links to SDF. Other additions could involve security definitions or more concrete protocol-specific vocabulary. Furthermore, we noticed that the concept of SDF namespaces could be refined in order to make the relationship to JSON-LD [json-ld] and Resource Description Framework features used in WoT TD clearer. Therefore, the proposed mappings in this thesis are still only a starting point for a comprehensive alignment of the two specifications and should also be codified in a specification or standard as soon as both WoT TD is published as version 1.1 and SDF reaches RFC status.

With our converter implementation, we were able to fulfill the functional requirements we set ourselves based on the feature sets of existing converters, the potential deployment scenarios, and the mappings themselves. Despite the fact that with Python we chose an interpreted, dynamically typed language for our converter implementation, our evaluation suggested that we did not arrive at a significantly worse performance compared to a compiled language and achieved an even higher performance than a converter between SDF and YANG written in C++. However, these results should not be over-interpreted, as the approach we chose for our evaluation has a number of drawbacks, limiting its explanatory power. Still, in conjunction with the (limited) set of code-metrics we collected, the evaluation indicated that our implementation is a viable solution for converting between SDF and WoT TDs and TMs. For future versions of our converter, however, we still consider choosing a compiled, statically typed language like Rust for the implementation instead, potentially improving its performance and making it more robust, while also making it possible to use the converter in environments where a Python interpreter is not available.

Overall, we are satisfied with the results, especially considering that our converter is already available as an installable package, providing a high degree of usability and

flexibility via the CLI and the web application we built with it.

A Mapping Examples

A.1 SDF Data Qualities and WoT Data Schemas

Listing A.1: SDF model for illustrating the conversion of dataqualities.

```
1  {
2      "sdfObject": {
3          "Test": {
4              "label": "SdfTestObject",
5              "description": "An sdfObject used for testing",
6              "$comment": "This sdfObject is for testing only!",
7              "sdfProperty": {
8                  "foo": {
9                      "label": "This is a label.",
10                     "$comment": "This is a comment!",
11                     "type": "integer",
12                     "readable": true,
13                     "observable": False,
14                     "const": 5,
15                     "default": 5,
16                     "minimum": 0,
17                     "maximum": 9002,
18                     "exclusiveMinimum": 0,
19                     "exclusiveMaximum": 9000,
20                     "multipleOf": 2,
21                 },
22                 "bar": {
23                     "type": "number",
24                     "writable": true,
25                     "observable": true,
26                     "const": 5,
27                     "unit": "C",
28                     "default": 5,
29                     "minimum": 0.0,
30                     "maximum": 9002.0,
31                     "exclusiveMinimum": 0.0,
32                     "exclusiveMaximum": 9000.0,
33                     "multipleOf": 2.0,
34                 },
35                 "baz": {
36                     "type": "string",
```

```
                    "observable": False,
                    "minLength": 3,
                    "maxLength": 5,
                    "enum": ["hi", "hey"],
                    "pattern": "email",
                    "format": "uri-reference",
                    "contentFormat": "audio/mpeg",
                    "sdfType": "byte-string",
                },
                "foobar": {
                    "type": "array",
                    "observable": False,
                    "minItems": 2,
                    "maxItems": 5,
                    "uniqueItems": true,
                    "items": {"type": "string"},
                },
                "barfoo": {
                    "type": "object",
                    "observable": False,
                    "properties": {"foo": {"type": "string"}},
                    "required": ["foo"],
                    "nullable": true,
                },
            },
        }
    }
}
```

Listing A.2: Mapped TM created from the SDF example in Listing A.1.

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        {"sdf": "https://example.com/sdf"},
    ],
    "@type": "tm:ThingModel",
    "title": "SdfTestObject",
    "description": "An sdfObject used for testing",
    "sdf:$comment": "This sdfObject is for testing only!",
    "properties": {
        "foo": {
            "title": "This is a label.",
            "sdf:$comment": "This is a comment!",
            "writeOnly": False,
            "observable": False,
            "type": "integer",
            "const": 5,
            "default": 5,
            "minimum": 0,
            "maximum": 9002,
            "exclusiveMinimum": 0,
            "exclusiveMaximum": 9000,
            "multipleOf": 2,
```

```
24         },
25         "bar": {
26             "readOnly": False,
27             "observable": True,
28             "type": "number",
29             "const": 5,
30             "unit": "C",
31             "default": 5,
32             "minimum": 0.0,
33             "maximum": 9002.0,
34             "exclusiveMinimum": 0.0,
35             "exclusiveMaximum": 9000.0,
36             "multipleOf": 2.0,
37         },
38         "baz": {
39             "type": "string",
40             "observable": False,
41             "minLength": 3,
42             "maxLength": 5,
43             "enum": ["hi", "hey"],
44             "pattern": "email",
45             "format": "uri-reference",
46             "contentMediaType": "audio/mpeg",
47             "sdf:sdfType": "byte-string",
48         },
49         "foobar": {
50             "type": "array",
51             "observable": False,
52             "minItems": 2,
53             "maxItems": 5,
54             "items": {"type": "string"},
55             "sdf:uniqueItems": True,
56         },
57         "barfoo": {
58             "type": "object",
59             "observable": False,
60             "properties": {"foo": {"type": "string"}},
61             "required": ["foo"],
62             "sdf:nullable": True,
63         },
64     },
65     "sdf:objectKey": "Test",
66 }
```

A.2 References and Required Elements

Listing A.3: SDF example for the mapping sdfRef and sdfRequired.

```
1 {
2   "sdfObject": {
```

```
    "ExampleThing": {
      "label": "Common Qualities Example",
      "description": "Example for the mapping of common qualities from SDF to WoT",
      "$comment": "This is a comment!",
      "sdfRequired": [
        "#/sdfObject/ExampleThing/sdfProperty/ExampleProperty"
      ],
      "sdfProperty": {
        "ExampleProperty": {
          "sdfRef": "#/sdfProperty/YetAnotherExampleProperty"
        },
        "AnotherExampleProperty": {
          "sdfRef": "#/sdfObject/ExampleThing/sdfProperty/ExampleProperty"
        }
      }
    }
  },
  "sdfProperty": {
    "YetAnotherExampleProperty": {
      "type": "string"
    }
  }
}
```

Listing A.4: Mapped TM created from the SDF example in Listing A.3.

```
{
  "@context": {
    "http://www.w3.org/ns/td",
    {
      "sdf": "https://example.com/sdf"
    }
  },
  "label": "Common Qualities Example",
  "description": "Example for the mapping of common qualities from SDF to WoT",
  "sdf:$comment": "This is a comment!",
  "tm:required": [
    "#/properties/ExampleProperty"
  ],
  "properties": {
    "ExampleProperty": {
      "type": "string"
    },
    "AnotherExampleProperty": {
      "tm:ref": "#/properties/ExampleProperty"
    }
  }
}
```

B Evaluation Results

B.1 Code Metrics

Listing B.1: Cyclomatic Complexity [McC76] values for all elements of our library implementation.

```
sdf_wot_converter\__init__.py
    F 14:0 main - A (1)
sdf_wot_converter\cli\__init__.py
    F 54:0 _load_model_or_collection - A (5)
    F 344:0 _get_origin_url - A (5)
    F 87:0 save_or_print_model - A (4)
    F 294:0 _get_sdf_infoblock - A (4)
    F 388:0 _handle_from_tm - A (4)
    F 477:0 use_converter_cli - A (4)
    F 69:0 _load_sdf_mapping_files - A (3)
    F 355:0 _handle_from_sdf - A (3)
    F 440:0 _handle_from_td - A (3)
    F 47:0 _load_model - A (2)
    F 102:0 _add_input_argument - A (2)
    F 158:0 _add_sdf_arguments - A (2)
    F 183:0 _add_tm_arguments - A (2)
    F 237:0 _add_td_arguments - A (2)
    F 470:0 _load_optional_json_file - A (2)
    F 36:0 _load_model_from_path - A (1)
    F 41:0 _load_model_from_url - A (1)
    F 81:0 save_model - A (1)
    F 121:0 _add_output_argument - A (1)
    F 131:0 _add_mapping_file_input_argument - A (1)
    F 141:0 _add_mapping_file_output_argument - A (1)
    F 149:0 _add_origin_url - A (1)
    F 268:0 _add_sdf_infoblock_arguments - A (1)
    F 317:0 parse_arguments - A (1)
    F 489:0 main - A (1)
    C 19:0 CommandException - A (1)
sdf_wot_converter\converters\jsonschema.py
    F 5:0 map_common_json_schema_fields - A (1)
    F 32:0 map_pattern - A (1)
    F 38:0 map_format - A (1)
    F 44:0 map_required - A (1)
    F 50:0 map_maximum - A (1)
```

```
36      F 56:0 map_minimum - A (1)
37      F 62:0 map_max_items - A (1)
38      F 68:0 map_min_items - A (1)
39      F 74:0 map_max_length - A (1)
40      F 80:0 map_min_length - A (1)
41      F 86:0 map_multiple_of - A (1)
42      F 92:0 map_default - A (1)
43      F 98:0 map_const - A (1)
44      F 104:0 map_unit - A (1)
45      F 110:0 map_jsonschema_type - A (1)
46  sdf_wot_converter\converters\sdf_to_tm.py
47      F 49:0 resolve_sdf_ref - B (6)
48      F 158:0 map_license - A (4)
49      F 261:0 map_sdf_choice - A (4)
50      F 331:0 _map_exclusive_min_max - A (4)
51      F 791:0 map_sdf_ref - A (4)
52      F 821:0 map_sdf_objects - A (4)
53      F 924:0 map_sdf_things - A (4)
54      F 92:0 map_namespace - A (3)
55      F 124:0 create_link - A (3)
56      F 170:0 map_version - A (3)
57      F 223:0 copy_sdf_ref - A (3)
58      F 401:0 map_properties - A (3)
59      F 505:0 map_action_qualities - A (3)
60      F 614:0 map_sdf_data - A (3)
61      F 640:0 map_sdf_action - A (3)
62      F 672:0 map_sdf_property - A (3)
63      F 753:0 map_sdf_event - A (3)
64      F 780:0 map_sdf_required - A (3)
65      F 801:0 add_origin_link - A (3)
66      F 903:0 map_additional_fields - A (3)
67      F 1043:0 _fix_thing_model_json_ld_types - A (3)
68      F 1055:0 convert_sdf_to_wot_tm - A (3)
69      F 43:0 resolve_namespace - A (2)
70      F 106:0 map_default_namespace - A (2)
71      F 181:0 map_infoblock - A (2)
72      F 197:0 map_copyright - A (2)
73      F 204:0 map_title - A (2)
74      F 231:0 map_comment - A (2)
75      F 286:0 map_data_qualities - A (2)
76      F 424:0 map_items - A (2)
77      F 446:0 map_sdf_type - A (2)
78      F 461:0 map_unique_items - A (2)
79      F 476:0 map_nullable - A (2)
80      F 699:0 map_event_qualities - A (2)
81      F 913:0 add_link_to_parent - A (2)
82      F 1027:0 _apply_mapping_file - A (2)
83      F 1036:0 consolidate_sdf_model - A (2)
84      F 1050:0 _validate_thing_models - A (2)
85      F 145:0 add_link - A (1)
86      F 211:0 map_common_qualities - A (1)
87      F 249:0 map_description - A (1)
88      F 255:0 map_label - A (1)
```

```
89      F 355:0 _map_exclusive_maximum - A (1)
90      F 367:0 _map_exclusive_minimum - A (1)
91      F 379:0 map_writable - A (1)
92      F 390:0 map_readable - A (1)
93      F 441:0 map_observable - A (1)
94      F 491:0 map_content_format - A (1)
95      F 501:0 map_enum - A (1)
96      F 552:0 map_property_qualities - A (1)
97      F 579:0 map_sdf_data_qualities - A (1)
98      F 667:0 get_json_pointer - A (1)
99      F 741:0 collect_sdf_required - A (1)
100     F 749:0 collect_mapping - A (1)
101     F 813:0 create_basic_thing_model - A (1)
102     C 25:0 SdfRefLoopError - A (1)
103     C 31:0 InvalidSdfRefError - A (1)
104     C 37:0 SdfRefUrlRetrievalError - A (1)
105 sdf_wot_converter\converters\td_to_tm.py
106     F 13:0 _replace_type - A (2)
107     F 26:0 convert_td_collection_to_tm_collection - A (2)
108     F 37:0 convert_td_to_tm - A (2)
109 sdf_wot_converter\converters\tm_to_sdf.py
110     F 855:0 convert_wot_tm_to_sdf - B (9)
111     F 507:0 map_context_to_namespaces - B (7)
112     F 832:0 map_infoblock_fields - B (6)
113     F 937:0 _get_submodel_keys - B (6)
114     F 366:0 map_enum - A (5)
115     F 668:0 determine_thing_model_key - A (5)
116     F 946:0 detect_top_level_models - A (5)
117     F 242:0 map_data_schema_fields - A (4)
118     F 490:0 map_version - A (4)
119     F 32:0 map_properties - A (3)
120     F 90:0 map_dataschema_properties - A (3)
121     F 115:0 map_actions - A (3)
122     F 152:0 map_action_fields - A (3)
123     F 182:0 map_events - A (3)
124     F 430:0 map_schema_definitions - A (3)
125     F 467:0 map_links - A (3)
126     F 596:0 map_tm_required - A (3)
127     F 696:0 map_additional_fields - A (3)
128     F 782:0 map_thing_model - A (3)
129     F 824:0 get_license_link - A (3)
130     F 956:0 convert_wot_tm_collection_to_sdf - A (3)
131     F 66:0 map_items - A (2)
132     F 217:0 map_event_fields - A (2)
133     F 531:0 filter_at_type - A (2)
134     F 548:0 map_default_namespace - A (2)
135     F 565:0 convert_pointer - A (2)
136     F 580:0 map_tm_ref - A (2)
137     F 710:0 map_thing_model_to_sdf_thing - A (2)
138     F 300:0 _map_exclusive_maximum - A (1)
139     F 311:0 _map_exclusive_minimum - A (1)
140     F 322:0 map_nullable - A (1)
141     F 332:0 map_sdf_type - A (1)
```

```
142     F 342:0 map_unique_items - A (1)
143     F 354:0 map_content_format - A (1)
144     F 380:0 map_read_only - A (1)
145     F 391:0 map_write_only - A (1)
146     F 404:0 map_observable - A (1)
147     F 409:0 map_interaction_affordance_fields - A (1)
148     F 416:0 map_title - A (1)
149     F 422:0 map_description - A (1)
150     F 553:0 map_sdf_comment - A (1)
151     F 613:0 map_thing_model_to_sdf_object - A (1)
152     F 688:0 map_additional_field - A (1)
153 sdf_wot_converter\converters\tm_to_td.py
154     F 38:0 _assert_tm_required - B (10)
155     F 28:0 _replace_version - A (4)
156     F 78:0 _resolve_submodels - A (4)
157     F 104:0 convert_tm_to_td - A (4)
158     F 18:0 replace_type - A (2)
159     F 64:0 _replace_meta_data - A (2)
160     F 71:0 _replace_bindings - A (2)
161     F 90:0 convert_tm_collection_to_td_collection - A (2)
162 sdf_wot_converter\converters\utility.py
163     F 31:0 map_field - A (4)
164     F 4:0 initialize_object_field - A (2)
165     F 11:0 initialize_list_field - A (2)
166     F 18:0 ensure_value_is_list - A (2)
167     F 25:0 negate - A (1)
168     F 55:0 map_common_field - A (1)
169 sdf_wot_converter\converters\wot_common.py
170     F 152:0 _resolve_tm_ref - C (11)
171     F 62:0 resolve_extension - B (8)
172     F 30:0 retrieve_thing_model - A (4)
173     F 133:0 replace_placeholders - A (4)
174     F 216:0 resolve_sub_things - A (4)
175     F 235:0 _get_submodel_key_from_link - A (4)
176     F 108:0 _format_placeholder_value - A (3)
177     F 97:0 _stringify_boolean - A (2)
178     F 206:0 is_thing_collection - A (2)
179     F 18:0 _retrieve_thing_model_from_url - A (1)
180     F 24:0 _retrieve_thing_model_from_file_path - A (1)
181     F 44:0 _perform_extension - A (1)
182     F 104:0 _has_placeholders - A (1)
183     F 118:0 _format_placeholder_key - A (1)
184     F 124:0 _replace_placeholder - A (1)
185     C 12:0 PlaceholderException - A (1)
186 sdf_wot_converter\converters\__init__.py
187     F 11:0 convert_wot_td_to_wot_tm - A (1)
188     F 15:0 convert_wot_td_to_sdf - A (1)
189     F 26:0 convert_sdf_to_wot_tm - A (1)
190     F 42:0 convert_sdf_to_wot_td - A (1)
191     F 72:0 convert_wot_tm_to_sdf - A (1)
192     F 88:0 convert_wot_tm_to_wot_td - A (1)
193 sdf_wot_converter\validation\__init__.py
194     F 18:0 validate_sdf_model - A (2)
```

```
195     F 25:0 validate_thing_model - A (1)
196     F 29:0 validate_thing_description - A (1)
197
198 185 blocks (classes, functions, methods) analyzed.
199 Average complexity: A (2.291891891891892)
```

B.2 Performance Comparison

Table B.1: Performance comparison between our converter and Jana Kiesewalter's SDF-YANG-Converter, based on the `sdfObjects` in the OneDM playground. The values in the second and third row are the number of seconds the conversion from SDF to the respective format took. Cases where the conversion process was unsuccessful (e.g., due to a validation error) are labelled as "failed".

	Conversion Time (s)	
File Name	**SDF → WoT (s)**	**SDF → YANG**
sdfobject-accelerometer.sdf.json	.196189118	.539649950
sdfobject-acidity.sdf.json	.193596280	.489524831
sdfobject-activity.sdf.json	.195034402	.489960837
sdfobject-actuation.sdf.json	.194102388	.538188129
sdfobject-addressable_text_display.sdf.json	.192091459	.484375857
sdfobject-airflow.sdf.json	.192804769	.483684946
sdfobject-airquality.sdf.json	.192656067	.486825791
sdfobject-alarm.sdf.json	.194620495	.488625517
sdfobject-altimeter.sdf.json	.190831141	.532621649
sdfobject-altitude.sdf.json	.193734382	.500958194
sdfobject-analog_input.sdf.json	.225133532	.490543643
sdfobject-analog_output.sdf.json	.194888698	.487133094
sdfobject-audio.sdf.json	.190214731	.484532657
sdfobject-audio_clip.sdf.json	.192186660	.484405455
sdfobject-autofocus.sdf.json	.221767884	.482190923
sdfobject-automaticdocumentfeeder.sdf.json	.190075829	.487002692
sdfobject-barometer.sdf.json	.197395934	.493460985

Continued on next page

Table B.1 – continued from previous page

File Name	*Conversion Time (s)*	
	SDF → WoT	**SDF → YANG**
sdfobject-batterymaterial.sdf.json	.195668110	.488771017
sdfobject-bitmap.sdf.json	.196217718	.492060065
sdfobject-blood_pressure.sdf.json	.195382205	.486291982
sdfobject-bmi.sdf.json	.191513650	.485512971
sdfobject-body_fat.sdf.json	.194718994	.485599870
sdfobject-body_ffm.sdf.json	.217881627	.538050322
sdfobject-body_location_temperature.sdf.json	.192006856	.480871702
sdfobject-body_slm.sdf.json	.194337189	.485710571
sdfobject-body_water.sdf.json	.190358432	.485482568
sdfobject-brewing.sdf.json	.191251445	.484645256
sdfobject-button.sdf.json	.212789754	.481383909
sdfobject-buzzer.sdf.json	.192153258	.484502853
sdfobject-cadence.sdf.json	.187943097	.483549540
sdfobject-calorificvalue.sdf.json	.187110385	.481470111
sdfobject-cgm_calibrate.sdf.json	.189734423	.485416167
sdfobject-cgm_samplinginterval.sdf.json	.191813253	.484074537
sdfobject-cgm_sensor.sdf.json	.187862487	.486280958
sdfobject-cgm_status.sdf.json	.189820816	.484720835
sdfobject-cgm_threshold.sdf.json	.191898245	.485002340
sdfobject-circuitbreaker.sdf.json	.190645327	.482687206
sdfobject-clock.sdf.json	.190002218	.483946824
sdfobject-colour.sdf.json	.194151778	.539046912
sdfobject-colour_autowhitebalance.sdf.json	.188725400	.528593963
sdfobject-colour_chroma.sdf.json	.210481712	.532276715
sdfobject-colour_colourtemperature.sdf.json	.189526012	.529785280
sdfobject-colour_csc.sdf.json	.211731930	.481659491
sdfobject-colour_hs.sdf.json	.213426654	.532317124
sdfobject-colour_rgb.sdf.json	.192389182	.483442689
sdfobject-colour_saturation.sdf.json	.190105048	.481838266
sdfobject-concentration.sdf.json	.197510356	.491790110

Continued on next page

Table B.1 – continued from previous page

File Name	*Conversion Time (s)*	
	SDF → WoT	**SDF → YANG**
sdfobject-conductivity.sdf.json	.193874802	.492116114
sdfobject-consumable.sdf.json	.212445972	.482465975
sdfobject-conversionfactor.sdf.json	.211687860	.481375059
sdfobject-current.sdf.json	.193641300	.488519162
sdfobject-cyclingpower.sdf.json	.189696642	.480265643
sdfobject-delaydefrost.sdf.json	.190135849	.532018691
sdfobject-deodorization.sdf.json	.211771862	.529829860
sdfobject-depth.sdf.json	.216537731	.518842401
sdfobject-digital_input.sdf.json	.194166107	.485138831
sdfobject-digital_output.sdf.json	.190604263	.481651980
sdfobject-dimmer.sdf.json	.189318544	.532719220
sdfobject-direction.sdf.json	.217092246	.485944443
sdfobject-distance.sdf.json	.191038169	.484809526
sdfobject-door.sdf.json	.188672634	.481981985
sdfobject-ecomode.sdf.json	.188063825	.484101116
sdfobject-energy.sdf.json	.193531605	.484267318
sdfobject-energy_battery.sdf.json	.189054840	.482902599
sdfobject-energy_consumption.sdf.json	.190658063	.479454849
sdfobject-energy_drlc.sdf.json	.190439460	.485001229
sdfobject-energy_generation.sdf.json	.188735035	.478815691
sdfobject-energy_overload.sdf.json	.188312976	.479546887
sdfobject-foaming.sdf.json	.189965395	.482755122
sdfobject-frequency.sdf.json	.218011404	.487105469
sdfobject-gas_consumption.sdf.json	.191380610	.482072015
sdfobject-generic_sensor.sdf.json	.193957438	.541075465
sdfobject-genericdefaulttransitiontime.sdf.json	.191341409	.484746744
sdfobject-genericlevel.sdf.json	.197583478	.497340883
sdfobject-genericonoff.sdf.json	.216978492	.541755972
sdfobject-glucose.sdf.json	.214773667	.534501992
sdfobject-glucose_carb.sdf.json	.190637002	.481461382

Continued on next page

Table B.1 – continued from previous page

File Name	*Conversion Time (s)* SDF → WoT	SDF → YANG
sdfobject-glucose_exercise.sdf.json	.191555839	.544375182
sdfobject-glucose_hba1c.sdf.json	.191506616	.480449568
sdfobject-glucose_health.sdf.json	.191293773	.530445345
sdfobject-glucose_meal.sdf.json	.189399344	.477980830
sdfobject-glucose_medication.sdf.json	.190137556	.481663687
sdfobject-glucose_samplelocation.sdf.json	.189766450	.533866598
sdfobject-glucose_tester.sdf.json	.188806435	.482552600
sdfobject-grinder.sdf.json	.223518775	.497720036
sdfobject-gyrometer.sdf.json	.193818013	.491398338
sdfobject-heartrate.sdf.json	.191533478	.488552894
sdfobject-heatingzone.sdf.json	.189857251	.481109078
sdfobject-height.sdf.json	.191335474	.483931022
sdfobject-humidity.sdf.json	.188552130	.482572237
sdfobject-hvac_capacity.sdf.json	.190405994	.478615071
sdfobject-icemaker.sdf.json	.188266664	.485595871
sdfobject-illuminance.sdf.json	.192352122	.485922375
sdfobject-impactsensor.sdf.json	.188995374	.480678401
sdfobject-inverter.sdf.json	.190862900	.484573556
sdfobject-ipso-humidity.sdf.json	.194807056	.485618271
sdfobject-ipso-temperature.sdf.json	.203247975	.555082253
sdfobject-keycardswitch.sdf.json	.189690784	.477698259
sdfobject-keypadchar.sdf.json	.190619697	.480887603
sdfobject-level.sdf.json	.204674196	Failed
sdfobject-light_brightness.sdf.json	.211102487	.482437540
sdfobject-light_control.sdf.json	.216417133	.540647128
sdfobject-light_dimming.sdf.json	.213111683	.533059312
sdfobject-light_ramptime.sdf.json	.217522250	.484196859
sdfobject-liquid_level.sdf.json	.191829555	.482248929
sdfobject-load.sdf.json	.219818786	.545301900
sdfobject-load_control.sdf.json	.194957103	.536385963

Continued on next page

Table B.1 – continued from previous page

	Conversion Time (s)	
File Name	**SDF → WoT**	**SDF → YANG**
sdfobject-location.sdf.json	.219459381	.548410648
sdfobject-lock_code.sdf.json	.190694438	.480450501
sdfobject-lock_status.sdf.json	.192033458	.493564103
sdfobject-loudness.sdf.json	.195602913	.539542211
sdfobject-magnetometer.sdf.json	.216736539	.488312917
sdfobject-mediasource.sdf.json	.191523679	.485907452
sdfobject-mode.sdf.json	.192139988	.479938865
sdfobject-movement_linear.sdf.json	.192350791	.481763992
sdfobject-multi-state_selector.sdf.json	.190988871	.482688905
sdfobject-multiple_axis_joystick.sdf.json	.192485894	.484495531
sdfobject-muscleoxygensaturation.sdf.json	.193902214	.482008095
sdfobject-nightmode.sdf.json	.188488235	.480082267
sdfobject-on_off_switch.sdf.json	.190172460	.485583847
sdfobject-onoff.sdf.json	.195652439	Failed
sdfobject-opaquedata.sdf.json	.210832459	.535945278
sdfobject-openlevel.sdf.json	.211801374	.484333329
sdfobject-operational_state.sdf.json	.191181248	.484163532
sdfobject-orfid_station.sdf.json	.188607917	.484979843
sdfobject-orfid_tag.sdf.json	.187102595	.481892198
sdfobject-percentage.sdf.json	.193629192	.489883517
sdfobject-positioner.sdf.json	.192025768	.487869186
sdfobject-power.sdf.json	.196293732	.486718269
sdfobject-power_control.sdf.json	.193713093	.482832012
sdfobject-power_factor.sdf.json	.195703523	.536069807
sdfobject-power_measurement.sdf.json	.201448608	.546054656
sdfobject-presence.sdf.json	.194656408	.537102923
sdfobject-pressure.sdf.json	.193635792	.536002806
sdfobject-printer_3d.sdf.json	.212633675	.532195855
sdfobject-ptz.sdf.json	.192110257	.534308066
sdfobject-pulsatilecharacteristic.sdf.json	.210637522	.530560312

Continued on next page

Table B.1 – continued from previous page

	Conversion Time (s)	
File Name	**SDF → WoT**	**SDF → YANG**
sdfobject-pulsatileoccurrence.sdf.json	.211646337	.531322824
sdfobject-pulserate.sdf.json	.211366833	.528618584
sdfobject-push_button.sdf.json	.211730038	.483904943
sdfobject-pvconnectionterminal.sdf.json	.191660150	.485740469
sdfobject-rate.sdf.json	.218687537	.487775498
sdfobject-selectablelevels.sdf.json	.189017912	.482763726
sdfobject-sensor.sdf.json	.189406818	.486743584
sdfobject-sensor_acceleration.sdf.json	.189674122	.482627425
sdfobject-sensor_activity_count.sdf.json	.192125056	.484012628
sdfobject-sensor_atmosphericpressure.sdf.json	.192136046	.479880159
sdfobject-sensor_carbondioxide.sdf.json	.190765527	.530415582
sdfobject-sensor_carbonmonoxide.sdf.json	.192641653	.481178478
sdfobject-sensor_contact.sdf.json	.189928115	.528643357
sdfobject-sensor_geolocation.sdf.json	.189286106	.532879417
sdfobject-sensor_glassbreak.sdf.json	.192674454	.482268394
sdfobject-sensor_heart_zone.sdf.json	.188671597	.480833373
sdfobject-sensor_illuminance.sdf.json	.232028217	.482200993
sdfobject-sensor_magneticfielddirection.sdf.json	.210760012	.484818630
sdfobject-sensor_motion.sdf.json	.192385550	.524372796
sdfobject-sensor_presence.sdf.json	.190345321	.480699387
sdfobject-sensor_props.sdf.json	.190643192	.483072173
sdfobject-sensor_radiation_uv.sdf.json	.190730692	.479148615
sdfobject-sensor_sleep.sdf.json	.188060354	.492349809
sdfobject-sensor_smoke.sdf.json	.190193784	.482119059
sdfobject-sensor_threeaxis.sdf.json	.190651191	.481551850
sdfobject-sensor_touch.sdf.json	.187995852	.482433264
sdfobject-sensor_water.sdf.json	.191960211	.483423978
sdfobject-set_point.sdf.json	.192884124	.488388551
sdfobject-signalstrength.sdf.json	.189972881	.481322347
sdfobject-sleep.sdf.json	.193555833	.501381340

Continued on next page

Table B.1 – continued from previous page

	Conversion Time (s)	
File Name	**SDF → WoT**	**SDF → YANG**
sdfobject-speech_tts.sdf.json	.190247285	.483205974
sdfobject-speed.sdf.json	.188690362	.483540884
sdfobject-spo2.sdf.json	.192798127	.487000641
sdfobject-stopwatch.sdf.json	.190547794	.485436618
sdfobject-switch_binary.sdf.json	.188742868	.483134584
sdfobject-switch_fault.sdf.json	.211293598	.532730812
sdfobject-switch_restricted.sdf.json	.189181774	.482355872
sdfobject-temperature.sdf.json	.190496793	.488568663
sdfobject-time.sdf.json	.190311991	.483284286
sdfobject-time_period.sdf.json	.190795997	.485288016
sdfobject-time_stamp.sdf.json	.189623380	.482725978
sdfobject-timer.sdf.json	.192732126	.486973040
sdfobject-torque.sdf.json	.190507893	.482597276
sdfobject-up_down_control.sdf.json	.195922225	.483935250
sdfobject-userid.sdf.json	.216420300	.512708328
sdfobject-vehicle_connector.sdf.json	.192646004	.479828681
sdfobject-voltage.sdf.json	.193980426	.485966183
sdfobject-waterinfo.sdf.json	.192836307	.532770961
sdfobject-weight.sdf.json	.212310932	.483480741

Bibliography

Technical Specifications

[CURIE] S. McCarron and M. Birbeck, "CURIE Syntax 1.0", W3C, W3C Working Group Note, Dec. 2010. [Online]. Available: https://www.w3.org/TR/2010/NOTE-curie-20101216/.

[I-D.-asdf-sdf] M. Koster and C. Bormann, "Semantic Definition Format (SDF) for Data and Interactions of Things", IETF, Internet-Draft draft-ietf-asdf-sdf-12, Jun. 2022, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-asdf-sdf-12.

[I-D.-jso-draft-2019-09] A. Wright, H. Andrews, and B. Hutton, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", IETF, Internet-Draft draft-handrews-json-schema-validation-02, Sep. 2019, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-validation-02.

[I-D.-jso-draft-7] A. Wright, H. Andrews, and G. Luff, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", IETF, Internet-Draft draft-handrews-json-schema-validation-01, Mar. 2018, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-validation-01.

[I-D.-sdf-mapping] C. Bormann and J. Romann, "Semantic Definition Format (SDF): Mapping files", IETF, Internet-Draft draft-bormann-asdf-sdf-mapping-00, Nov. 2021, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-bormann-asdf-sdf-mapping-00.

[I-D.-sdf-relations] P. Laari, "Extended relation information for Semantic Definition Format (SDF)", IETF, Internet-Draft draft-laari-asdf-relations-00, Jun. 2022, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-laari-asdf-relations-00.

[I-D.-yang-sdf] J. Kiesewalter and C. Bormann, "Mapping between YANG and SDF", IETF, Internet-Draft draft-kiesewalter-asdf-yang-sdf-01, Nov. 2021, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-kiesewalter-asdf-yang-sdf-01.

[json-ld] D. Longley, P.-A. Champin, and G. Kellogg, "Json-ld 1.1", W3C, W3C Recommendation, Jul. 2020. [Online]. Available: https://www.w3.org/TR/json-ld11/.

[rdf] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax", W3C, W3C Recommendation, Feb. 2014. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/.

[RFC2288] M. Nottingham, "Web Linking", IETF, RFC 8288, Oct. 2017. DOI: 10.17487/RFC8288.

[RFC4946] J. M. Snell, "Atom License Extension", IETF, RFC 4946, Jul. 2007. DOI: 10.17487/RFC4946.

[RFC6570] R. T. Fielding, M. Nottingham, D. Orchard, J. Gregorio, and M. Hadley, "URI Template", IETF, RFC 6570, Mar. 2012. DOI: 10.17487/RFC6570.

[RFC6749] D. Hardt, "The OAuth 2.0 Authorization Framework", IETF, RFC 6749, Oct. 2012. DOI: 10.17487/RFC6749.

[RFC6901] P. C. Bryan, K. Zyp, and M. Nottingham, "JavaScript Object Notation (JSON) Pointer", IETF, RFC 6901, Apr. 2013. DOI: 10.17487/RFC6901.

[RFC7228] C. Bormann, M. Ersue, and A. Keränen, "Terminology for Constrained-Node Networks", IETF, RFC 7228, May 2014. DOI: 10.17487/RFC7228.

[RFC7396] P. E. Hoffman and J. M. Snell, "JSON Merge Patch", IETF, RFC 7396, Oct. 2014. DOI: 10.17487/RFC7396.

[RFC7617] J. Reschke, "The 'Basic' HTTP Authentication Scheme", IETF, RFC 7617, Sep. 2015. DOI: 10.17487/RFC7617.

[RFC7641] K. Hartke, "Observing Resources in the Constrained Application Protocol (CoAP)", IETF, RFC 7641, Sep. 2015. DOI: 10.17487/RFC7641.

[RFC7950] M. Bjorklund, "The YANG 1.1 Data Modeling Language", IETF, RFC 7950, Aug. 2016. DOI: 10.17487/RFC7950.

[RFC8259] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format", IETF, RFC 8259, Dec. 2017. DOI: 10.17487/RFC8259.

[RFC8610] H. Birkholz, C. Vigano, and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", IETF, RFC 8610, Jun. 2019. DOI: 10.17487/RFC8610.

[wot-architecture] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of Things (WoT) Architecture", W3C, W3C Recommendation, Apr. 2020. [Online]. Available: https://www.w3.org/TR/wot-architecture/.

[wot-td] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "Web of Things (WoT) Thing Description", W3C, W3C Recommendation, Apr. 2020. [Online]. Available: https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/.

[wot-td11] S. Käbisch, T. Kamiya, M. McCool, and V. Charpenay, “Web of Things (WoT) Thing Description”, W3C, W3C Working Draft, Jun. 2022. [Online]. Available: https://www.w3.org/TR/2021/WD-wot-thing-description11-20210607/.

Additional References

[Kie21] J. Kiesewalter, “Jana Kiesewalter: Design and Implementation of an SDF/YANG Converter in the Context of Standardization”, unpublished, M.S. thesis, University of Bremen, 2021.

[Kra21] R. Kravtsov, “Thing Model for the Web of Things”, unpublished, M.S. thesis, University of Passau, 2021.

[McC76] T. McCabe, “A complexity measure”, *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. DOI: 10.1109/TSE.1976.233837.

<u>**E R K L Ä R U N G**</u>

Ich versichere, den Bachelor-Report oder den von mir zu verantwortenden Teil einer Gruppenarbeit*) ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

*) Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und den Anforderungen entsprechen.

Bremen, den ________________

____
(Unterschrift)