

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: 11 May 2022

J. Kieseewalter  
Universität Bremen  
C. Bormann, Ed.  
Universität Bremen TZI  
7 November 2021

Mapping between YANG and SDF  
draft-kieseewalter-asdf-yang-sdf-01

## Abstract

YANG and SDF are two languages for modelling the interaction with and the data interchanged with devices in the network. As their areas of application (network management, IoT, resp.) overlap, it is useful to be able to translate between the two.

The present specification provides information about how models in one of the two languages can be translated into the other. This specification is not intended to be normative, but to help with creating translators.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 11 May 2022.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the [Trust Legal Provisions](#) and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

<a href="#">1.</a>	<a href="#">Introduction</a>	<a href="#">3</a>
<a href="#">2.</a>	<a href="#">Pairing SDF and YANG features</a>	<a href="#">3</a>
<a href="#">3.</a>	<a href="#">Mapping from YANG to SDF</a>	<a href="#">11</a>
<a href="#">3.1.</a>	<a href="#">Module</a>	<a href="#">11</a>
<a href="#">3.2.</a>	<a href="#">Submodule</a>	<a href="#">13</a>
<a href="#">3.3.</a>	<a href="#">Container Statement</a>	<a href="#">13</a>
<a href="#">3.4.</a>	<a href="#">Leaf Statement</a>	<a href="#">15</a>
<a href="#">3.5.</a>	<a href="#">Leaf-List Statement</a>	<a href="#">17</a>
<a href="#">3.6.</a>	<a href="#">List Statement</a>	<a href="#">18</a>
<a href="#">3.7.</a>	<a href="#">Grouping Statement</a>	<a href="#">19</a>
<a href="#">3.8.</a>	<a href="#">Uses Statement</a>	<a href="#">20</a>
<a href="#">3.9.</a>	<a href="#">Choice Statement</a>	<a href="#">21</a>
<a href="#">3.10.</a>	<a href="#">RPC Statement</a>	<a href="#">24</a>
<a href="#">3.11.</a>	<a href="#">Action Statement</a>	<a href="#">24</a>
<a href="#">3.12.</a>	<a href="#">Notification Statement</a>	<a href="#">26</a>
<a href="#">3.13.</a>	<a href="#">Augment Statement</a>	<a href="#">27</a>
<a href="#">3.14.</a>	<a href="#">Anydata and Anyxml Statements</a>	<a href="#">28</a>
<a href="#">3.15.</a>	<a href="#">Type Statement</a>	<a href="#">28</a>
<a href="#">3.16.</a>	<a href="#">String Built-In Type</a>	<a href="#">29</a>
<a href="#">3.17.</a>	<a href="#">Decimal64 Built-In Type</a>	<a href="#">31</a>
<a href="#">3.18.</a>	<a href="#">Integer Built-In Types</a>	<a href="#">33</a>
<a href="#">3.19.</a>	<a href="#">Boolean Built-In Type</a>	<a href="#">34</a>
<a href="#">3.20.</a>	<a href="#">Binary Built-In Type</a>	<a href="#">34</a>
<a href="#">3.21.</a>	<a href="#">Enumeration Built-In Type</a>	<a href="#">35</a>
<a href="#">3.22.</a>	<a href="#">Bits Built-In Type</a>	<a href="#">35</a>
<a href="#">3.23.</a>	<a href="#">Union Built-In Type</a>	<a href="#">36</a>
<a href="#">3.24.</a>	<a href="#">Leafref and Identityref Built-In Types</a>	<a href="#">37</a>
<a href="#">3.25.</a>	<a href="#">Empty Built-In Type</a>	<a href="#">37</a>
<a href="#">3.26.</a>	<a href="#">Instance-Identifier Built-In Type</a>	<a href="#">37</a>
<a href="#">3.27.</a>	<a href="#">Typedef Statement</a>	<a href="#">38</a>
<a href="#">3.28.</a>	<a href="#">Identity Statement</a>	<a href="#">38</a>
<a href="#">3.29.</a>	<a href="#">Config Statement</a>	<a href="#">38</a>
<a href="#">3.30.</a>	<a href="#">Status Statement</a>	<a href="#">39</a>
<a href="#">3.31.</a>	<a href="#">Reference Statement</a>	<a href="#">39</a>
<a href="#">3.32.</a>	<a href="#">When and Must Statements</a>	<a href="#">39</a>
<a href="#">3.33.</a>	<a href="#">Extension Statement</a>	<a href="#">40</a>

<a href="#">4.</a>	Mapping from SDF to YANG . . . . .	<a href="#">40</a>
<a href="#">4.1.</a>	Information Block . . . . .	<a href="#">40</a>
<a href="#">4.2.</a>	Namespace Section . . . . .	<a href="#">41</a>
<a href="#">4.3.</a>	SdfThing Quality . . . . .	<a href="#">41</a>
<a href="#">4.4.</a>	SdfObject Quality . . . . .	<a href="#">41</a>

<a href="#">4.5.</a>	Common Qualities . . . . .	<a href="#">42</a>
<a href="#">4.6.</a>	Data Qualities . . . . .	<a href="#">50</a>
<a href="#">4.7.</a>	SdfData Quality . . . . .	<a href="#">57</a>
<a href="#">4.8.</a>	SdfProperty Quality . . . . .	<a href="#">59</a>
<a href="#">4.9.</a>	SdfAction Quality . . . . .	<a href="#">62</a>
<a href="#">4.10.</a>	SdfEvent Quality . . . . .	<a href="#">63</a>
<a href="#">5.</a>	Challenges . . . . .	<a href="#">64</a>
<a href="#">5.1.</a>	Differences in Expressiveness of SDF and YANG . . . . .	<a href="#">64</a>
<a href="#">5.2.</a>	Round Trips . . . . .	<a href="#">66</a>
<a href="#">5.3.</a>	Type References . . . . .	<a href="#">67</a>
<a href="#">6.</a>	Implementation Considerations . . . . .	<a href="#">70</a>
<a href="#">7.</a>	IANA Considerations . . . . .	<a href="#">70</a>
<a href="#">8.</a>	Security considerations . . . . .	<a href="#">70</a>
<a href="#">9.</a>	References . . . . .	<a href="#">70</a>
<a href="#">9.1.</a>	Normative References . . . . .	<a href="#">70</a>
<a href="#">9.2.</a>	Informative References . . . . .	<a href="#">70</a>
	Acknowledgements . . . . .	<a href="#">71</a>
	Authors' Addresses . . . . .	<a href="#">71</a>

## [1.](#) Introduction

YANG [[RFC7950](#)] and SDF [[I-D.ietf-asdf-sdf](#)] are two languages for modelling the interaction with and the data interchanged with devices in the network. As their areas of application (network management, IoT, resp.) overlap, it is useful to be able to translate between the two.

The present specification provides information about how models in one of the two languages can be translated into the other. This specification is not intended to be normative, but to help with creating translators.

## [2.](#) Pairing SDF and YANG features

Table 1 gives an overview over how language features of YANG can be mapped to SDF features. In many cases, several translations are

possible, and the right choice depends on the context. The mappings in this draft often accommodate the use of the YANG parser Libyang [[LIBYANG](#)].

For YANG statements that are not mentioned in the table no conversion to SDF was found that preserves the statement's semantics.

For possible conversions of YANG's built-in types please refer to [Section 3](#). Please note that a 'type object' is not the same as an `sdfObject` but refers to SDF's built-in type 'object', also called compound-type. This built-in type makes use of the 'properties' quality which is not to be confused with the `sdfProperty` class. The

data types number/decimal64, integer, boolean, string are also referred to as simple (data) types. In turn, the types array and object are sometimes referred to as complex (data) types. Concerning YANG, the expression 'schema tree' refers to the model's tree whereas 'data tree' describes the tree of an instance of the model.

YANG statement	remark on YANG statement	converted to SDF
module		SDF model (i.e., info block, namespace section & definitions)
submodule	included in supermodule	integrated into SDF model of supermodule
	on its own	SDF model
container	top-level	<code>sdfObject</code>
	one level below top- level	<code>sdfProperty</code> of type object (compound-type)
	on any other level	property (type object) of the 'parent' definition of type object (compound-type)

leaf	on top-level and one level below	sdfProperty (type integer/number/boolean/string)
	on any other level	property (type integer/number/boolean/string) of the 'parent' definition of type object (compound-type)
leaflist	on top-level and one level below	sdfProperty of type array
	on any other level	property (type array) of the 'parent' definition of type object (compound-type)
list	on top-level	sdfProperty of type array with

	and one level below	items of type object (compound-type)
	on any other level	property (type array with items of type object (compound-type)) of the 'parent' definition of type object* (compound-type)
choice		sdfChoice
case	belonging to choice	element of the sdfChoice
grouping		sdfData of compound-type (type object) at the top level which can then be referenced
uses	referencing a grouping	sdfRef to the SDF definition corresponding to the referenced grouping

rpc		sdfAction at the top-level of the SDF model	
action		sdfAction of the sdfObject corresponding to a container the action is a descendant node to	
notification		sdfEvent	
anydata		not converted	
anyxml		not converted	
augment		augment's target is converted with the augmentation already applied, mentioned in the description	
type	referring to a built-in type	type with other data qualities (e.g., default) if necessary	
type	referring to a typedef	sdfRef to the corresponding sdfData element	

base		sdfRef to the sdfData definition corresponding to the base	
bit		'parent' definition is of compound-type and gets one entry in the properties quality of type boolean for each bit	
enum		each enum statement's argument is added as an element to the SDF enum quality's string array	

fraction-digits		multipleOf quality	
+-----+	+-----+	+-----+	+-----+
length	single length range	minLength/maxLength qualities	
+-----+	+-----+	+-----+	+-----+
	single value	minLength and maxLength qualities set to the same value	
+-----+	+-----+	+-----+	+-----+
	contains alternatives	sdfChoice with alternatives for minLength/maxLength qualities	
+-----+	+-----+	+-----+	+-----+
path		sdfRef to the corresponding SDF definition	
+-----+	+-----+	+-----+	+-----+
pattern	single pattern	pattern quality	
+-----+	+-----+	+-----+	+-----+
	multiple patterns	pattern quality, the regular expressions are combined using positive lookahead	
+-----+	+-----+	+-----+	+-----+
	invert-match	pattern quality, the regular expression is modified using negative lookahead	
+-----+	+-----+	+-----+	+-----+
range	single range	minimum/maximum qualities	
+-----+	+-----+	+-----+	+-----+
	single value (constant)	const quality	
+-----+	+-----+	+-----+	+-----+
	contains	sdfChoice with either minimum/	

	alternatives	maximum or const quality as alternatives	
+-----+	+-----+	+-----+	+-----+
typedef		sdfData definition, sdfRef where it is used	
+-----+	+-----+	+-----+	+-----+
identity		sdfData definition, sdfRef where it is used	

config	of a container that became an sdfObject	set writable for all elements in the sdfObject that can be marked as writable (i.e., that use the data qualities)
	of any other YANG element	set writable
import		the module that the import references is converted (elements can now be referenced by sdfRef) and its prefix and namespace are added to the namespace section
revisions		first revision date becomes version in information block
namespace		added to namespace section
prefix		added to namespace section

Table 1: Mapping YANG to SDF

Table 2 provides the inverse mapping.

SDF quality	remark on SDF quality	converted to YANG
sdfThing		container node
sdfObject		container node
sdfProperty	type integer/number/boolean/ string	leaf node
	type array with items	leaf-list node



	integer/number/boolean/ string	
	type array with items of type object (compound-type)	list node
	type object (compound- type)	container node
sdfAction	at the top-level, *not* part of an sdfObject	rpc node
	inside of an sdfObject	action node as child node to the container corresponding to the sdfObject
sdfEvent		notification node with child nodes that were translated like sdfProperty
sdfData	type integer/number/boolean/ string	typedef
	type array with items of type integer/number/boolean/ string	grouping node with leaf-list child node
	type array with items of type object (compound-type)	grouping node with list child node
	type object (compound- type)	grouping node
sdfRef	referenced definition was converted to typedef	type is set to the typedef corresponding to the sdfData definition
	referenced definition was converted to leaf or leaf-list node	leafref

	referenced definition was converted to grouping node	"uses" node that references corresponding grouping (and refine if necessary)
sdfRequired	referenced definition was converted to a leaf or choice node	set the mandatory statement of the corresponding leaf/choice node to true
		find the first descendant node that is either a leaf/choice node and set their mandatory statement to true or that is a leaf-list/list node and set their min-elements statement to 1 (if not already $\geq 0$ )
sdfChoice		choice node with one case node for each alternative of the sdfChoice, each alternative is converted like sdfProperty
type		
const	corresponding YANG element has empty range	range statement with a single value
	range not empty	add single value alternative to range statement (must be disjunct)
default	type is one of integer/number/boolean/string or array with items of these types	default statement of leaf/leaf-list nodes

minimum/	corresponding YANG	range statement	
maximum	element has empty range		

	range not empty	add range alternative to range statement (must be disjunct)	
multipleOf		fraction-digits statement	
minLength/ maxLength		length statement	
pattern		pattern statement	
minItems/ maxItems		min-elements/max- elements statements	
uniqueItems set to true	if the 'parent' SDF definition is converted to a list node	unique statement mentioning all of the leaf/leaf-list nodes in the list node's sub- tree	
items		sub-statements of list/ leaf-list node corresponding to the item quality's 'parent' definition	
properties		child nodes of container/grouping node corresponding to the properties quality's 'parent' definition	
unit		units statement	
enum		type enumeration with enum statements for each string in the SDF	

		enum quality	
+-----+	+-----+	+-----+	+-----+
sdfType	has value 'byte-string'	built-in type 'binary'	
+-----+	+-----+	+-----+	+-----+
writable		config statement	
+-----+	+-----+	+-----+	+-----+

Table 2: Mapping SDF to YANG

### 3. Mapping from YANG to SDF

This section specifies one possible mapping for each of the YANG statements to SDF in detail. For reference on the individual YANG statements see [RFC7950] and [I-D.ietf-asdf-sdf] for SDF. Examples have been included where they serve to assist the reader's understanding of the conversion.

#### 3.1. Module

- \* YANG: [Section 7.1](#) (module) of [RFC7950]
- \* SDF:
  - [Section 3.1](#) (information block) of [I-D.ietf-asdf-sdf]
  - Sections [3.2](#) and [4](#) (namespaces section) of [I-D.ietf-asdf-sdf]

The module statement in YANG subsumes all other statements included in a module. After conversion the SDF model as a whole corresponds to the YANG module. The argument of the namespace statement of the YANG module is added to the SDF namespace quality together with the argument of the prefix statement of the YANG module which also becomes the entry of the defaultNamespace quality in the SDF model. Additionally, the namespaces and prefixes of each of the modules mentioned in the import statements are added to the namespace quality of the SDF model. Libyang loads the imported modules automatically and in the correct version. These modules are then also converted and stored so their definitions can be referenced via the sdfRef common quality when necessary. Figure 2 and Figure 1 illustrate these mappings.

The contents of the organization, contact and yang-version statements are stored alongside the description of the YANG module in a special sdfData definition designated to hold information on the module that does not fit into the SDF information block. This is done in with a conversion note to facilitate round trips in the future as described in [Section 5.2](#). To illustrate this conversion, Figure 2 contains a converted model with an sdfData definition called ietf-foo-info. The original YANG module can be found in Figure 1. The description of the module is scanned for information regarding copyright and licensing which are then transferred to the copyright and license qualities of the information block in the SDF model. The version quality of the information block is set to the first revision date given in the YANG revision statement. All other revision dates are ignored as of now.

YANG modules can define features via the feature statement to make parts of the module conditional. The abilities of a server are checked against the features stated in the module. Nodes reference features as an argument to the if-feature statement. If a server does not support a certain feature, nodes that reference that feature are ignored by the server. Since this functionality cannot be represented in SDF yet, YANG features are stored in the description of the sdfData definition designated to hold information on the module. The conversion note that is added to the descriptions looks as described in [Section 5.2](#).

If the deviation statement (introducing a deviation from the original YANG module) is present in the YANG module, Libyang applies the deviation directly and the converter converts the module that way. The presence of the deviation in the original YANG module is not indicated in the resulting SDF model as of now which might cause inconsistencies after round trips. This is not believed to be of great importance because deviations are supposed to only occur in unpublished modules.

```
module ietf-foo {  
    namespace "urn:ietf:params:xml:ns:yang:ietf-foo";  
    prefix "foo";  
    organization "Foo Inc.";  
    contact "foo@mail.com";
```

```

description
  "This is an example module

  Copyright Foo Inc.

  License XY";
revision 2016-03-20;
feature bar;
feature baz;

// ... more statements
}

```

Figure 1: Example YANG module

```

{
  "defaultNamespace": "foo",
  "info": {
    "copyright": "Copyright Foo Inc.",
    "license": "License XY",
    "title": "ietf-foo",
    "version": "2016-03-20"
  },
  "namespace": { "foo": "urn:ietf:params:xml:ns:yang:ietf-foo" },
  "sdfData": {
    "ietf-foo-info": {
      "description": "This is an example module\n\nCopyright Foo I
    }
  }
}

```

Figure 2: SDF conversion of YANG module from the last figure

### [3.2.](#) Submodule

- \* YANG: [Section 7.2](#) (submodule) of [[RFC7950](#)]

If a complex YANG module is composed of several components, the single components can be represented via the submodule statement. For conversion, the nodes of a submodule that is included into its super-module with the include statement are integrated into the super-module and converted that way. This is due to the way Libyang represents included submodules. Submodules on their own cannot be converted since Libyang does not parse files that solely contain a submodule.

### [3.3.](#) Container Statement

- \* YANG: [Section 7.5](#) (container) of [[RFC7950](#)]
- \* SDF:
  - Sections [2.2.1](#) and [5.1](#) (sdfObject) of [[I-D.ietf-asdf-sdf](#)]
  - Sections [2.2.6](#) and [6.3](#) (sdfThing) of [[I-D.ietf-asdf-sdf](#)]

YANG uses container nodes to group together other nodes. Containers on the top-level of a module are converted to sdfObject definitions. This is illustrated in the definition called level0 in Figure 3 and Figure 4. A container that is a direct child node to a top-level container is converted to a compound-type sdfProperty definition inside an sdfObject, as illustrated in the definition called level1 in Figure 3 and Figure 4. Any other container becomes an entry to

the properties quality of the compound-type definition corresponding to the parent node of the container. An example of this mapping can be found in Figure 3 and Figure 4 in the definition called level2.

Since the first SDF Internet-Draft did not contain the compound-type as a possible argument to the type quality, containers used to be translated to sdfThing definitions. This, was not a very suitable conversion semantically, however. At that time, sdfThings were the only elements that could contain elements of the same class, that is sdfThings could contain other sdfThings. This ability is required to represent the tree structure of YANG where, for example, containers

can contain other containers. In the second SDF Internet-Draft the compound-type was introduced. This feature effectively makes it possible for elements of the `sdfData` and `sdfProperty` classes to contain elements that share the same qualities.

A sub-statement to the container statement that cannot be represented in SDF as of now is the optional presence statement. The argument of the presence statement assigns a meaning to the presence or absence of a container node in an instance of the module. This concept is expressed in the description of the SDF definition in question as shown in [Section 5.2](#). This is also illustrated in the definition `level2` in Figure 3 and Figure 4.

```
module container-example {  
  // [...]  
  container level0 {  
    container level1 {  
      container level2 {  
        presence "Enables SSH";  
        // [...]  
      }  
    }  
  }  
}
```

Figure 3: YANG module with multiple nested container statements

```
{  
  ; [...]  
  "sdfObject": {  
    "level0": {
```



```

    "sdfProperty": {
      "level1": {
        "properties": {
          "level2": {
            "properties": {
              "description": "!Conversion note: present
              ; [...]
            },
            "type": "object"
          }
        },
        "type": "object"
      }
    }
  }
}

```

Figure 4: SDF conversion of the YANG module from the last figure

### 3.4. Leaf Statement

- \* YANG: [Section 7.6](#) (leaf) of [[RFC7950](#)]
- \* SDF:
  - Sections [2.2.2](#) and [5.2](#) (sdfProperty) of [[I-D.ietf-asdf-sdf](#)]
  - [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

Leaf nodes in YANG represent scalar variables. If a leaf statement occurs at the top-level of the module or as a direct child node of a top-level container (which is converted to sdfObject) it becomes an sdfProperty. On any other level a leaf is mapped to an entry of the properties quality of the compound-type definition corresponding to the parent node of the leaf. In both cases the SDF type quality is set to one of the simple data types because leaf nodes can only have simple data types. Leaf nodes can be assigned default values which are used in case the node does not exist in an instance of the YANG module. The default value of a leaf is converted to SDF through the quality default. The units sub-statement of a leaf node in YANG becomes the SDF quality unit. An example of such a conversion can be found in the level0 element in Figure 5 and Figure 6. The SDF quality unit is constrained to the SenML unit names. Although it

could cause conformance issues, the content of the YANG units statement is not processed to fit the SenML unit names as of now. This is due to the low probability that a unit from a YANG module is not listed in the SenML unit names in comparison to the time required to implement a mechanism to check conformance and convert non-conforming units. This feature might be added in later versions of the converter. YANG leaf nodes can be marked as mandatory to occur in an instance of the module by the mandatory statement. The statement takes true and false as arguments. This can easily be mapped to SDF through the sdfRequired quality. A reference to the SDF equivalent of the mandatory YANG leaf node is added to the sdfRequired quality of the containing sdfObject. If a mandatory leaf is transformed to an entry in the properties quality of a compound-type definition in SDF, said entry is mentioned in the required quality. If the sdfRequired or required quality does not already exist it is added at this point. The latter is demonstrated in the level2 element in Figure 5 and Figure 6.

```
module leaf-example {  
  // [...]  
  leaf level0 {  
    type int32;  
    units "kg";  
    default 14;  
  }  
  container dummy0 {  
    leaf level1 { type string; }  
    container dummy1 {  
      leaf level2 {  
        type string;  
        mandatory true;  
      }  
    }  
  }  
}
```

Figure 5: YANG module containing multiple leaf statements

```
{
  ; [...]
  "sdfObject": {
    "dummy0": {
      "sdfProperty": {
        "dummy1": {
          "properties": {
            "level2": { "type": "string" }
          },
          "required": [ "level2" ],
          "type": "object"
        },
        "level1": { "type": "string" }
      }
    }
  },
  "sdfProperty": {
    "level0": {
      "default": 14,
      ; [...]
      "type": "integer",
      "unit": "kg"
    }
  }
}
```

Figure 6: SDF conversion of the YANG module from the last figure

### [3.5.](#) Leaf-List Statement

- \* YANG: [Section 7.7](#) (leaf-list) of [[RFC7950](#)]
- \* SDF:
  - Sections [2.2.2](#) and [5.2](#) (sdfProperty) of [[I-D.ietf-asdf-sdf](#)]
  - [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

Similarly to leaf nodes, leaf-list nodes hold data of simple types in YANG but as items in an array. As such, leaf-lists are converted to

sdfProperties if they occur on the top-level or one level below in a module. On any other level a leaf-list becomes an entry to the properties quality of the compound-type definition corresponding to the parent node of the leaf-list. In both cases the type is set to array. The items of the array are of simple data types since leaf-list nodes can only have simple data types as well. The minimal and maximal number of elements in a YANG leaf-list can be specified by the min-elements and max-elements sub-statements. This is analogue

to the minItems and maxItems qualities of SDF which are set accordingly by the converter. A leaf-list can specify whether the system or the user is responsible for ordering the entries of the leaf-list. This information is stored in the ordered-by statement in YANG which is represented in SDF by a remark in the description (as shown in [Section 5.2](#)) of the SDF equivalent to the leaf-list node in question. Since leaf-list nodes are just leaf nodes that can occur multiple times, the units and default statements of leaf-list nodes are converted as described for leaf nodes in [Section 3.4](#).

### [3.6](#). List Statement

- \* YANG: [Section 7.8](#) (list) of [[RFC7950](#)]
- \* SDF:
  - Sections [2.2.2](#) and [5.2](#) (sdfProperty) of [[I-D.ietf-asdf-sdf](#)]
  - [Section 4.7](#) (dataqualities) of [[I-D.ietf-asdf-sdf](#)]

The list statement of YANG is similar to the leaf-list statement. The only difference is that, opposed to leaf-lists, lists represent an assortment of `_nodes_` that can occur multiple times. Therefore, YANG lists are mapped to SDF similarly to leaf-lists. List nodes on the top-level or one level below become sdfProperties. On any other level a list is converted to an entry to the properties quality of the compound-type definition corresponding to the parent node of the list. The type is set to array for both alternatives. Since lists contain a set of nodes, the items of the corresponding array are of type object. The minimal and maximal number of elements in a list can be specified by the min-elements and max-elements sub-statements. This is analogue to the minItems and maxItems qualities of SDF which are set accordingly by the converter. List nodes in YANG can define

one or multiple keys leafs of the list via the key statement. There is no SDF quality that could represent this feature. To preserve the information the names of the list keys are stored in the description of the SDF definition in question as described in section [Section 5.2](#). The unique sub-statement of the YANG list defines a number of descendant leaf nodes of the list that must have a unique combination of values in a module instance. This concept can be partly represented through the uniqueItems quality of SDF. However, the boolean-typed uniqueItems quality only specifies that the items of an SDF array have to be unique with `_all_` of their values combined. The YANG statement unique specifies a `_selection_` of leaf node values in the list that must be unique when combined. Thus, in addition to setting the uniqueItems quality of the SDF equivalent of the YANG list to true, a conversion note is added to the SDF equivalents of all leafs that are mentioned in the unique statement.

This is done as shown in Section [Section 5.2](#). The ordered-by statement of a list is also preserved in a conversion note. An example conversion of a list node with the mentioned sub-statements to SDF can be found in Figure 7 and Figure 8.

```
list server {
  key "name";
  unique "ip";
  ordered-by user;
  min-elements 1;
  max-elements 100;
  leaf name { type string; }
  leaf ip { type string; }
}
```

Figure 7: YANG list node

```
"sdfProperty": {
  "server": {
    "description": "!Conversion note: key name!\n!Conversion note: o
    "items": {
      "properties": {
        "ip": {
          "description": "!Conversion note: unique!\n",
          "type": "string"
        },

```

```

        "name": { "type": "string" }
      },
      "type": "object"
    },
    "maxItems": 100.0,
    "minItems": 1.0,
    "type": "array",
    "uniqueItems": true
  }
}

```

Figure 8: SDF conversion of the YANG list node from the last figure

### 3.7. Grouping Statement

- \* YANG: [Section 7.12](#) (grouping) of [[RFC7950](#)]
- \* SDF: [Section 5.5](#) (sdfData) of [[I-D.ietf-asdf-sdf](#)]

Grouping nodes are very similar to container nodes with the difference that the set of nodes contained in a grouping does not occur in the data tree unless the grouping has been referenced at

least once by a uses node. Thus, a grouping node is converted to a compound-type sdfData definition which defines a reusable definition that is not a declaration as well. The nodes inside the grouping are converted as entries to the properties quality in SDF. Figure 9 and Figure 10 contain an example conversion of a grouping.

### 3.8. Uses Statement

- \* YANG: [Section 7.13](#) (uses) of [[RFC7950](#)]
- \* SDF: [Section 4.4](#) (sdfRef) of [[I-D.ietf-asdf-sdf](#)]

A uses node has the purpose of referencing a grouping node. The set of child nodes of the referenced grouping are copied to wherever the uses node is featured. Some of the sub-statements of the referenced grouping can be altered via the refine statement of the uses node. In SDF a uses node is represented by the sdfRef quality which is added to the definition in question. As an argument the sdfRef contains a reference to the sdfData definition corresponding to the

grouping referenced by the uses node. If the uses node contains a refine statement, the specified refinements are also applied in the target SDF definition. An example for such a conversion is illustrated in Figure 9 and Figure 10.

```
module restaurant {  
    // [...]  
    grouping dish {  
        leaf name { type string; }  
        leaf price { type int32; }  
    }  
    list menu {  
        // [...]  
        uses dish {  
            refine name { mandatory true; }  
        }  
    }  
}
```

Figure 9: YANG module with uses and grouping statements

```
{  
    ; [...]  
    "sdfData": {  
        "dish": {  
            "properties": {  
                "name": { "type": "string" },  
                "price": {  
                    ; [...]  
                    "type": "integer"  
                }  
            }  
        },  
        "type": "object"  
    }
```

```

    }
  },
  "sdfProperty": {
    "menu": {
      "items": {
        "properties": {
          "dish": {
            "sdfRef": "#/sdfData/dish",
            "required": [ "name" ],
          }
        }
      }
    },
    "type": "object"
  },
  "type": "array"
}
}
}

```

Figure 10: SDF conversion of the YANG module from the last figure

### 3.9. Choice Statement

- \* YANG: [Section 7.9](#) (choice) of [[RFC7950](#)]
- \* SDF: [Section 4.7.2](#) (sdfChoice) of [[I-D.ietf-asdf-sdf](#)]

Conversion of the choice statement from YANG is simple since it is similar to the sdfChoice quality. The choice statement is used to define alternative sub-trees for the node the choice occurs in. Only one of the alternatives is present in the data tree. A YANG choice is converted to an sdfProperty if it occurs on top-level or one level below, like the snack definition in Figure 11 and Figure 12. On any other level a choice is mapped to an entry of the properties quality of the compound-type definition corresponding to the parent node of the choice. The food-level2 definition in Figure 11 and Figure 12 is an example of this kind of mapping. The SDF equivalent of the choice

contains the sdfChoice quality. Case or other child nodes of the choice are mapped to SDF as one of the named alternatives of the sdfChoice each. What cannot be represented is the default sub-statement of the YANG choice that defines which of the alternatives is considered the default one. This information is preserved in a



conversion note as described in [Section 5.2](#).

```
container food {
  container food-level2 {
    choice dinner {
      default home-cooked;
      case restaurant {
        leaf steak { type boolean; }
        leaf pizza { type boolean; }
      }
      case home-cooked {
        leaf pasta { type boolean; }
      }
    }
  }
  choice snack {
    case sports-arena {
      leaf pretzel { type boolean; }
      leaf beer { type boolean; }
    }
    case late-night {
      leaf chocolate { type boolean; }
    }
  }
}
```

Figure 11: YANG container using the choice statement

```

"sdfObject": {
  "food": {
    "sdfProperty": {
      "food-level2": {
        "properties": {
          "dinner": {
            "description": "!Conversion note: default home-c
            "sdfChoice": {
              "home-cooked": {
                "properties": {
                  "pasta": { "type": "boolean" }
                },
                "type": "object"
              },
              "restaurant": {
                "properties": {
                  "pizza": { "type": "boolean" },
                  "steak": { "type": "boolean" }
                },
                "type": "object"
              }
            }
          },
          "type": "object"
        },
        "snack": {
          "description": "!Conversion note: default late-night!\n"
          "sdfChoice": {
            "late-night": {
              "properties": {
                "chocolate": { "type": "boolean" }
              },
              "type": "object"
            },
            "sports-arena": {
              "properties": {
                "beer": { "type": "boolean" },
                "pretzel": { "type": "boolean" }
              },
              "type": "object"
            }
          }
        }
      }
    }
  }
}

```

Figure 12: SDF conversion of the YANG container from the last figure

### [3.10.](#) RPC Statement

- \* YANG: [Section 7.14](#) (rpc) of [[RFC7950](#)]
- \* SDF: Sections [2.2.3](#) and [5.3](#) (sdfAction) of [[I-D.ietf-asdf-sdf](#)]

Remote procedure calls (RPCs) can be modeled in YANG with rpc nodes which have up to one input child node holding the commands input data and up to one output node for the output data. In YANG RPCs can only occur on the top-level because in contrast to actions in YANG they do not belong to a container. This can easily be represented by sdfActions. The corresponding sdfAction is not placed inside an sdfObject or sdfThing but at the top-level of the SDF model to represent independence from a container. The input node of the RPC is converted to the sdfInputData quality of the sdfAction which is of type object. Equivalently, the output node of the RPC becomes the sdfOutputData of the sdfAction, which is also of type object. Groupings and typedefs in the RPC are converted to sdfData definitions inside the sdfAction.

### [3.11.](#) Action Statement

- \* YANG: [Section 7.15](#) (action) of [[RFC7950](#)]
- \* SDF: Sections [2.2.3](#) and [5.3](#) (sdfAction) of [[I-D.ietf-asdf-sdf](#)]

Action nodes in YANG work similarly to rpc nodes in the way that they are used to model operations that can be invoked in the module and also have up to one input and output child node respectively. As mentioned before, YANG actions are affiliated to a container. The representation of this affiliation is not quite trivial because YANG containers are not translated to sdfObjects in all cases. Only sdfObjects can have sdfActions, however. If an action occurs in a container that is a below-top-level container (and thus not converted to sdfObject), as illustrated in Figure 13, the affiliation cannot be represented directly in SDF as of now. Figure 14 shows how an XML instance of calling the action in Figure 13 and the reply would look like. As an input, the action specifies the container server it is affiliated to and its name. The actual action, reset and the value of its input, reset-at are specified inside the container instance.

The result after converting the container from Figure 13 to SDF can be found in Figure 15: To ensure equivalence of model instances a copy of the contents of the converted container is set as the `sdfInputData` of the `sdfAction`. The `sdfInputData` is of type object. The conversion of the actual action along with its input is added to the copy of the container conversion as another entry to its

properties quality. Furthermore, a conversion note is added as described in [Section 5.2](#). Equivalently, the output nodes of the action become the `sdfOutputData` of the `sdfAction` which is also of type object. Groupings and typedefs in the action node are converted to `sdfData` definitions inside the `sdfAction`.

```

container example-container {}
  container server {
    leaf name { type string; }
    action reset {
      input {
        leaf reset-at { type string; }
      }
      output {
        leaf reset-finished-at { type string; }
      }
    }
  }
}

```

Figure 13: YANG container using the action statement

```

<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <action xmlns="urn:ietf:params:xml:ns:yang:1">
    <server xmlns="urn:example:server-farm">
      <name>apache-1</name>
      <reset>
        <reset-at>2014-07-29T13:42:00Z</reset-at>
      </reset>
    </server>
  </action>
</rpc>

<rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <reset-finished-at xmlns="urn:example:server-farm">

```

```
2014-07-29T13:42:12Z
</reset-finished-at>
</rpc-reply>
```

Figure 14: XML instance of the action from the last figure

```
"sdfObject": {
  "example-container": {
    "sdfAction": {
      "reset": {
        "description": "Action connected to server\n\n",
        "sdfInputData": {
          "properties": {
            "server": {
              "properties": {
                "name": { "type": "string" },
                "reset": {
                  "properties": {
                    "reset-at": { "type": "string" }
                  },
                  "type": "object"
                }
              },
              "type": "object"
            }
          },
          "required": [ "server" ],
          "type": "object"
        },
        "sdfOutputData": {
          "properties": {
            "reset-finished-at": { "type": "string" }
          },
          "type": "object"
        }
      }
    }
  }
}
```

```

    }
  },
  "sdfProperty": {
    "server": {
      "properties": {
        "name": { "type": "string" }
      },
      "type": "object"
    }
  }
}

```

Figure 15: SDF conversion of the YANG container from Figure 13

### 3.12. Notification Statement

- \* YANG: [Section 7.16](#) (notification) of [[RFC7950](#)]

- \* SDF: Sections [2.2.4](#) and [5.4](#) (sdfEvent) of [[I-D.ietf-asdf-sdf](#)]

In YANG, notification nodes are used to model notification messages. Notification nodes are converted to sdfEvent definitions. Their child nodes are converted to the sdfOutputData of the sdfEvent which is of type object. Groupings and typedefs in the notification node are converted to sdfData definitions inside the sdfEvent.

### 3.13. Augment Statement

- \* YANG: [Section 7.17](#) (augment) of [[RFC7950](#)]
- \* SDF: [Section 4.6](#). (common qualities) of [[I-D.ietf-asdf-sdf](#)]

The augment statement can either occur at the top-level of a module to add nodes to an existing target module or sub-module, or in a uses statement to augment the targeted and thus integrated grouping. The conversion of the augment statement to SDF is not trivial because SDF does not feature this mechanism.

The tool used to deserialize YANG modules, Libyang, adds the nodes

into the target of the augment statement automatically for targets that are modules or sub-modules. This is adopted in the mapping: The SDF model that corresponds to target of the the augment statement is converted with the augmentation already applied. A conversion note is added to the description as described in [Section 5.2](#) to preserve where the augmentation was issued from. This mapping is illustrated in Figure 16, Figure 17 and Figure 18. If the resulting SDF model has to be converted back to YANG, definitions that are marked as augmentations are converted back accordingly. This way of mapping the augment statement to SDF causes problems if the augmentation target lies within a module whose converted version is already available and should not be replaced. Because, as of now, SDF does not offer means to extend already existing models retroactively these augmentations cannot be converted to SDF.

When the target of the augment is a grouping the augmentation cannot be represented in SDF, either. The reason for this is that grouping nodes are converted to SDF definitions with the type object. The nodes inside the grouping are converted with the help of the properties quality. It is currently not possible to add properties to the properties quality, it can only be overridden as a whole.

```
module example-module {
  // [...]
  leaf leaf1 { type string; }
}
```

Figure 16: YANG module that serves as an augmentation target

```
module augmenting-module {
  // [...]
  augment "/example" {
    leaf additional-leaf { type string; }
  }
}
```

Figure 17: YANG module using the augment statement on the module from the last figure

```
{
  ; [...]
```

```

    "sdfProperty": {
      "leaf1": { "type": "string" },
      "additional-leaf": {
        "description": "!Conversion note: augmented-by augmenting-mo
        "type": "string"
      }
    }
  }
}

```

Figure 18: SDF conversion of the YANG module from Figure 16 after conversion of the YANG module from Figure 17

### 3.14. Anydata and Anyxml Statements

- \* YANG: Sections [7.10](#) and [7.11](#) (augment) of [[RFC7950](#)]
- \* SDF: [Section 4.6](#) (common qualities) of [[I-D.ietf-asdf-sdf](#)]

The anydata and anyxml statements are designated for nodes in the schema tree whose structure is unknown at the design time of the module or in general. Since this is not a concept that can be represented in SDF as of now, anydata and anyxml nodes are not converted. Instead, to preserve the information a conversion note is added to the SDF element corresponding to the parent node of the anydata or anyxml node as described in [Section 5.2](#).

### 3.15. Type Statement

- \* YANG: [Section 7.4](#) (type) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

The type statement of YANG is used to specify the built-in or derived type used by a leaf or typedef node. Mapping this statement to YANG is trivial if the argument is a simple data type because the SDF data qualities also contain a type quality. A derived type used as an argument to the YANG type statement is converted via the sdfRef quality. As an argument, the sdfRef quality contains a reference to the sdfData definition corresponding to the derived type. If the



derived type is restricted, for example with the length statement, the restrictions are converted as they would be for the base type and added to the SDF definition containing the type in question.

There are multiple sub-statements to the type statement that depend on its value. The conversion of those sub-statements is discussed in the section of the built-in type the sub-statement belongs to.

### 3.16. String Built-In Type

- \* YANG: [Section 9.4](#) (string) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

The YANG built-in type string is converted to the SDF built-in type string. Strings in YANG can be restricted in length and by regular expressions.

The length statement can specify either a constant length, a lower inclusive length, an upper inclusive length or both a lower and upper inclusive length. A length statement can also specify more than one disjoint constant length or length ranges. The values min and max in a length statement represent the minimum and maximum lengths accepted for strings. If the length statement in YANG does not contain a constant value but a length range it is converted to the minLength and maxLength SDF qualities. This is illustrated in Figure 19 and Figure 20. If a constant value is defined through the YANG length statement the minLength and maxLength qualities are set to the same value. If the length statement specifies multiple length ranges or constant values the sdfChoice quality is used for conversion. The named alternatives of the sdfChoice contain the single converted length ranges or constant values each. If the min and max values are present in the YANG length statement they are converted to the respective minimum and maximum lengths accepted for strings.

The YANG pattern statement can be used to hold regular expressions that the affiliated string has to match. To patterns from YANG in SDF the pattern quality can be used. One problem in the conversion of patterns is that YANG strings can be restricted by multiple patterns but SDF strings can have at most one pattern. To represent multiple patterns from YANG in SDF the patterns are combined into one

regular expression with the help of positive look-ahead. Figure 19 contains an example leaf of type string with multiple defined patterns which is converted as shown in Figure 20. This does not always convey the meaning of the original regular expression. Another issue is the possibility to declare invert-match patterns in YANG. These types of patterns are converted to SDF by adding negative look-ahead to the regular expression, as illustrated in Figure 21 and Figure 22. To preserve the original patterns and to facilitate round trips, the original patterns are stored with a conversion note in the description of the containing definition as described in section [Section 5.2](#).

```
leaf example {
  type string {
    length "1..4";
    pattern "[0-9]*";
    pattern "[a-z]*";
  }
}
```

Figure 19: YANG leaf node with type string, multiple pattern statements and a length statement

```
"sdfProperty": {
  "example": {
    "description": "!Conversion note: pattern [0-9]*!\n!Conversion n
    "maxLength": 4.0,
    "minLength": 1.0,
    "pattern": "(?=[0-9]*)[a-z]*",
    "type": "string"
  }
}
```

Figure 20: SDF conversion of the YANG leaf from the last figure

```
leaf example {
  type string {
    pattern "[0-9]*" { modifier invert-match; }
  }
}
```

Figure 21: YANG leaf definition with type string and an invert-match pattern

```
"sdfProperty": {
  "example": {
    "description": "!Conversion note: pattern [0-9]*!\n",
    "pattern": "((?!([0-9]*)).)*",
    "type": "string"
  }
}
```

Figure 22: SDF conversion of the YANG leaf from the last figure

Another, more general problem regarding the conversion of regular expressions from YANG to SDF is the fact that YANG uses a regular expression language as defined by W3C Schema while SDF adopts ECMAScript regular expressions. Both regular expression languages share most of their features. Since this does not cause problems in most cases and regarding the time constraints of this thesis, this issue is not given any further attention beyond what was stated in this paragraph. There is, however, a project of the IETF Network Working Group to create an interoperable regular expression format. Once the work on the draft has progressed the format might be adopted by the SDF/YANG converter.

### 3.17. Decimal64 Built-In Type

- \* YANG: [Section 9.3](#) (decimal64) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

The decimal64 built-in type of YANG is converted to the number type in SDF. A decimal64 type in YANG has a mandatory fraction-digits sub-statement that specifies the possible number of digits after the decimal separator. The value of the fraction-digits statement is converted to the multipleOf quality of SDF which states the resolution of a number, that is the size of the minimal distance between number values. Figure 23 and Figure 24 contain examples for the conversion of the decimal64 built-in type.

A YANG decimal64 type can be restricted by means of the range statement specifying either a constant value, a lower inclusive bound, an upper inclusive bound or both a lower and upper inclusive value. The range statement can also be used to specify multiple disjoint constant values or ranges. The min and max key words in a range statement represent the minimum and maximum values of the type

in question. If the range statement in YANG contains a range and not a constant value it is converted to the minimum and maximum data qualities in SDF. This is illustrated in the definition called my-sensor-value in the example. If a constant value is defined through the YANG range the SDF const quality is set accordingly, as shown for

the definition room-temperature in the example. If the range specifies multiple ranges or constant values the sdfChoice quality is used for conversion. The named alternatives of the sdfChoice contain the single converted ranges or constant values each. An example for this conversion can be found in the my-sensor-value3 example definition. If the min and max values are present in the YANG range they are converted to the respective minimum and maximum values for the type in question, as shown for the max value in the example definition my-sensor-value2.

```
module decimal64-example {
  // [...]
  leaf my-sensor-value {
    type decimal64 {
      fraction-digits 2;
      range "-50.0..150.0";
    }
  }
  leaf my-sensor-value2 {
    type decimal64 {
      fraction-digits 4;
      range "0..max";
    }
  }
  leaf my-sensor-value3 {
    type decimal64 {
      fraction-digits 6;
      range "0.0..1.0 | 5.0";
    }
  }
  leaf room-temperature {
    type decimal64 {
      fraction-digits 1;
      range "21.5";
    }
  }
}
```

```
}
```

Figure 23: YANG module using the decimal64 built-in type

```
{
  ; [...]
  "sdfProperty": {
    "my-sensor-value": {
      "maximum": 150.0,
      "minimum": -50.0,
      "multipleOf": 0.01,
      "type": "number"
    },
    "my-sensor-value2": {
      "maximum": 3.3999999521443642e+38,
      "minimum": 0.0,
      "multipleOf": 0.0001,
      "type": "number"
    },
    "my-sensor-value3": {
      "sdfChoice": {
        "range_option_1": {
          "maximum": 0.0,
          "minimum": 1.0,
          "multipleOf": 0.000001,
          "type": "number"
        },
        "range_option_2": {
          "const": 5.0,
          "multipleOf": 0.000001,
          "type": "number"
        }
      }
    }
  }
}
```

```

    },
    "room-temperature": {
        "const": 21.5,
        "multipleOf": 0.1,
        "type": "number"
    }
}

```

Figure 24: SDF conversion of the YANG module from the last figure

### 3.18. Integer Built-In Types

- \* YANG: [Section 9.2](#) (integer) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

In YANG there are 8 different integer types: int8, uint8, int16, uint16, int32, uint32, int64 and uint64. Each of them is converted to type integer in SDF. A conversion note specifying the exact type is added as described in [Section 5.2](#). Additionally, the minimum and maximum qualities of the SDF definition that the converted type belongs to are set to the respective minimum and maximum values of the integer type in question. If the YANG type also specifies a range, the minimum and maximum SDF qualities are altered accordingly. Like the decimal64 YANG built-in type, the YANG integer types can also be restricted by a range statement. The integer range statement is converted as described in [Section 3.17](#).

```

leaf example {
    type int32;
}

```

Figure 25: YANG leaf with the int32 built-in type

```

"sdfProperty": {
    "example": {
        "description": "!Conversion note: type int32!\n",
        "maximum": 2147483647,

```

```

        "minimum": -2147483648,
        "type": "integer"
    }
}

```

Figure 26: SDF conversion of the YANG leaf from the last figure

### 3.19. Boolean Built-In Type

- \* YANG: [Section 9.5](#) (boolean) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

The YANG boolean built-in type holds a boolean value, that is one of either true or false. It is converted to the SDF boolean type. There are no further sub-statements to this type in YANG.

### 3.20. Binary Built-In Type

- \* YANG: [Section 9.8](#) (binary) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

To represent binary data, the YANG built-in type binary can be used. If the argument of the YANG type statement is binary the SDF type quality is set to string. In addition, the sdfType quality is set to

byte-string. A YANG binary can have a sub-statement restricting its length. This is converted to SDF via the minLength and maxLength qualities. Like the string YANG built-in type, the binary type can also be restricted by a length statement. This length statement is converted as described in [Section 3.16](#).

### 3.21. Enumeration Built-In Type

- \* YANG: [Section 9.6](#) (enumeration) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

The YANG built-in type enumeration is used to map string-valued alternatives to integer values. Additionally each string can have a description and other sub-statements. SDF also specifies an enum

quality which is used to represent YANG enumerations. The SDF enum quality only holds an array of strings. All other information is stored in conversion notes in the description of the SDF definition the enum belongs to, as specified in [Section 5.2](#).

### [3.22](#). Bits Built-In Type

- \* YANG: [Section 9.8](#) (bits) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

SDF does not specify a built-in type to represent a set of named bits and their positions like YANG does with its built-in type bits. Therefore, this built-in type has to be converted to SDF type object with one entry to the properties quality of type boolean for each bit. The property is named after the name of the bit. The position of the bit is stored in a conversion note as described in [Section 5.2](#). An example conversion of a leaf with type bits to SDF can be found in Figure 27 and Figure 28.

```
leaf example {  
  type bits {  
    bit auto-adapt {  
      description "1 if automatic adaption is enabled, 0 otherwise"  
      position 1;  
    }  
    bit battery-only { position 2; }  
    bit disable-sensor { position 0; }  
  }  
}
```

Figure 27: YANG leaf that is of the built-in type bits

```
"sdfProperty": {  
  "example": {  
    "description": "!Conversion note: type bits!\n",  
    "properties": {  
      "auto-adapt": {  
        "description": "Bit at position 1: 1 if automatic adapti  
        "type": "boolean"  
      },  
    },  
  },  
}
```



```

        "battery-only": {
            "description": "Bit at position 2",
            "type": "boolean"
        },
        "disable-sensor": {
            "description": "Bit at position 0",
            "type": "boolean"
        }
    },
    "type": "object"
}

```

Figure 28: SDF conversion of the YANG leaf from the last figure

### [3.23.](#) Union Built-In Type

- \* YANG: [Section 9.12](#) (union) of [[RFC7950](#)]
- \* SDF: [Section 4.7.2](#) (sdfChoice) of [[I-D.ietf-asdf-sdf](#)]

YANG unions hold a set of alternatives for the type statement. Although the union built-in type of YANG does not exist as a built-in type in SDF, its meaning can be easily represented by the sdfChoice quality. The sdfChoice corresponding to the union contains a set of named alternatives each named after the respective type in the YANG union and each containing nothing but the SDF type quality set to the SDF equivalent of the respective type. Figure 29 and Figure 30 illustrate this mapping.

```

leaf example {
    type union {
        type string;
        type boolean;
    }
}

```

Figure 29: YANG leaf that uses the union built-in type

```

"sdfProperty": {

```

```

    "example": {
      "description": "!Conversion note: type union!\n",
      "sdfChoice": {
        "boolean": {
          "type": "boolean"
        },
        "string": {
          "type": "string"
        }
      }
    }
  }
}

```

Figure 30: SDF conversion of the YANG leaf from the last figure

### 3.24. Leafref and Identityref Built-In Types

- \* YANG: [Section 9.9](#) (leafref) of [\[RFC7950\]](#) [Section 9.10](#) (identityref) of [\[RFC7950\]](#)
- \* SDF: [Section 4.4](#) (sdfRef) of [\[I-D.ietf-asdf-sdf\]](#)

The YANG built-in types leafref and identityref are used to reference a leaf node or identity definition respectively. They are represented in SDF by the sdfRef quality. As an argument said sdfRef quality contains a reference to the SDF element corresponding to the target of the leafref or identityref statement.

### 3.25. Empty Built-In Type

- \* YANG: [Section 9.11](#) (empty) of [\[RFC7950\]](#)
- \* SDF: [Section 4.7](#) (data qualities) of [\[I-D.ietf-asdf-sdf\]](#)

Another concept that is not contained in SDF directly is that of the YANG built-in type empty. YANG elements with this type convey meaning by their mere existence or non-existence. This is represented in SDF using the compound-type with an empty set of properties.

### 3.26. Instance-Identifier Built-In Type

- \* YANG: [Section 9.13](#) (instance-identifier) of [\[RFC7950\]](#)

The instance-identifier built-in type of YANG is used to refer to a particular instance of a node in the data tree. As of now, it cannot be represented functionally in SDF because there is currently no

possibility to refer to specific instances of SDF definitions. This feature might be added to SDF in the future. For now, this type is represented by the string built-in type of SDF. Furthermore, a conversion note is added to the resulting SDF definition as specified in [Section 5.2](#). .

### [3.27](#). Typedef Statement

- \* YANG: [Section 9.3](#) (typedef) of [[RFC7950](#)]
- \* SDF: [Section 4.4](#) (sdfRef) of [[I-D.ietf-asdf-sdf](#)]

The typedef statement has the purpose to define derived types in YANG. The SDF class sdfData is used to represent typedefs after conversion. The usage of a derived type via the type statement is converted to an sdfRef to the corresponding sdfData definition. If a derived type is restricted according to its base type, for example with a range statement, the restrictions are converted as they would be for the base type and added to the sdfData definition.

### [3.28](#). Identity Statement

- \* YANG: [Section 7.18](#) (identity) of [[RFC7950](#)]
- \* SDF: [Section 5.5](#) (sdfData) of [[I-D.ietf-asdf-sdf](#)]

The YANG identity statement is used to denote the name and existence of an identity. Identities can be based on one or more other identities. They are referenced with the identityref statement. This concept is converted to SDF by sdfData definitions for each identity. If an identity is based on one other identity this is represented by an sdfRef reference to the sdfData definition corresponding to the base identity. If an identity has multiple base identities it is converted to a compound-type sdfData definition with one property for each base identity. Each property contains an sdfRef reference to the sdfData definition corresponding to one of the base identities.

### [3.29](#). Config Statement

- \* YANG: [Section 7.21.1](#) (config) of [[RFC7950](#)]
- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]

The config statement of YANG can have the boolean values true or false as arguments. If config is set to true the element containing

the config statement represents readable and writable configuration data. If the config statement is set to false the element containing

the statement represents read-only state data. This is transferred to SDF via the readable and writable qualities. If the config statement is set to true it is mapped to the readable and writable qualities both being set to true. If the config statement is set to false it is converted by setting the readable quality to true and the writable quality to false. There are, however, cases in which the SDF definition corresponding to the YANG element containing the config statement is not one that can use data qualities. This is the case, for example, if a top-level container, which is converted to `sdfObject`, holds a config statement. In this case, all definitions inside the `sdfObject` that can use data qualities have the readable and writable qualities set as described above.

### [3.30.](#) Status Statement

- \* YANG: [Section 7.21.2](#) (status) of [[RFC7950](#)]
- \* SDF: [Section 4.6](#) (common qualities) of [[I-D.ietf-asdf-sdf](#)]

The status statement of YANG is used to express whether a definition is either current, deprecated or obsolete. In SDF there is no quality with a similar meaning. Thus, the YANG status statement is represented by a conversion note in the description of the SDF definition corresponding to the YANG element the status statement occurred in as described in [Section 5.2](#).

### [3.31.](#) Reference Statement

- \* YANG: [Section 7.21.4](#) (reference) of [[RFC7950](#)]
- \* SDF: [Section 4.6](#) (common qualities) of [[I-D.ietf-asdf-sdf](#)]

In YANG the reference statement holds a human-readable reference to an external document related to its containing YANG definition. This information is preserved through a conversion note in the description of the SDF definition equivalent to the node containing the reference statement as described in [Section 5.2](#).

### [3.32.](#) When and Must Statements

- \* YANG: [Section 7.5.3](#) (must) of [\[RFC7950\]](#) [Section 7.21.5](#) (when) of [\[RFC7950\]](#)
- \* SDF: [Section 4.6](#) (common qualities) of [\[I-D.ietf-asdf-sdf\]](#)

As mentioned before, YANG provides means to impose conditions on its definitions. If a node in the data tree has an unfulfilled must or when condition it is invalidated. Must and when conditions use XML

Path Language expressions to indicate dependencies. This feature is not realizable in SDF as of now and is thus preserved through conversion notes as described in [Section 5.2](#).

There is a query language similar to XML Path Language for JSON called JSONPath. If SDF adopts JSONPath or something similar in the future the converter can be extended to process the functionality of must and when statements.

### [3.33](#). Extension Statement

- \* YANG: [Section 7.19](#) (extension) of [\[RFC7950\]](#)
- \* SDF: [Section 4.6](#) (common qualities) of [\[I-D.ietf-asdf-sdf\]](#)

The extension statement in YANG has the purpose of defining new statements for the YANG language. This is not a concept that can be transferred to SDF yet. When an extension is used, this fact has to be stored in a conversion note in the description of the SDF definition that is analogue to the YANG definition containing the extension statement, as described in [Section 5.2](#). The definition of the extension is not converted.

## [4](#). Mapping from SDF to YANG

In this section the conversion of each element of SDF to YANG is explained in detail. For reference on the individual YANG statements see [\[RFC7950\]](#) and [\[I-D.ietf-asdf-sdf\]](#) for SDF. Examples have been inserted where they are necessary to understand the mapping.

### [4.1](#). Information Block

- \* SDF: [Section 3.1](#) (information block) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG: [Section 7.1](#) (module) of [[RFC7950](#)]

At the top of an SDF model the information block holds meta data, that is the title, version, copyright and license information, about the model. When mapping an SDF model to YANG, the content of the title quality is used as the name for the YANG module. For this, the title string has to be modified to only contain lower case letters, digits and the characters "\_", "-" and ".". If the version quality contains a date in the format `_month-day-year_` it is analogue to the revision statement of YANG and converted as such. The strings from the copyright and license qualities are stored in the description of the resulting YANG module since there are no dedicated YANG statements equivalent to these qualities.

#### [4.2.](#) Namespace Section

- \* SDF: Sections [3.2](#) and [4](#) (namespaces section) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG: [Section 7.1.3](#) (namespace) of [[RFC7950](#)] [Section 7.1.5](#) (import) of [[RFC7950](#)]

The purpose of the namespace section in an SDF model is to specify its (optional) namespace and the namespaces of external models whose definitions are referenced. The namespace section has a namespace quality mapping namespace URIs to a shortened name for that URI. The shortened name is also used as a prefix when referring to external definitions. If an SDF model is supposed to contribute globally available definitions, a value is given to the `defaultNamespace` quality and mapped to a namespace URI in the namespace quality. To map this to YANG, three of its statements are necessary: the import, the prefix and the namespace statement. To be able to use definitions from external modules in YANG, their names have to be declared by one import statement each. As a first step, each external SDF model that is mentioned in the namespace map also has to be converted to a YANG module. The default namespaces of the external SDF models are represented in the prefix sub-statement of the respective import statement. To represent the namespace and short name of the model, if present, the YANG namespace and prefix

statements that are set accordingly. Both are top-level statements.

#### [4.3.](#) SdfThing Quality

- \* SDF: Sections [2.2.6](#) and [6.3](#) (sdfThing) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG: [Section 7.5](#) (container) of [[RFC7950](#)]

An sdfThing definition holds the definition of a complex device that can be made up of multiple sdfObjects and multiple other sdfThings. SdfThings are converted to YANG container nodes. The sdf-spec extension is inserted to inform about the origin of the container as an sdfThing. This is necessary to facilitate round-trips because the container could also originate from an sdfObject.

#### [4.4.](#) SdfObject Quality

- \* SDF: Sections [2.2.1](#) and [5.1](#) (sdfObject) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG: [Section 7.5](#) (container) of [[RFC7950](#)]

SdfObject definitions are the main building blocks of an SDF model, grouping together definitions of the classes sdfProperty, sdfData, sdfAction and sdfEvent. They can also be used as arrays via their minItems and maxItems qualities. An sdfObject is mapped to a YANG container node if it is not defined as an array. Otherwise the sdfObject can be converted to a list node with the min-elements and max-elements statements set analogous to the minItems and maxItems qualities. This feature was only recently added to SDF and is thus not yet implemented neither in the SDF serializer/deserializer nor in the SDF/YANG converter.

#### [4.5.](#) Common Qualities

- \* SDF: [Section 4.6](#) (common qualities) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG:
  - [Section 7.21.3](#) (description) of [[RFC7950](#)]

- [Section 7.3](#) (typedef) of [\[RFC7950\]](#)
- [Section 9.9](#) (leafref) of [\[RFC7950\]](#)
- [Section 7.13](#) (uses) of [\[RFC7950\]](#)
- [Section 3](#) (terminology for mandatory) of [\[RFC7950\]](#)

The set of qualities that is grouped under the name of `_common` qualities\_ can be used to provide meta data for SDF definitions.

The description quality is converted to the YANG description statement. The label quality is ignored because it is identical to the identifier of the definition in most cases.

The `sdfRef` quality is supposed to hold references to other definitions whose qualities are then copied into the referencing definition. Qualities of the referenced definition can also be overridden by defining them again in the referencing definition. The conversion of an `sdfRef` depends on what is referenced by it and what that is converted to. Figure 31 and Figure 32, as well as Figure 33 and Figure 34 illustrate different conversions of the `sdfRef` quality. If the referenced definition is converted to a typedef the `sdfRef` is analogous to the type statement in YANG which points to the typedef. Overridden qualities can be represented by the respective sub-statements of the type which in turn override the sub-statements of the type of the typedef. This is the case for `simpleDataRef` in Figure 31 and Figure 32. If the referenced definition is mapped to a leaf or leaf-list node it can be referenced by the `leafref` built-in

type in YANG. This is the case for `simplePropertyRef` and `simpleArrayPropertyRef` in Figure 33 and Figure 34. In this case overridden qualities cannot be represented in SDF. If the YANG equivalent of the referenced definition is a grouping node the `sdfRef` is converted to a `uses` node which points to said grouping. The `uses` node is placed inside an additional container to preserve the name of the referencing SDF definition and to avoid sibling nodes with identical names (which is invalid in YANG). This is what is done for `compoundDataRef`, `simpleArrayDataRef` and `compoundArrayDataRef` in Figure 31 and Figure 32. In all other cases the YANG equivalent of the referenced SDF definition cannot be referenced directly but has



first to be packaged in a grouping node. This is done by first creating a grouping on the top-level of the module in order for the grouping to be available globally (in case it is also referenced in another model). The YANG node that is equivalent to the referenced SDF definition is copied into the new grouping and afterwards replaced with a uses node referencing the grouping. This is done to avoid redundancy. Lastly, the actual sdfRef is represented by another uses node referencing the newly created grouping. The uses node is placed inside a container node that represents the SDF definition that contains the sdfRef to preserve the name of the SDF definition. Furthermore, there cannot be two sibling nodes with the same name in YANG. The definitions compoundPropertyRef and compoundArrayPropertyRef in Figure 33 and Figure 34 are examples of such conversions. If SDF qualities of the referenced definition are overridden in the referencing definition this is represented with the refine statement which can be a sub-statement to uses node (see compoundArrayPropertyRef in Figure 33 and Figure 34).

```
{  
  ; [...]  
  "sdfObject": {  
    "ExampleObject": {
```

```

"sdfData": {
  "simpleData": { "type": "string" },
  "compoundData": {
    "type": "object",
    "properties": {
      "A": { "type": "string" },
      "B": { "type": "string" }
    }
  },
  "simpleArrayData": {
    "type": "array",
    "items": { "type": "string" }
  },
  "compoundArrayData": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "A": { "type": "string" },
        "B": { "type": "string" }
      }
    }
  }
},
"sdfProperty": {
  "simpleDataRef": {
    "sdfRef": "#/sdfObject/ExampleObject/sdfData/simpleData",
    "pattern": "[a-z]*"
  },
  "compoundDataRef": { "sdfRef": "#/sdfObject/ExampleObject/sdfData/compoundData" },
  "simpleArrayDataRef": { "sdfRef": "#/sdfObject/ExampleObject/sdfData/simpleArrayData" },
  "compoundArrayDataRef": { "sdfRef": "#/sdfObject/ExampleObject/sdfData/compoundArrayData" }
}
}
}
}

```

Figure 31: SDF model that uses the sdfRef with different sdfData definitions

```
module exampleModel {  
  // [...]  
  typedef simpleData { type string; }  
  grouping compoundArrayData {  
    helper:sdf-spec "sdfData";  
    list compoundArrayData {  
      config false;  
      leaf A { type string; }  
      leaf B { type string; }  
    }  
  }  
  grouping compoundData {  
    helper:sdf-spec "sdfData";  
    leaf A { type string; }  
    leaf B { type string; }  
  }  
  grouping simpleArrayData {  
    helper:sdf-spec "sdfData";  
    leaf-list simpleArrayData { type string; }  
  }  
  container ExampleObject {  
    helper:sdf-spec "sdfObject";  
    container compoundArrayDataRef {  
      helper:sdf-spec "sdfProperty";  
      uses compoundArrayData;  
    }  
    container compoundDataRef {  
      helper:sdf-spec "sdfProperty";  
      uses compoundData;  
    }  
    container simpleArrayDataRef {  
      helper:sdf-spec "sdfProperty";  
      uses simpleArrayData;  
    }  
    leaf simpleDataRef {  
      type simpleData { pattern "[a-z]*"; }  
    }  
  }  
}
```

Figure 32: YANG conversion of the SDF model from the last figure

```

{
  ; [...]
  "sdfObject": {
    "ExampleObject2": {
      "sdfProperty": {
        "simpleProperty": { "type": "string" },
        "compoundProperty": {
          "type": "object",
          "properties": {
            "A": { "type": "string" },
            "B": { "type": "string" }
          }
        },
        "simpleArrayProperty": {
          "type": "array",
          "items": { "type": "string" }
        },
        "compoundArrayProperty": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "A": { "type": "string" },
              "B": { "type": "string" }
            }
          }
        },
        "simplePropertyRef": { "sdfRef": "#/sdfObject/ExampleObj"
        "compoundPropertyRef": { "sdfRef": "#/sdfObject/Example0"
        "simpleArrayPropertyRef": { "sdfRef": "#/sdfObject/Examp"
        "compoundArrayPropertyRef": {
          "sdfRef": "#/sdfObject/ExampleObject2/sdfProperty/co"
          "minItems": 4
        }
      }
    }
  }
}

```

Figure 33: SDF model that uses the sdfRef with sdfProperty

---

```

module exampleModel {
    // [...]
    grouping compoundArrayProperty {
        list compoundArrayProperty {
            helper:sdf-spec "sdfProperty";
            key "A";
            leaf A { type string; }
            leaf B { type string; }
        }
    }
    grouping compoundProperty {
        helper:sdf-spec "sdfProperty";
        leaf A { type string; }
        leaf B { type string; }
    }
    container ExampleObject2 {
        helper:sdf-spec "sdfObject";
        container compoundPropertyRef {
            helper:sdf-spec "sdfProperty";
            uses compoundProperty;
        }
        uses compoundProperty;
        leaf-list simpleArrayProperty { type string; }

        leaf-list simpleArrayPropertyRef {
            type leafref { path "/ExampleObject2/simpleArrayProperty"; }
        }
        leaf simpleProperty { type string; }
        leaf simplePropertyRef {
            type leafref { path "/ExampleObject2/simpleProperty"; }
        }
        container compoundArrayPropertyRef {
            uses compoundArrayProperty {

```

```

        refine compoundArrayProperty { min-elements 4; }
    }
}
uses compoundArrayProperty;
}
}

```

Figure 34: YANG conversion of the SDF model from the last figure

The common quality `sdfRequired` contains a list of SDF declarations that are mandatory to be present in an instance of the SDF model. The issue with the conversion of this quality is that in YANG not all nodes can be marked with the mandatory statement while in SDF all declarations (that means `sdfProperties`, `sdfActions` and `sdfEvents` that occur in an `sdfObject`) can be mentioned in the `sdfRequired` list. In

YANG only leaf and choice nodes (and `anyxml` and `anydata` nodes but these are not used for conversion) can be directly labeled as mandatory. List and leaf-list nodes can indirectly be made mandatory through the `min-elements` statement. Furthermore, container nodes without a presence statement that have at least one mandatory node as a child are also mandatory themselves. Not all SDF declarations are always converted to YANG leaf, choice, list or leaf-list nodes, however. Thus, if the YANG node equivalent to the mandatory SDF declaration is a non-presence container, its sub-tree is traversed until a leaf or choice node is found. This leaf or choice node is labeled as mandatory, now making its parent container mandatory as well because one of its child nodes is mandatory. An example for such a conversion is illustrated in the `compoundProperty` definition in Figure 35 and Figure 36. Consequently, if the parent node of the now mandatory container would be a container it would now be mandatory as well. Alternatively, if a list or leaf-list node is found first, the `min-elements` statement of the node is set to 1 if it is not already set to a value greater than zero, which also makes a node mandatory. This is illustrated in the `simpleArrayProperty` and `compoundArrayProperty` definitions in Figure 35 and Figure 36. To prevent loss of information and to facilitate round trips, the declaration originally listed in the `sdfRequired` quality is preserved in the `sdf-spec` extension as described in [Section 5.2](#).

```
"sdfObject": {
  "ExampleObject": {
    "sdfRequired": [
      "#/sdfObject/ExampleObject/sdfProperty/simpleProperty",
      "#/sdfObject/ExampleObject/sdfEvent/compoundProperty",
      "#/sdfObject/ExampleObject/sdfEvent/simpleArrayProperty",
      "#/sdfObject/ExampleObject/sdfEvent/compoundArrayProperty"
    ],
    "sdfProperty": {
      "simpleProperty": { "type": "string" },
      "compoundProperty": {
        "type": "object",
        "properties": {
          "A": { "type": "string" },
          "B": { "type": "string" }
        }
      }
    },
    "simpleArrayProperty": {
      "type": "array",
```

```

        "items": { "type": "string" }
    },
    "compoundArrayProperty": {
        "type": "array",
        "items": {
            "type": "object",
            "properties": {
                "A": { "type": "string" },
                "B": { "type": "string" }
            }
        }
    }
}

```

Figure 35: SDF model that contains the sdfRequired quality

```

container ExampleObject {
    helper:sdf-spec "sdfObject";

    list compoundArrayProperty {
        helper:sdf-spec "sdfProperty";
        helper:sdf-spec "sdfRequired";
        config false;
        min-elements 1;
        leaf A { type string; }
        leaf B { type string; }
    }
}

```



```

    container compoundProperty {
      helper:sdf-spec "sdfProperty";
      helper:sdf-spec "sdfRequired";
      leaf A {
        type string;
        mandatory true;
      }
      leaf B { type string; }
    }

    leaf-list simpleArrayProperty {
      helper:sdf-spec "sdfProperty";
      helper:sdf-spec "sdfRequired";
      type string;
      min-elements 1;
    }

    leaf simpleProperty {
      helper:sdf-spec "sdfRequired";
      type string;
      mandatory true;
    }
  }
}

```

Figure 36: YANG conversion of the last figure

#### 4.6. Data Qualities

- \* SDF: [Section 4.7](#) (data qualities) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG:
  - [Section 7.4.1](#) (type) of [[RFC7950](#)]

The set of qualities labeled as `_data qualities_` contains qualities inspired by the [json-schema.org](#) specifications that SDF adopted as well as qualities specifically defined for SDF. In the first group there is a total of 18 qualities out of which some are interdependent.

The quality that a lot of the other qualities presence or absence depends on is the type quality. The type can be one of number, string, boolean, integer, array or object. This quality is directly converted to the YANG type statement for all simple type. The type number becomes decimal64, integer becomes int64. The types string and boolean have built-in type equivalents in YANG. The types array and object cannot be converted to a YANG built-in type directly. Instead SDF definitions with these types are converted as described in [Section 4.8](#) and [Section 4.7](#), that is type array is mapped to leaflist or list nodes and type object is mapped to container nodes.

If a constant value is defined in an SDF definition, the data quality const is used to hold it. If the value of the type quality is number or integer the const quality is mapped to the range sub-statement of the type statement of YANG, which can also contain a single value. An example of such a conversion is illustrated in displayWidth in Figure 37 and Figure 38. For constant string values the YANG pattern statement containing the constant string is used, as shown in the displayText definition in Figure 37 and Figure 38. Unfortunately, constant values of types boolean and array can only be preserved in YANG through the sdf-spec extension.

```
"sdfObject": {
  "Display": {
    "sdfProperty": {
      "displayText": {
        "type": "string",
        "const": "Hello World!"
      },
      "displayWidth": {
        "type": "integer",
        "const": 300
      }
    }
  }
}
```

Figure 37: SdfObject that contains the const quality

```

container Display {
    helper:sdf-spec "sdfObject";
    leaf displayText {
        type string { pattern "Hello World!"; }
    }
    leaf displayWidth {
        type int64 { range "300"; }
    }
}

```

Figure 38: YANG conversion of the sdfObject from the last figure

The default data quality in SDF holds the default value for its definition. Since YANG leaf and leaf-list nodes have a default sub-statement, SDF default values of simple types or of type array with items of simple types can easily be represented.

The data qualities minimum, maximum, exclusiveMinimum and exclusiveMaximum which are only valid for the types number and integer are converted using the YANG range statement again. For exclusive boundaries the range is reduced accordingly in YANG. This is only possible for integer types or if the multipleOf quality specifies the size by which the number limit has to be reduced. Alternatives in the YANG range have to be disjoint, however. This poses a problem when the range statement is already used to map a constant value. Thus, if both minimum or maximum and constant values are defined, this is represented through the YANG union built-in type, instead. As illustrated in Figure 39 and Figure 40, in the YANG conversion of the definition the union contains the same type twice, but with different ranges.

```

"sdfProperty": {
    "displayWidth": {
        "type": "integer",
        "const": 300,
        "minimum": 100,
        "maximum": 1000
    }
}

```

Figure 39: SdfProperty that uses the minimum and maximum qualities in conjunction with the const quality

```
leaf displayWidth {  
    type union {  
        type int64 { range "300"; }  
        type int64 { range "100..1000"; }  
    }  
}
```

Figure 40: YANG conversion of the sdfProperty from the last figure

The multipleOf data quality is one that can only be used in conjunction with the number type in SDF and states the resolution of the decimal value, that is, of which decimal number the value is a multiple of. This quality is converted to the fraction-digits sub-statement to the type statement in YANG by counting the digits after the decimal separator of the value of the multipleOf quality. Since the fraction-digits statement is mandatory in YANG, it is set to 6 by default. This is done because six is also the default decimal resolution of the `std::to_string()` method of the C++ standard library. This method is used for transferring data from the C++ objects that represent SDF definitions into JSON.

The minLength and maxLength data qualities of SDF are used to hold the minimal and maximal length of strings. This concept can be transferred to YANG by using the length sub-statement of the type statement that specifies a length range.

The SDF pattern data quality holds regular expressions for string typed definitions. This can be converted directly to the pattern sub-statement of the type statement in YANG. As already mentioned in [Section 3.16](#) regular expressions cannot be converted directly between SDF and YANG in theory, due to the differing languages used for regular expressions. Because of the time limitations of this thesis no further measures are taken to insure the conformance of converted regular expressions.

The string type in SDF can be supplemented by the format quality. This quality can specify one of the formats found on [json-schema.org](https://json-schema.org). This could be translated to YANG referencing typedefs from the widely used `ietf-yang-types` module. To not rely on external modules, the format is only preserved through an addition of the sdf-spec extension to the YANG equivalent of the SDF definition the format quality is contained in.

The length of an array in SDF can be restricted by the `minItems` and `maxItems` qualities. In YANG, both list and leaf-list nodes use the sub-statements `min-elements` and `max-elements` to express the same concept. They are therefore used to convert the SDF array length qualities.

Another restriction for SDF arrays is the `uniqueItems` quality that can be set to either `true` or `false`. If it is set to `true` all items of an array are required to be different. For this purpose, YANG specifies the key and the `unique` sub-statements for list nodes. The combined values of the mentioned nodes have to be unique. These statements can only be applied to leaf nodes in the sub-tree. This does not pose a problem, however, because the uniqueness of a definition can only be measured by the uniqueness of its scalar values anyway. Thus, if an SDF array is converted to a YANG list node and the `uniqueItems` SDF quality is set to `true`, the key statement of the list states the first descendant leaf node of the list as the key, as illustrated in the `compoundArrayProperty` definition in Figure 41 and Figure 42. The key statement is chosen over the `unique` statement because it must be present in all writable lists anyway. It is not possible to explicitly represent the `uniqueItems` quality in leaf-list nodes. However, the values of leaf-list nodes that represent configuration data, and are therefore writable, must be unique. The `writable` quality is set to `true` by default. Thus, to represent an SDF array with unique items, in YANG the `config` statement is set to `true` whenever the `writable` quality in SDF is not set to `false`. An example of such a conversion can be found in the `simpleArrayProperty` definition in Figure 41 and Figure 42. Non-writable arrays with unique items cannot be represented as YANG leaf-lists.

```
"sdfObject": {
  "ExampleObject": {
    "sdfProperty": {
      "simpleArrayProperty": {
        "type": "array",
        "uniqueItems": true,
        "items": { "type": "string" }
      },
      "compoundArrayProperty": {
        "type": "array",
```

```

        "uniqueItems": true,
        "items": {
            "type": "object",
            "properties": {
                "A": { "type": "string" },
                "B": { "type": "string" }
            }
        }
    }
}

```

Figure 41: SdfObject containing the uniqueItems quality

```

container ExampleObject {
    helper:sdf-spec "sdfObject";

    list compoundArrayProperty {
        helper:sdf-spec "sdfProperty";
        key "A";
        leaf A { type string; }
        leaf B { type string; }
    }

    leaf-list simpleArrayProperty {
        type string;
        config true;
    }
}

```

Figure 42: YANG conversion of the sdfObject from the last figure

The items data quality of SDF is a quality that specifies item constraints for the items of an array-typed SDF definition using a subset of the common and data qualities. SDF definitions with the type array are converted to list or leaf-list nodes. These node types in themselves indicate that a node represents an array. Thus, the qualities defined in the item constraints of an array are converted to the sub-statements of the equivalent list or leaf-list node as described in this section. Figure 41 and Figure 42 contain

an illustration of this mapping.

Another SDF data quality is the properties quality. Properties defined through this quality are different from `sdfProperties`. The properties quality is used in conjunction with the object type and contains a set of named definitions made up of data qualities themselves. SDF definitions of type object are converted to container or grouping nodes. Thus, the named entries in the properties quality are each transformed to the child nodes of the container or grouping in question. This is illustrated in [Section 4.8](#) in the compoundProperty definition of Figure 45 and Figure 46. To label the properties as mandatory the required quality is used. Since it is resembling the `sdfRequired` quality, it is translated in the same way. The SDF type object was first introduced in SDF version 1.1 and made conversion of SDF models to YANG significantly more complicated. On the other hand, it is crucial to represent the tree structure of YANG.

The second group of qualities that is part of the data qualities includes 11 qualities that are defined specifically for SDF.

The unit quality can be set to any of the SenML unit names to represent the unit of an SDF definition. There is also a similar statement that can be defined as a sub-statement to `typedef` definitions, leaf nodes and leaf-list nodes. The units statement in YANG can contain any string and thus is simply set to the SenML unit name from the SDF definition.

An important data quality is the `sdfChoice` quality. It represents the choice between several sets of named definitions made up of data qualities themselves. YANG provides a very similar statement, the choice statement. An `sdfChoice` is turned into a YANG choice node. Each of the alternatives of the `sdfChoice` is converted like an `sdfProperty` (see [Section 4.8](#)) and added to the choice node inside its own case node. `SdfChoice` definitions that give the choice between the type quality could also be mapped to the YANG type union. This is omitted for reasons of simplicity. An example conversion of the `sdfChoice` quality can be found in Figure 43 and Figure 44.

```
"sdfObject": {
  "ExampleObject": {
    "sdfProperty": {
```

```

        "choiceProperty": {
            "sdfChoice": {
                "foo": { "type": "string" },
                "bar": { "type": "boolean" },
                "baz": { "type": "integer" }
            }
        }
    }
}

```

Figure 43: SdfObject with an sdfChoice quality

```

container ExampleObject {
    helper:sdf-spec "sdfObject";
    choice choiceProperty {
        case bar {
            leaf bar { type boolean; }
        }
        case baz {
            leaf baz { type int64; }
        }
        case foo {
            leaf foo { type string; }
        }
    }
}

```

Figure 44: YANG conversion of the sdfObject from the last figure

SDF also offers the possibility to define the choice between string values by means of the enum data quality. It consists of an array of strings. This concept also exists in YANG with the enumeration type and the corresponding enum sub-statement to the type statement. For an SDF definition that contains the enum quality the YANG type of its equivalent is set to enumeration. Each of the strings in the array of the enum SDF quality is converted to an enum entry in the type statement in YANG. The enum entries are also assigned an associated value.

The scaleMinimum and scaleMaximum qualities represent limits in units as specified by the unit quality. They are not mapped to YANG



because they will not be included in future versions of SDF. They are to be replaced in the future, therefore a mapping will have to be developed for their replacement.

The `contentType` quality of SDF can provide an additional IANA content type. This information is preserved with the help of `sdf-spec` extension in the YANG equivalent of the SDF definition.

Another way to complement the type quality is the `sdfType` quality that can either be set to `byte-string` or `unix-time`. A byte string is converted to the YANG type `binary`. There is no built-in YANG type corresponding to `unix-time` it is thus converted through the YANG `units` statement. The unit of the YANG conversion mentions `unix-time` as an argument.

SDF defines the `readable` and `writable` qualities to flag whether read or write operations are allowed on definitions. Read operations are always allowed in YANG modules so a `readable` quality that is set to `false` cannot be represented in YANG. The `config` YANG statement can be used to represent the value of the `writable` quality, however. If an SDF definition is explicitly marked as `writable` `config` is set to `true`. Otherwise, it is set to `false`.

The `observable` and `nullable` qualities in SDF cannot be represented in YANG but are preserved by adding an `sdf-spec` extension to the YANG equivalent of their containing SDF definition.

#### [4.7.](#) SdfData Quality

- \* SDF: Sections [2.2.5](#) and [5.5](#) (`sdfData`) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG:
  - [Section 7.13](#) (uses) of [[RFC7950](#)]

- [Section 7.12](#) (grouping) of [[RFC7950](#)]

Elements of the `sdfData` class are meant to hold data type definitions to be shared by `sdfProperty`, `sdfAction` and `sdfEvent` definitions. `SdfData` definitions can make use of the data qualities and the common qualities described in [Section 4.6](#) and [Section 4.5](#) respectively. Because an `sdfData` definition embodies a data type definition the

YANG statements typedef and grouping have to be used for conversion. Which of the two is used depends on the value of the type quality of the sdfData definition. If the type is one of the simple data types, that is integer, number, boolean or string, the sdfData definition is converted to a YANG typedef. If the type is object the sdfData definition is mapped to a grouping node with each of the entries of the properties quality of the compound-type being mapped to a child node of the grouping. When mapping sdfData definitions with type array to YANG, the type mentioned in the type quality of the items quality is essential as well. If an array has items of any of the simple types the resulting YANG element is a grouping node containing a single leaf-list node. Otherwise, if the array items are compound-types the sdfData definition is converted into a grouping node containing a single list node. The child nodes of the list node are equivalent to the entries of the properties quality that is contained in the item quality.

One issue with converting sdfData definitions of type array is the added grouping node that is necessary to hold the equivalent leaf-list or list node. If the grouping is used in the schema tree the added level will cause model instances of the original and converted model to be in-equivalent. If the sdfData definition is referenced in the SDF model via the sdfRef common quality this is represented in YANG with the uses statement pointing to the grouping equivalent to the sdfData definition. The sdfRef quality can occur at most once in each definition while there can be multiple uses statements in a single container, list or grouping. Thus, instead of representing definitions containing an sdfRef by a parent node containing a uses node, the aforementioned issue with array-typed sdfData definitions could be solved by replacing the parent node with the uses node itself, effectively removing the excess level. This, however, gives rise to other issues because the name of the superordinate definition of the sdfRef is lost this way. An example for this issue is illustrated in Figure 53 and Figure 54. If the sdfData definition is converted to a typedef no such issues arise. The typedef in question is inserted as an argument to the YANG type quality wherever the original sdfData definition was referenced by an sdfRef.

Another issue is a different view on global accessibility of data type definitions in YANG and SDF. In SDF, all definitions are globally available as long as a default namespace is defined in the

SDF model. In YANG on the other hand, only data type definitions, that is groupings and typedefs, that occur on the top-level of a YANG module are globally accessible. Thus, to represent the global accessibility of all data type definitions in SDF, all converted sdfData definition equivalents in YANG are added to the top-level of the created module.

Since these issues are also discussed in [Section 4.5](#), examples conversion can be found there in Figure 31 and Figure 32.

#### [4.8](#). SdfProperty Quality

- \* SDF: Sections [2.2.2](#) and [5.2](#) (sdfProperty) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG:
  - [Section 7.6](#) (leaf) of [[RFC7950](#)]
  - [Section 7.7](#) (leaf-list) of [[RFC7950](#)]
  - [Section 7.8](#) (list) of [[RFC7950](#)]

SdfProperty definitions represent elements of state as suggested by their name. SdfProperty definitions can make use of the data qualities and the common qualities described in [Section 4.6](#) and [Section 4.5](#). The mapping of an sdfProperty definition to YANG depends on the value of the type quality. SdfProperties with simple types are mapped to leaf nodes in YANG, as illustrated in the simpleProperty definition in Figure 45 and Figure 46. If the type is complex, that is type object, conversion results in a container node with each of the entries in the properties quality being mapped to a child node of the container. An example of such a conversion is the compoundProperty definition in Figure 45 and Figure 46. If the sdfProperty is of type array the deciding factor is the type quality inside the items quality. If an array has items of a simple type, it is converted to a leaf-list node. This is demonstrated by the simpleArrayProperty definition in Figure 45 and Figure 46. Otherwise, if the items are of compound-type the sdfProperty becomes a list node in YANG. The child nodes of the list node are equivalent to the entries of the properties quality in the compound-type, as illustrated in Figure 45 and Figure 46 through the compoundArrayProperty definition. List nodes that represent configuration data, that means data that is writable, must specify at least one of its descendant leaf nodes as a key identifier. In SDF definitions that use the data qualities, such as sdfProperties, the writable quality is set to true by default. Therefore, the key statement of the list node is set to the first descendant leaf node of the list by default by the converter to comply with this rule. For round trips, this work-around is noted through the sdf-spec extension.

```
"sdfObject": {
  "ExampleObject": {
    "sdfProperty": {
      "simpleProperty": { "type": "string" },
      "compoundProperty": {
        "type": "object",
        "properties": {
          "A": { "type": "string" },
          "B": { "type": "string" }
        }
      },
      "simpleArrayProperty": {
        "type": "array",
        "items": { "type": "string" }
      },
      "compoundArrayProperty": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "A": { "type": "string" },
            "B": { "type": "string" }
          }
        }
      }
    }
  }
}
```

Figure 45: SdfObject with an sdfProperty definition

```
container ExampleObject {
  helper:sdf-spec "sdfObject";
  list compoundArrayProperty {
    key "A";
    leaf A { type string; }
  }
}
```

```

        leaf B { type string; }
    }
    container compoundProperty {
        leaf A { type string; }
        leaf B { type string; }
    }
    leaf-list simpleArrayProperty { type string; }
    leaf simpleProperty { type string; }
}

```

Figure 46: YANG conversion of the sdfObject from the last figure

#### [4.9.](#) SdfAction Quality

- \* SDF: Sections [2.2.3](#) and [5.3](#) (sdfAction) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG:
  - [Section 7.14](#) (rpc) of [[RFC7950](#)]
  - [Section 7.15](#) (action) of [[RFC7950](#)]

To represent operations that can be invoked in a model the sdfAction class is used. Since operations can have input and output data the sdfAction class is equipped with the sdfInputData and sdfOutputData qualities that can both make use of the data qualities and the common qualities described in [Section 4.6](#) and [Section 4.5](#). An sdfAction can also define its own set of data types in the form of sdfData definitions. Whether an sdfAction is converted to an rpc node (which can only occur at the top-level of a module) or an action node (which is always tied to a container node) depends on its location inside the SDF model. SdfActions that are not part of an sdfObject but can be found independently at the top of an SDF model are converted to rpc nodes. All other sdfActions occurring inside an sdfObject become action nodes inside the YANG container equivalent to the sdfObject, as illustrated in Figure 47 and Figure 48. The sdfInputData and sdfOutputData of an sdfAction are converted like sdfProperties (see [Section 4.8](#)) and added as the input and output node of the YANG RPC/action respectively.

```
"sdfObject": {
  "ExampleObject": {
    "sdfAction": {
      "printString": {
        "sdfInputData": {
          "type": "object",
          "properties": {
            "content": { "type": "string" },
            "colour": { "type": "string" }
          }
        },
        "sdfOutputData": {
          "type": "object",
          "properties": {
            "success": { "type": "boolean" }
          }
        }
      }
    }
  }
}
```

Figure 47: SdfObject definition that contains an sdfAction definition

```
container ExampleObject {
  helper:sdf-spec "sdfObject";
  action printString {
    input {
      leaf colour { type string; }
      leaf content { type string; }
    }
    output {
      leaf success { type boolean; }
    }
  }
}
```

Figure 48: YANG conversion of the sdfObject from the last figure

#### [4.10.](#) SdfEvent Quality

- \* SDF: Sections [2.2.4](#) and [5.4](#) (sdfEvent) of [[I-D.ietf-asdf-sdf](#)]
- \* YANG: [Section 7.16](#) (notification) of [[RFC7950](#)]

The purpose of the sdfEvent class is to model signals that inform about occurrences or events in an sdfObject. To represent the emitted output data, sdfEvents can make use of the sdfOutputData

quality which in turn uses the data qualities. An sdfEvent is converted to a notification node with one child node representing the sdfOutputData definition. The sdfOutputData definition is converted like an sdfProperty (see [Section 4.8](#)). Figure 49 and Figure 50 contain the SDF and YANG representations of a warning notification which communicates the device and reason of the warning.

```
"sdfEvent": {
  "warning": {
    "sdfOutputData": {
      "type": "object",
      "properties": {
        "warningDevice": { "type": "string" },
        "warningReason": { "type": "string" }
      }
    }
  }
}
```



```

    }
  }
}

```

Figure 49: SdfEvent definition

```

notification warning {
  leaf warningDevice { type string; }
  leaf warningReason { type string; }
}

```

Figure 50: YANG conversion of the sdfEvent from the last figure

## 5. Challenges

Since conversion between SDF and YANG is not always trivial this section takes a look at the various challenges that arose in the process of finding an adequate mapping for each of the language's features to one another.

### 5.1. Differences in Expressiveness of SDF and YANG

SDF and YANG differ in their expressiveness in different areas. Compared to the other format, both are stronger in some areas and weaker in others.

Areas in which YANG is more expressive are regular expressions, operations, some of the built-in types (bits and empty) and the retrospective augmentation of existing definitions. In YANG, multiple regular expressions to be matched can be defined and they can also be labeled as invert-match expressions. Both features are difficult to express in SDF as of now. Furthermore, YANG and SDF use slightly different regular expression languages. YANG uses a regular

expression language as defined by W3C Schema while SDF adopts ECMAScript regular expressions. Operations in YANG can be defined on their own or with an affiliation to a YANG container. This affiliation is not always trivial to represent in SDF. The YANG built-in types bits and empty do not have equivalents in SDF. The semantics of those types can, however, easily be mapped to SDF. A YANG statement whose semantics cannot be fully mapped to SDF is the augment statement. The augmentation can be applied and then

converted but cannot be represented as a retrospective addition to an SDF definition or model. Another Language feature of YANG that SDF does not offer is the option to place constraints on valid data via XPath expressions and the option to make sections of the model conditional with the feature statement. YANG, furthermore, puts no constraints on the value of its units statement, whereas SDF does only allow SenML unit names in the unit quality.

SDF offers more possibilities to define default and constant values, the latter especially in conjunction with minimum and maximum values. YANG uses a single statement, the range statement, for constant, minimum and maximum values. Although there can be multiple values or ranges in one range statement that are interpreted as alternatives they all need to be disjoint. This imposes a strict limit on what can be expressed through the statement. An example for a conversion where this is a problem would be an sdfData definition with values for the minimum and maximum qualities but also a given constant value that fits inside the given minimum and maximum range, like the example in Figure 51. Such a definition could be converted to a YANG typedef with a range that states the minimum and maximum value as one range and the constant as an alternative, like the example conversion in Figure 52. This example does not validate in YANG because the range alternatives are not disjoint. This problem is solved through use of the union built-in type. Furthermore, labeling definitions as readable, observable and nullable, as possible in SDF, is foreign to YANG. SDF is also more expressive in the way it labels definitions that must obligatorily occur in model instances. Basically all definitions can be labeled as such through the sdfRequired and required qualities. In YANG, only leaf, choice, anydata and anyxml nodes can be marked with the mandatory statement directly. Container, list and leaf-list nodes can only be made mandatory indirectly and there is no general mechanism in YANG for all kinds of nodes.

```
"sdfData": {
```

```

    "someValue": {
      "type": "integer",
      "minimum": 1,
      "maximum": 5,
      "const": 3
    }
  }
}

```

Figure 51: SdfData definition with the qualities minimum, maximum and const

```

typedef someValue {
  type int32;
  range "1 .. 5 | 3" // invalid in YANG
}

```

Figure 52: YANG conversion of the SDF definition in the last figure

## 5.2. Round Trips

One of the bigger issues in building the converter is the facilitation of round trips, i.e. converting a model from one format to the other and in a next step back to the original. This issue is tightly linked to the differences in expressiveness between the two formats which makes mapping between them non-injective and thus non-traceable without additional measures.

To be able to track the origins of an SDF element after conversion from YANG, currently, a so-called `\textit{conversion note}` is added to the description of the element. The note specifies a statement and optionally an argument to the statement. An example for a note stating that the original argument to the type statement was bits is: `!Conversion note: type bits!`. This approach is not able to preserve all information from the YANG module without exceptions since sub-statements cannot be specified. It is, however, sufficient in the majority of cases.

This issue was also discussed in one of the meetings of the ASDf working group. The possibility to introduce a new mechanism for round trips was suggested. Instead of overloading the SDF file with information that adds no functionality, the possibility to preserve information from the original model in a separate mapping file for each model was proposed. Mapping files for SDF models could contain selectors that assign additional information to the selected SDF element or element group. No decision has been made yet on the definite structure of such mapping files. Therefore, some requirements from the perspective of the SDF/YANG converter are

listed here. Generally speaking, the information attached to an SDF element should have at least the same information content in the mapping file as in the previously mentioned conversion note, that is a statement and optionally an argument. To also cater to statements with further sub-statements, nesting should be possible, that is defining further statements as arguments should be possible. It is also necessary to be able to specify multiple statements to attach to a selected SDF elements. Another solution to round trips with mapping files would be to reference the associated YANG element of the selected SDF element. This way, all information would be preserved. Round trips would be easy because the original YANG definition would stay attached to the converted SDF definition. Opposing to that, if the SDF conversion of the YANG model is used to be converted further into other languages, the supplementary information of the original YANG element would still have to be extracted. This defeats the purpose of SDF to reduce the number of necessary mappings between languages. Thus, to attach statements with arguments to SDF definitions in mapping files is the better solution, in our opinion.

To preserve the original SDF language element after conversion to YANG a new `sdf-spec` extension is defined in YANG. The extension states the original SDF quality and optionally an argument, similarly to the conversion note used to preserve information from YANG.

The eventuality that round trips occur in model conversion makes building the converter significantly more complex because all features of the target format have to be accounted for. Features of the target format that would otherwise not be used for conversion must now be considered in the case of a round trip.

### [5.3.](#) Type References

Both SDF and YANG offer the possibility to reference predefined types. SDF uses only a single quality for this purpose (`sdfRef`) whereas YANG has several statements that are all used in different referencing contexts (`leafref`, `identityref`, `type`, `uses`). The way the `uses` statement and the `sdfRef` quality are converted regularly leads to additional containers in YANG or additional properties (when using the `compound-type`) in SDF that make instances of the same model in SDF and YANG in-equivalent and complicate round trips. If no additional elements are inserted, information, for example the name of an element, is lost.

Both the `uses` statement and the `sdfRef` quality embed the content of the referenced expression where they are located. Issues arise

because YANG provides only groupings to be embedded via the uses statement. Groupings are the non-declaration-equivalent to

containers. There is no non-declaration-equivalent to YANG lists, however. This means that list definitions in YANG need to be packaged in a grouping. If such a grouping with a single list inside is transcribed from YANG to SDF there will be an extra layer that looks redundant but otherwise does no harm. For the reasons stated above, an sdfData definition of type array with items of compound-type is converted to a list node inside a grouping in YANG. Problems arise when said sdfData definition is embedded via sdfRef because this cannot be converted directly to YANG. Such a scenario is illustrated in Figure 53 and Figure 54. The sdfData definition menu is converted to the YANG list menu inside a grouping menu. Referencing the menu via sdfRef in the sdfProperty definitions menu\_english and menu\_german is equivalent to copying the qualities of the menu there. In the YANG conversion the containers menu\_english and menu\_german both use the grouping menu. This means the menu list from said grouping is copied into the containers. The containers are necessary to preserve the names menu\_english and menu\_german and also because there cannot be two sibling uses nodes with the same target grouping (because no two sibling nodes must have the same name).

Another issue with the mapping of type references is the accessibility of elements. Only typedefs and groupings that appear on the top-level of the tree can be reused globally. If these nodes appear within a sub-tree they are only available in the scope of the sub-tree. Since there is no such restriction in SDF, mapping sdfData definitions directly would cause accessibility problems in the resulting YANG module. Thus, mapped sdfData definitions have to be moved to the top-level. In YANG it is furthermore assumed that every type of node in the tree is addressable, while SDF focuses on sdfProperties, sdfActions and sdfEvents as addressable affordances.

```
; [...]
"sdfData": {
  "dish": {
    "type": "object",
    "properties": {
      "name": { "type": "string" },
      "price": { "type": "number" }
    }
  },
  "menu": {
    "type": "array",
    "items": { "sdfRef": "#/sdfData/dish" }
  }
},
"sdfObject": {
  "restaurant" : {
    "sdfProperty": {
      "menu_english": { "sdfRef": "#/sdfData/menu" },
      "menu_german": { "sdfRef": "#/sdfData/menu" },
      "dish_of_the_day": { "sdfRef": "#/sdfData/dish" }
    }
  }
}
}
```

Figure 53: SDF model with type definitions of types object and array

```
module restaurant {
  // [...]
  grouping dish {
    leaf name { type string; }
    leaf price {
```

```

        type decimal64 { fraction-digits 6; }
    }
}
grouping menu {
    list menu {
        key "name";
        uses dish;
    }
}
container restaurant {
    container dish_of_the_day { uses dish; }
    container menu_english { uses menu; }
    container menu_german { uses menu; }
}
}

```

Figure 54: YANG conversion of the SDF model in the last figure

## [6.](#) Implementation Considerations

An implementation of an initial converter between SDF and YANG can be found at [[SDF-YANG-CONVERTER](#)]; the source code can be found at [[SDF-YANG-CONVERTER-IMPL](#)].

## [7.](#) IANA Considerations

This document makes no requests of IANA.

## [8.](#) Security considerations

The security considerations of [[RFC7950](#)] and [[I-D.ietf-asdf-sdf](#)] apply.

## [9.](#) References

### [9.1.](#) Normative References

[I-D.ietf-asdf-sdf]  
 Koster, M. and C. Bormann, "Semantic Definition Format (SDF) for Data and Interactions of Things", Work in Progress, Internet-Draft, [draft-ietf-asdf-sdf-08](#), 25

October 2021, <<https://www.ietf.org/archive/id/draft-ietf-asdf-sdf-08.txt>>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", [RFC 7950](#), DOI 10.17487/RFC7950, August 2016, <<https://www.rfc-editor.org/info/rfc7950>>.

## 9.2. Informative References

[LIBYANG] Vasko, M., Sedláč, D., and more contributors, "libyang", <<https://github.com/CESNET/libyang>>.

[SDF-YANG-CONVERTER]  
Kiesewalter, J., "SDF YANG converter playground", n.d., <[sdf-yang-converter.org](https://sdf-yang-converter.org)>.

[SDF-YANG-CONVERTER-IMPL]  
Kiesewalter, J., "SDF YANG converter", n.d., <<https://github.com/jkiesewalter/sdf-yang-converter>>.

## Acknowledgements

TBD.

## Authors' Addresses

Jana Kiesewalter  
Universität Bremen

Email: [jankie@uni-bremen.de](mailto:jankie@uni-bremen.de)

Carsten Bormann (editor)  
Universität Bremen TZI  
Postfach 330440  
D-28359 Bremen  
Germany



Phone: +49-421-218-63921  
Email: [cabo@tzi.org](mailto:cabo@tzi.org)