XSD2SHACL: Capturing RDF Constraints from XML Schema

Xuemin Duan
xuemin.duan@kuleuven.be
Department of Computer Science
KU Leuven – Leuven.AI
Flanders Make@KULeuven
Leuven, Belgium

David Chaves-Fraga
david.chaves@usc.es
Grupo de Sistemas Intelixentes
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
Department of Computer Science
KU Leuven – Leuven.AI
Flanders Make@KULeuven
Leuven, Belgium

Anastasia Dimou
anastasia.dimou@kuleuven.be
Department of Computer Science
KU Leuven – Leuven.AI
Flanders Make@KULeuven
Leuven, Belgium

ABSTRACT

SHACL shapes describe the constraints of RDF subgraphs which are constructed from heterogeneous data, such as RDBs, JSONs, XMLs, etc. These heterogeneous data often already have constraints defined in their schemas, e.g., JSON Schema for JSON or XSD for XML, but this information is ignored when the RDF graph is constructed, as there are currently not many works that translate such schemas into SHACL. In this paper, we focus on the incorporation of XSD constraints for XML data sources in SHACL shapes. We define a translation from XSD to SHACL, and provide a corresponding system. We compare our solution with XMLSchema2ShEx which translates XSD constraints to ShEx and validate our solution against two use cases. Our solution provides the desired SHACL shapes in a reasonable time. This allows us to automatically derive SHACL shapes for some original raw data without any manual effort.

CCS CONCEPTS

• **Information systems** → **Semantic web description languages**; • **Computing methodologies** → *Knowledge representation and reasoning*; • **Software and its engineering** → **Constraints**.

KEYWORDS

SHACL, XSD, XML Schema, RDF shapes, Validation

ACM Reference Format:
Xuemin Duan, David Chaves-Fraga, and Anastasia Dimou. 2023. XSD2SHACL: Capturing RDF Constraints from XML Schema. In *Knowledge Capture Conference 2023 (K-CAP '23), December 05–07, 2023, Pensacola, FL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3587259.3627565

1 INTRODUCTION

Shape constraint languages, such as Shapes Constraint Language (SHACL) [13] and the Shapes Expression language (ShEx) [14], were proposed to validate RDF graphs against a set of constraints. Different methods have been considered so far to define shapes for an RDF graph. Such shapes may be manually, e.g., for RINF or TED (see Section 5.2), or automatically generated. In the latter case, the SHACL shapes may be derived from *RDF graphs* [7–9, 15, 16, 18], *ontologies* [2, 11], or *mapping rules* that define how the RDF graph should be constructed from some raw data [4].

However, while these RDF graphs are often derived from raw data [21], e.g., CSV, XML, or JSON, and these raw data may already have constraints defined in their schema, e.g., SQL [12] for relational databases, XSD [6] for XML, or JSON Schema [23] for JSON, these constraints are usually ignored when the RDF graph is constructed. Thus far, most of the solutions for automated creation of constraints for RDF graphs do not leverage the schema and constraints of the raw data from which the RDF graphs are derived. Garcia and Labra-Gayo [10] proposed a method to translate the constraints of the XSD to ShEx. Nonetheless, even though it paves the way for a novel research area, it comes with various limitations and proves inadequate in more intricate scenarios. Thapa and Giese [19, 20] proposed a source-to-target rewriting of SQL constraints to SHACL constraints. However, their solution complements the direct mapping recommendation [1] of relational databases to RDF which is not as frequently used to construct RDF graphs from raw data.

In this work, we investigate a method to preserve the already existing constraints defined in XSD of the raw XML data from which an RDF graph is derived. We describe our translation algorithm and provide an open-source implementation of our translation mechanism. Our objective is to streamline the construction of RDF graphs and the corresponding SHACL shapes. We compare XSD2SHACL with XMLSchema2ShEx [10], and apply our solution to two real-world use cases to validate that our solution is applicable in more intrinsic real-world use cases. With our work, we aim to reduce the effort to create SHACL shapes and minimize the time needed for defining constraints once the RDF graph is already constructed or even before the RDF graph is constructed.

Contributions. Our contributions can be summarized as follows: (i) a translation from XSD to SHACL; (ii) XSD2SHACL[1], an open source system that implements the translation; (iii) a comparison of XSD2SHACL and XMLSchema2ShEx; and (iv) its validation in two real-world use cases: ERA-RINF and TED.

The paper is organized as follows: In Section 2 we describe preliminaries and related works, in Section 3 we describe in detail

K-CAP '23, December 05–07, 2023, Pensacola, FL, USA

ACM ISBN 979-8-4007-0141-2/23/12…$15.00
https://doi.org/10.1145/3587259.3627565

[1] https://github.com/dtai-kg/XSD2SHACL

the translation of the XSD into SHACL, and in Section 4 we introduce our implementation: XSD2SHACL. In Section 5, we compare XSD2SHACL with XMLSchema2ShEx, and validate our solution against two real-world use cases. In Section 6, we conclude our paper and outline possible future directions.

2 PRELIMINARIES AND RELATED WORK

In this section, we introduce background knowledge on XSD and SHACL required to follow this paper (Section 2.1) and explain the related work on the generation of constraints for RDF (Section 2.2).

2.1 Preliminaries

In this subsection, we summarize the concepts of XSD and SHACL.

XSD. The XML Schema Definition Language (XSD) [6] is a W3C-recommended language for describing the XML document using a set of declarations and definitions [22]. Declarations (e.g., Listing 2, line 1) describe the element and attribute (e.g., Listing 1, line 1) that may appear in an instance document (i.e. XML). Instance documents model elements using a parent-child relationship. For example, in Listing 1, the `student` element could contain a `grade` child element. Hence, in Listing 2, its XSD defines it as an element declaration with `student` as `name` associated with a complex type definition that contains the `grade` child element declaration.

Type definitions can be global or local. A *globally defined type* (e.g., Listing 2, lines 3-16) is named and can be referred to by any other element and attribute declarations, whereas a *locally defined type* (e.g., Listing 2, lines 7-12) is anonymous and scoped to the definition or declaration that contains it. Type definitions can be *simple* or *complex*. Simple-type elements only contain character data, (e.g., Listing 1, line 2), while complex-type elements contain character data, attributes, or child elements. (e.g., Listing 1, line 1). Simple type definitions can be *built-in* (e.g., `xs:string`) or *user-defined* (e.g., Listing 2, lines 7-12). Complex type definitions have different types of content, where *simple content* (e.g., `xs:simpleContent`) and *mixed content* (e.g., *attribute* `mixed`="`true`") allows character data, and *element-only content* and *empty-content* do not.

Attributes (e.g., `minOccurs`) within the declarations and definitions add more constraints. *Unprefixed* `names` (e.g., `name`="`student`") of globally defined components belong to a particular namespace defined by the *attribute* `targetNamespace`. If one of the *attributes* `elementFormDefault` or `attributeFormDefault` is set to "*qualified*" or the *attribute* `form` within the declaration is set to "*qualified*", the unprefixed `names` of the locally defined element or attribute declaration will also be included in the namespace.

SHACL. Shapes Constraint Language (SHACL) [13], a W3C recommendation, describes how to validate RDF graphs against a set of constraints in shapes. Shapes can be categorized into *node shapes* that cannot contain property path and *property shapes* that are required to contain a property path. Target declarations (e.g., `sh:targetClass`) are used to produce focus nodes as the input of the validator and are optional for all shapes. The value of a property path (i.e. `sh:path`) should be the predicate in RDF graphs to reach the object value from the focus node for validation purposes. For example, in Listing 3, the node shape may not have a target declaration (lines 4-7), but the property shapes need to define a property path (see `sh:path` in lines 11, 16, 20).

SHACL has a set of built-in core constraint components to validate nodes targeted by a shape, e.g., value type, cardinality, value range, etc. SHACL embodies the reference relationships between shapes through `sh:property` and `sh:node` to indicate that the current node must conform to the referenced shape as well. For example, in Listing 3, a node shape with `sh:targetClass :student` (lines 1-2) references another node shape using `sh:node`, and the referenced node shape uses `sh:property` to reference a property shape with `sh:path :grade` (line 6), to define that `:student` instances need to conform the `:grade` property shape.

2.2 Related work

Limited works generate RDF shapes from the schema and constraints of raw data. Most of the previous works are focused on deriving SHACL shapes from *RDF graphs*, *ontologies*, and *mapping rules* that define how the RDF graph should be constructed. Deriving shapes from *RDF graphs* [7–9, 15, 16, 18] aligns closely with the target validation graphs, but the computational burden is contingent upon the data scale. In contrast, deriving shapes from *ontologies* [2, 11] is independent of the data scale, resembling with the raw data schema, but is confined to deriving constraints within the defined ontology. Deriving shapes from *mapping rules* (e.g., [4] which leverages on RML [5]) captures the implied constraints within the mapping rules, but is limited to the ones described by the mapping language, e.g., if a mapping rule defines the data type, then this data type is considered as a constraint for the RDF graph.

Garcia and Labra-Gayo [10] proposed XMLSchema2ShEx, a system that converts XSD to ShEx. They treat the element declaration as the triple predicate and object, and convert it into triple constraints where the predicate is the name of the declaration. They extend this to attribute declarations due to the absence of the attribute concept in the RDF model. Such conversions are aligned with RDF where all XML element names are used as properties in the predicate position, e.g., given `<student><id>r001</id></student>`, the constructed RDF would be `:student1 :student [:id "r001"]`. However, this is not how this XML would be intuitively modeled in RDF, as the name of the element containing sub-elements could be employed as the class in the object position, e.g., `:student1 a :student; :id "r001"`. Their system, XMLSchema2ShEx[2] covers 14 XSD components, but only generates valid ShEx shapes for 7 of them. Last, it also comes with even more limitations as it cannot handle input XSD containing unsupported components.

Thapa and Giese [19] proposed a source-to-target semantics preserving rewriting of SQL constraints on the direct mapping [1] to SHACL constraints to construct the RDF graph while preserving integrity constraints information. Thapa and Giese [20] further introduce a constraint rewriting for relational to RDF mappings, such as R2RML [3], to convert SQL constraints to SHACL constraints, that better aligns the SHACL shapes with the RDF graph.

```
1 <student nationality="BE">
2   <id>r001<id/>
3   <grade>18<grade/>
4 <student>
```

Listing 1: Example XML

[2],https://github.com/herminiogg/XMLSchema2ShEx

```
<xs:element name="student" type="StudentType"/>

<xs:complexType name="StudentType">
  <xs:all>
    <xs:element name="id" type="xs:string"/>
    <xs:element name="grade">
      <xs:simpleType>
        <xs:restriction base="xs:integer">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="20"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:all>
  <xs:attribute name="nationality" type="xs:string"/>
</xs:complexType>
```

Listing 2: Example XSD

3 TRANSLATION

In this section, we introduce our translation from XSD to SHACL. We first explain how we preserve the parent-child hierarchy of XSD into the reference relationships in SHACL (Section 3.1). This involves the generation of node and property shapes, target declarations, and property paths from element and attribute declarations, and type definitions. Then we present how we generate SHACL core constraints from XSD facets and attributes (Section 3.2). It is important to note that we retain the namespaces for the names of elements and attribute declarations that are either explicitly qualified by a prefix or implicitly by the defined `targetNamespace`.

3.1 Components Hierarchy Preservation

Both XSD and SHACL delineate the relationship among their components through hierarchical structures (parent-child or reference). To preserve the hierarchical relationships during the translation (e.g., a `student` should have a `grade`), we rely on the type of declaration to generate a shape and establish the reference relationships between translated shapes using `sh:node` and `sh:property`.

Element and attribute declarations. We translate element and attribute declarations to node or property shapes according to their type definitions, as shown in Table 1. Element declarations associated with simple type definitions together with all attribute declarations are translated into property shapes. In contrast, element declarations associated with complex type definitions are translated into node shapes. If the type content is simple or mixed, we generate one more property shape and incorporate the constraints translated from its `restriction` and facets within the shape. The name of the declaration becomes the value of a sh:targetClass within node shapes and of a sh:path within property shapes.

The element declaration associated with a local complex type definition uses `sh:property` to reference its child declarations that are translated into property shapes, and `sh:node` to reference its child declarations or definitions that are translated into node shapes. However, if that is globally defined, it uses `sh:node` to reference the node shape translated from this complex type definition. For example, in Listing 3, the `student` element declaration with a complex type definition `StudentType` is translated into a node shape with its name as the value of `sh:targetClass` (lines 1-2), the `grade` element declaration with a simple type definition is translated into a property shape with `sh:path` (lines 9-12), and the `student`'s node shape references its type's node shape using `sh:node` (line 2).

```
1  <NS/student> a sh:NodeShape ; sh:targetClass :student ;
2      sh:node <NS/StudentType> ;  sh:name "student" .
3
4  <NS/studentType> a sh:NodeShape ; sh:name "StudentType" ;
5      sh:property <PS/StudentType/id>,
6                  <PS/StudentType/grade>,
7                  <PS/StudentType/nationality> .
8
9  <PS/StudentType/grade> a sh:PropertyShape ;
10     sh:datatype xs:integer ;  sh:name "grade" ;
11     sh:minCount 1 ; sh:maxCount 1 ; sh:path :grade;
12     sh:minInclusive 0 ; sh:maxInclusive 20 .
13
14 <PS/StudentType/id> a sh:PropertyShape ;
15     sh:datatype xs:string ;  sh:name "id" ;
16     sh:minCount 1 ; sh:maxCount 1 ; sh:path :id .
17
18 <PS/StudentType/nationality> a sh:PropertyShape ;
19     sh:datatype xs:string ;  sh:name "nationality" ;
20     sh:path :nationality .
```

Listing 3: Example SHACL shapes

Type definition. We generate a node shape without a target for the global complex type definition, thereby enhancing the compactness of shapes through repeatedly referencing the generated shape by sh:node. For example, supposed that in Listing 2, there is another element declaration `graduate`, also associated with `StudentType` in addition to `student`, then we only need to reference the generated node shape of `StudentType` (Listing 3, lines 4-7) from `student`'s shape and `graduate`'s shape through `sh:node` (e.g., line 2). The generated node shape from the definition references the shapes translated from the child declarations (e.g., `id`, `grade`, `nationality`) within the definition using `sh:node` if the child's shape is a node shape or `sh:property` if that is a property shape.

We directly translate the child components of local complex type definitions and simple type definitions and incorporate the generated shapes or constraints into its associated declaration's shape. For example, supposed that the `StudentType` is locally defined and associated with the `student` declaration, then the referenced property shapes (Listing 3, lines 5-7) will be directly referenced by the `student`'s node shape instead of generating a new shape for `StudentType`. For references (e.g., `<xs:element ref="student">`) within a definition, we directly reference the corresponding shape that is translated from the referenced declaration (e.g., `<xs:element name="student">`) from the definition's shape. Last, we translate globally declared `attributeGroup` and `group`, as the global complex type definition as shown in Table 1.

Statements in the type definition. We translate various statements within type definitions following Table 1, where simple type can define `union` and `list`, and complex type definition can define `all`, `sequence`, `choice`, `group`, and `attributeGroup`. The `union` (e.g., `<xs:union memberTypes="`*Type1 Type2*`"`) introduces a `sh:or`

Table 1: Translation from XSD components to SHACL shapes

XML Schema	SHACL
Complex type *element-only content & empty content* <xs:element name="*N*"> <xs:element name="*N*" type="*C*"/>	a sh:NodeShape sh:targetClass :*N*
Complex type *simple content & mixed content* <xs:element name="*N*"> <xs:element name="*N*" type="*C*"/>	a sh:NodeShape sh:targetClass :*N* a sh:PropertyShape sh:path :*N*
Simple type: user-defined <xs:element name="*N*"> <xs:attribute name="*N*"> <xs:element name="*N*" type="*C*"/> <xs:attribute name="*N*" type="*C*"/>	a sh:PropertyShape sh:path :*N*
Simple type: built-in <xs:element name="*N*" type="*C*"/> <xs:attribute name="*N*" type="*C*"/>	a sh:PropertyShape sh:path :*N* sh:datatype *C*
<xs:complexType name="*N*"> <xs:group name="*N*"> <xs:attributeGroup name="*N*">	a sh:NodeShape
xs:sequence	sh:order
xs:choice	sh:xone
xs:union	sh:or
xs:annotation	sh:description

within the corresponding property shape translated from the declaration that defines it followed by an RDF list that contains several sets of constraints translated from the unioned types. We do not translate `list` since SHACL does not support list validation. We do not translate `all` as it defines a set of unordered elements and shapes are unordered by default. The `sequence` introduces the `sh:order` (e.g., `sh:order 1`) within the property shapes translated from its child element declarations to indicate the relative order. The `choice` introduces the `sh:xone` in the corresponding node shape translated from its parent definition to reference the shapes translated from its child element declarations (e.g., `node_shape sh:xone (s_1 s_2).`). For `group` and `attributeGroup` with an attribute `ref`, we use `sh:node` to reference the node shape translated from the referenced group declaration.

3.2 Core Constraints Generation

In this subsection, we present the correspondence between XSD components and SHACL constraints (Tables 2 and 3).

Extension and restriction. The `extension` and `restriction` within a complex type definition are translated according to their `base` type. If their `base` type is a complex type definition, we use `sh:node` to reference that `base` type's node shape from this definition's node shape. For example, in Listing 2, supposed that the `all` group within the `StudentType` is extended by another global complex type definition `FamilyType` with two child element declarations `phoneNumber` and `address` within `all`. Then we reference the node shape translated from `FamilyType` using `sh:node` from the `StudentType`'s node shape to specify that a `student` may have an address and phone number, besides an id and grade score. When the `base` type is a built-in type, we define a `sh:datatype` constraint

Table 2: Correspondences from XSD extension, restriction, and facets to SHACL constraints

XML Schema	SHACL
Complex type C <xs:restriction base="*C*"/> <xs:extension base="*C*">	sh:node
Built-in simple type C <xs:extension base="*C*"/> <xs:restriction base="*C*"/>	sh:datatype *C*
<xs:pattern value="*C*"/>	sh:pattern *C*
<xs:minLength value="*C*"/>	sh:minLength *C*
<xs:maxLength value="*C*"/>	sh:maxLength *C*
<xs:Length value="*C*"/>	sh:minLength *C* sh:maxLength *C*
<xs:minInclusive value="*C*"/>	sh:minInclusive *C*
<xs:maxInclusive value="*C*"/>	sh:maxInclusive *C*
<xs:minExclusive value="*C*"/>	sh:minExclusive *C*
<xs:maxExclusive value="*C*"/>	sh:maxInclusive *C*
<xs:enumeration value="*C1*"/> <xs:enumeration value="*C2*"/>	sh:in (*C1*, *C2*)

with the built-in type as value. For example, assuming that a simple content complex definition `studentType` contains an `extension` with `xs:integer` as `base`, then we translate it to `sh:datatype` and incorporate it into the corresponding property shape.

We translate `restriction` within simple type definition according to its `base` type. If the type is a user-defined simple type, we incorporate the constraints translated from the user-defined type into the shape translated from the declaration that defines this definition. For example, supposed that the simple type definition (Listing 2, lines 7-12) has another user-defined definition with facets `pattern` and `maxInclusive` `100` as its `base` type, then we add translated `sh:pattern` and `sh:maxInclusive` to the `grade`'s shape but the `sh:maxInclusive 100` will be override by `sh:maxInclusive 20` later. If the type is a built-in type (e.g., `xs:integer`), then we incorporate a `sh:datatype` within the shape translated from the declaration that defines it (e.g. Listing 3, line 10).

Facets. We translate the facets within a `restriction` into corresponding SHACL constraints following Table 2. Boundary facets, such as `minExclusive`, are translated into the corresponding value range constraints, such as `sh:minExclusive`. The length facets of `minLength` or `maxLength`, is translated to `sh:minLength` or `sh:maxLength`. The length facet of `Length`, with no aligned constraint in SHACL, is translated to `sh:minLength` and `sh:maxLength` with the same value to specify the unique range. For enumeration facets, we translate all those within the same `restriction` to a `sh:in` followed by an RDF list containing all enumerated values. For facet of `pattern`, we translate it to `sh:pattern`. We skip translating the facets of `assertion`, `explicitTimezone`, `whiteSpace`, `totalDigits`, and `fractionDigits` that do not have corresponding constraints in SHACL. We incorporate these translated constraints into the property shape translated from the declaration that defines the type definition that contains this `restriction` as its type. For example, in Listing 2 and Listing 3, we translate the facets within the simple type definition (lines 7-12) that is locally defined within `grade` declaration, such as `minInclusive`, to the

Table 3: Correspondences from *attributes* to constraints

XML Schema	SHACL
Local element declaration	
`minOccurs`="*C*" *default to 1*	sh:minCount *C*
`maxOccurs`="*C*" *default to 1*	sh:maxCount *C*
`maxOccurs`="*unbounded*"	–
Local attribute declaration	
`use`="*required*"	sh:minCount 1 sh:maxCount 1
`use`="*optional*"	sh:minCount 0 sh:maxCount 1
`use`="*prohibited*"	sh:minCount 0 sh:maxCount 0
`default`="*C*"	sh:defaultValue *C*
`fixed`="*C*"	sh:in (*C*)
`name`="*N*"	sh:name *N*

corresponding constraints such as `sh:minInclusive`, and incorporate them into the `grade`'s property shape.

Attributes. The translations of attributes defined in element declarations and attribute declarations are shown in Table 3. For Occurrence attribute `minOccurs` and `maxOccus` within local element declaration, we translate them to the cardinality constraints `sh:minCount` and `sh:maxCount`, The default value for both occurrence attributes is 1. For the `use` attribute, we translate it to `sh:minCount 1` and `sh:maxCount 1` if its value is "*required*", to `sh:minCount 0` and `sh:maxCount 1` if its value is "*optional*", and to `sh:minCount 0` and `sh:maxCount 0` if its value is "*prohibited*". We translate `default` to `sh:defaultValue`, and `fixed` to `sh:in` followed by an RDF list contains the value of `fixed`. The translated constraints will also be incorporated into the shape deriving from the element or attribute declaration that defines the attributes.

4 XSD2SHACL IMPLEMENTATION

In this section, we present the pseudocode (Algorithm 1) of the implementation[1] for our XSD to SHACL translation.

The input XSD file, including all relevant files associated through `include` and `import`, are recursively parsed to a tree, where the names involving different targetNamespace in `import` will be converted to explicit prefixed names, then the tree root, initialized empty graph, and *previous* shape (*None*) are fed into the *translate* function for translating its children through the loop (line 8).

For each element and attribute reference, the shape translated from the referenced declaration becomes the current shape (lines 10-11). For each element and attribute declaration associated with a simple type definition, we create a property shape with its `name` as `sh:path`, incorporate the constraints translated from its *attributes* and type definition, and then end the iteration (lines 12-17). For each element declaration associated with a complex type definition, we create a node shape with its `name` as `sh:targetClass`, add `sh:node` to reference the node shape translated from its type definition if it is global, and add possible extra property shape as Table 1 shows (lines 18-21). If the *previous* shape is not *None*, which signifies an inherent hierarchical association between the two shapes, we use `sh:node` or `sh:property` to reference the current shape from the

Algorithm 1: Pseudocode for translating XSD to SHACL

```
1  Function XSD2SHACL(XSD_file):
2      tree ← parse(XSD_file)
3      tree ← recursiveParse(tree, tree.include, tree.import)
4      S ← empty graph
5      S ← translate(tree.root, S, None)
6      return S
7  Function translate(element, S, previous_shape):
8      for child in element do
9          if isElement(child) or isAttribute(child) then
10             if isRef(child) then
11                 s ← redirect(child.ref)
12             else if isSimple(child) then
13                 s ← createPropertyShape(path, child.name)
14                 s.addConstraints(child.attributes)
15                 s.addConstraints(find(child.type))
16                 previous_shape ← None
17                 continue
18             else if isComplex(child) then
19                 s ← createNodeShape(target, child.name)
20                 s.addNode(redirect(child.type))
21                 s.addExtraShape(contentType(child.type))
22             if previous_shape is not None then
23                 s.addNodeOrProperty(shapetype(previous_shape))
24         else if isGlobalComplexType(child) or isGroup(child) or
             isAttributeGroup(child) then
25             s ← createNodeShape(child)
26         else
27             previous_shape.addConstraints(child)
28         translate(child, S, s)
29         previous_shape ← None
30     return S
```

previous shape (lines 22-23). For each global complex type definition, group, and attribute group, we create a node shape without `sh:targetClass` (lines 24-25). Components that do not involve creating new shapes and are not in a simple type definition are translated to corresponding constraints following Section 3, and are incorporated into the *previous* shape (line 27). Subsequently, we reintroduce the current child node into the *translate* function, thereby ensuring continued traversal through its potential child nodes (line 28). Eventually, the resulting SHACL shape is returned as the final outcome.

5 VALIDATION

We validate our translation against the shapes from other translations (Section 5.1) and human-defined SHACL shapes (Section 5.2). All resources are published with the code[1].

5.1 XSD2SHACL Vs. XMLSchema2ShEx

We compare our implementation with XMLSchema2ShEx[2] [10]. We assess their coverage across various XSD and SHACL components, and examine the similarities and differences of the produced shapes.

Table 4: Supported XSD components of XMLSchema2ShEx and XSD2SHACL. The "*" indicates that the generated ShEx constraint cannot be converted to SHACL using Shaclex.

XMLSchema2ShEx	XSD2SHACL
all, attribute, complexType, element, enumeration*, fixed,* minOccurs, maxOccurs, maxExclusive*, maxInclusive*, minExclusive*, minInclusive*, pattern*, simpleType	all, annotation, appinfo, attribute, attributeGroup, complexContent, choice, complexType, default, element, documentation, enumeration, extension, fixed, group, import, include, minOccurs, maxOccurs, length, maxExclusive, maxInclusive, maxLength, minExclusive, minInclusive, minLength, pattern, sequence, simpleContent, simpleType, union, use

Methodology. We compare the two implementations on different examples; we indicatively discuss the purchase order schema example from the XML Schema specification[3]. We generated SHACL shapes with XSD2SHACL and ShEx shapes with XMLSchema-2ShEx. We used Shaclex[4] to convert the ShEx shapes generated by XMLSchema2ShEx to SHACL. Then we observe their similarities and differences. Originally, the XMLSchema2ShEx could not produce results for the complete example XSD file, due to certain components that are not supported as confirmed by the authors[5].

Therefore, we divided this purchase order schema example to more fine-grained test cases so that XMLSchema2ShEx can produce shapes. Components within the example, e.g., `complexType`, are extracted separately as individual cases with the smallest combined unit that can be translated as target. For example, if the schema contains multiple components, we extract a complex type definition, an element declaration with a built-in, user-defined or complex type definition, etc., as independent cases. Components, such as the `attributeGroup`, that are not included in the example are added for completeness. An extract of the XSD file is in Listing 4 and an extract of the shapes produced by XMLSchema2ShEx+Shaclex and XSD2SHACL in Listings 5 and 6 respectively.

Results. The comparative evaluation reveals that our implementation exhibits a broader coverage of both XSD and SHACL components (Table 4). Seven of the XSD components are fully supported by XMLSchema2ShEx. Another seven components are covered but produce ShEx shapes with syntax errors which cannot be converted to SHACL using Shaclex. XSD2SHACL successfully generates SHACL shapes on all cases and with valid syntax.

Both implementations generate equivalent property shapes and constraints for declarations that are associated with simple type definitions. For example, in Listing 5 and Listing 6, both shapes contain the same property shapes translated from `productName` and `USPrice`, while XSD2SHACL generates one more `sh:name`.

Shapes translated from declarations with complex type definitions are not equivalent due to the different assumptions that XMLSchema-2ShEx used as explained in Section 2.2. For example, given the XSD in Listing 4, XMLSchema2ShEx+Shaclex translates the `item` declaration associated with a complex type definition to a property shape with `:item` as `sh:path` (Listing 5, line 2), and references the node shape generated from the complex type definition. XSD2SHACL translates `item` declaration into a node shape with `:item` as `sh:targetClass` (Listing 6, lines 6-10), and references the property shapes translated from the child declarations (lines 8-9). In addition, XMLSchema2ShEx+Shaclex does not translate the element declaration `items` while XSD2SHACL covers (lines 1-3).

```
1  <xs:element name="items" type="Items"/>
2  <xs:complexType name="Items">
3    <xs:all>
4      <xs:element name="item" minOccurs="0" maxOccurs="unbounded">
5        <xs:complexType>
6          <xs:all>
7            <xs:element name="productName" type="xs:string"/>
8            <xs:element name="USPrice" type="xs:decimal"/>
9          </xs:all>
10       </xs:complexType>
11     </xs:element>
12   </xs:all>
13 </xs:complexType>
```

Listing 4: Example XSD

```
1  <Items>  a  sh:NodeShape ;
2    sh:property  [ sh:path  :item ;  sh:node  <item> ] .
3  <item>  a  sh:NodeShape ;
4    sh:property  [ sh:path  :productName ;
5                   sh:minCount  1 ;  sh:maxCount  1 ;
6                   sh:datatype  xs:string ] ;
7    sh:property  [ sh:path  :USPrice ;
8                   sh:minCount  1 ;  sh:maxCount  1 ;
9                   sh:datatype  xs:decimal ] .
```

Listing 5: SHACL shapes from XMLSchema2ShEx & Shaclex

```
1  <NS/items> a sh:NodeShape ;
2    sh:nodeKind sh:IRI  ;  sh:node <NS/Items> ;
3    sh:targetClass :items ;  sh:name "items" .
4  <NS/Items> a sh:NodeShape ;
5    sh:node <NS/Items/item> ;  sh:name "Items" .
6  <NS/Items/item> a sh:NodeShape ;
7    sh:nodeKind sh:IRI  ;
8    sh:property <PS/Items/item/USPrice>,
9      <PS/Items/item/productName> ;
10   sh:targetClass :item  ;  sh:name "item" .
11 <PS/Items/item/USPrice> a sh:PropertyShape ;
12   sh:path :USPrice ;
13   sh:minCount  1 ;  sh:maxCount  1 ;
14   sh:datatype xsd:decimal ;  sh:name "USPrice" .
15 <PS/Items/item/productName> a sh:PropertyShape ;
16   sh:path :productName ;
17   sh:minCount  1 ;  sh:maxCount  1 ;
18   sh:datatype xsd:string ;  sh:name "productName"  .
```

Listing 6: SHACL shapes from XSD2SHACL

5.2 RINF and TED Use Cases

We conduct an analysis on two use cases by comparing our generated SHACL shapes with human-created SHACL shapes. We

[3]https://www.w3.org/TR/xmlschema-0/#POSchema
[4]https://github.com/weso/shaclex
[5]https://github.com/herminiogg/XMLSchema2ShEx/issues/7

Table 5: Validation results on the RINF and TED use cases

Use Case	Target Declaration			Property Path		
	C_T	R/T	R/T'	C_P	R/P	R/P'
RINF						
contact-line	1	1/1	1/1	8	8/8	**10/10**
etcs-levels	1	1/1	1/1	2	2/2	**3/3**
meso-net-e	1	1/1	1/1	2	2/2	2/2
meso-net-r	1	1/1	1/1	2	2/2	2/2
op-tracks	1	1/1	1/1	11	11/11	**13/13**
operational	2	2/2	**5/5**	13	13/13	**23/23**
platforms	1	1/1	1/1	7	7/7	**9/9**
sections-of	1	1/1	1/1	9	9/9	**14/14**
sidings	1	1/1	1/1	16	16/16	**18/18**
sol-tracks	1	**1/2**	**1/1**	95	95/107	**110/110**
train-detect	1	1/1	1/1	24	24/26	**30/30**
tunnels	1	1/1	**3/3**	13	13/14	**21/21**
TED						
F03	50	50/278	50/**50**	129	132/493	**144/144**
F06	50	50/278	50/**50**	129	134/493	**144/144**
F13	50	50/278	50/**50**	127	132/493	**142/142**

validate our translation and implementation on two real-world use cases in which the RDF graphs are constructed using RML mappings [5], and the XSD and SHACL shapes are available: Register of Infrastructure (RINF)[6] and Tenders Electronic Daily (TED)[7].

RINF is a base registry maintained by the European Union Agency for Railways (ERA)[8]. We use the version 1.5 of RINF XSD Schema[9], and version 2.6.3 of the SHACL shapes [17]. The Publications Office of the European Union publishes public procurement notices on the **TED** website using pre-defined XML Schema, and we use the Publication XSD[10] and SHACL shapes[11]. In both cases, the SHACL shapes are defined without considering the XSD.

Methodology. We employ the XSD files as input for XSD2SHACL to generate *preliminary SHACL shapes*. As the *human-defined SHACL shapes* use custom classes and properties, they cannot be directly compared. Thus, we *post-adjust* the *preliminary SHACL shapes* to align the classes and properties to the custom ones, and align the constraints to the target RDF data yielding the final *post-adjusted SHACL shapes* used for the comparison. To achieve this, we leverage the available RML mappings[1] [5] of the two use cases, which describes how classes and properties are applied to raw data to construct RDF graphs. We replace the classes and properties in the *preliminary SHACL shapes* with those in the mappings. Then we conduct the comparison based on diverse metrics.

Metrics. We use the number of common target classes C_T and common property paths C_P, the proportion of the number of target classes and property paths that are declared in RML (R/T, R/T', R/P and R/P'), and the observed constraint richness as metrics. Regarding common target classes, given the *human-defined shapes* S with a set of target classes T_S and *post-adjusted shapes* S' with $T_{S'}$, we have $C_T = |T_S \cap T_{S'}|$. Given the declared classes V in RML mappings, we have the proportion of *human-defined target classes* that are also in V as $R/T = |T_S \cap V|/|T_S|$. For S', we have $R/T' = |T_{S'} \cap V|/|T'_S|$. Regarding common property paths, given a set of property paths P_S in S and $P_{S'}$ in S', we have the $C_P = |P_S \cap P_{S'}|$, $R/P = |P_S \cap V|/|P_S|$, and $R/P' = |P_{S'} \cap V|/|P'_S|$.

RINF results. XSD2SHACL generates SHACL shapes comparable to the *human-defined SHACL shapes* in the RINF use case (Table 5). It covers all *human-defined target classes* (C_T) across all but 1 case and all *human-defined property paths* across 9 cases. In the *sol-tracks* case (R/T has value 1/2), the *human-defined SHACL shapes* comprise 2 target classes, whereas only 1 of these appeared in the RML mappings and our shapes. As the *post-adjusted SHACL shapes* are based on the mappings, only classes and properties declared in the mappings will appear in these shapes. This means that some constraints could have been in the *preliminary SHACL shapes* but were discarded during the post-adjustment, but also that, in this case, the *human-defined SHACL shapes* include constraints for RDF terms which do not appear in the RDF graph so far.

In some cases, the *post-adjusted SHACL shapes* have more target classes than the *human-defined* (e.g., for the *tunnels* and *operational*). In these cases, the *human-defined target classes* are all declared in the RML mappings (e.g., R/T of 2/2 and 1/1 respectively), but they cover fewer RML classes than the *post-adjusted SHACL shapes* (e.g., R/T' of 5/5 and 3/3 respectively). Thus, the *human-defined SHACL shapes* do not define constraints to validate part of the RDF graph.

Similarly, not all property shapes of the *human-defined SHACL shapes* appear in the *post-adjusted SHACL shapes* as the corresponding properties may not be declared in the RML mappings. For example, in the case of *sol-tracks*, only 95 of the 107 *human-defined property paths* are declared in the RML mappings, and, thus, the *post-adjusted SHACL shapes* do not include constraints for these *human-defined property paths* not declared in RML (e.g., C_P of 95).

We observe that the node and property shapes include the same cardinality and data type constraints. However, the SHACL shapes produced by XSD2SHACL contain length-related constraints that do not exist in the *human-defined SHACL shapes*. On the other hand, the *human-defined SHACL shapes* contain pattern constraints that the SHACL shapes produced by the XSD2SHACL do not have. This divergence is primarily due to inconsistent versions of XSD and *human-defined SHACL shapes*. While the XSD is legacy, the SHACL shapes are defined according to the Application Guide v1.6.1[10], which is updated compared to the XSD. Thus, the XSD contains length-related but no pattern constraints.

Last, the *human-defined SHACL shapes* include `sh:severity` and `sh:message` properties which could not be inferred from the XSD.

TED results. *Human-defined SHACL shapes* cover more target classes and property paths than the *post-adjusted SHACL shapes*. The significant disparity between T and T' necessitates an exploration in the context of the RML mappings. Taking F03 as an example, the R/T of 50/278 indicates that out of the 278 *human-defined target classes*, only 50 are declared in the RML mappings. The R/T' of 50/50 indicates that all of the 50 *post-adjusted target classes* are declared in RML mappings, and the C_T of 50 indicates that all of those coincided with the *human-defined target classes*. Therefore,

[6] https://www.rinf-ch.ch
[7] https://ted.europa.eu
[8] https://www.era.europa.eu/
[9] https://www.era.europa.eu/domains/registers/rinf_en
[10] https://op.europa.eu/en/web/eu-vocabularies/e-procurement/tedschemas
[11] https://github.com/OP-TED/ted-rdf-mapping

the *post-adjusted SHACL shapes* are also covered by the the *human-defined SHACL shapes*, but the latter also contain constraints for RDF terms that do not appear yet in the RDF graph.

In terms of common property paths, the *human-defined property paths* are more than the property paths generated from the XSD2SHACL because they refer to RDF constructed from more data sources than just the XML files. Take the F03 as an example, the C_P of 129 and R/P of 132/493 indicate that there are three properties declared in the RML mappings but not covered by the SHACL shapes produced by XSD2SHACL. These properties are defined within rules associated with the CSV source. This also underscores the fact that the target RDF data of the three human-defined property paths are not derived from XML. Consequently, our inability to discern these properties should not be perceived as a negative result.

Performance. While a performance evaluation is beyond the scope of this paper, we indicatively report our system's time performance to demonstrate that results are produced in an acceptable timeframe. Our system effectively translates XSD to 198 SHACL shapes in less than 1 second for RINF which has 1 XSD file and to 1,144 SHACL shapes in around 19 seconds for TED which is an assembly of 30 XSD files. Thus, XSD2SHACL consistently exhibits robust performance even for larger use cases.

Discussion. The SHACL shapes generated by XSD2SHACL are comparable to the *human-defined SHACL shapes* to the extent we can compare. On one hand, in RINF, the target declarations of the *human-defined SHACL shapes* closely resemble the *post-adjusted SHACL shapes* but the latter includes additional property paths, encompassing all properties present in the RML mappings. On the other hand, in TED, while the *post-adjusted SHACL shapes* encompass all target classes and property paths found in the RML mappings, the *human-defined SHACL shapes* go well beyond the target classes covered by the *post-adjusted SHACL shapes*. Although the *preliminary SHACL shapes* may initially encompass more of the *human-defined SHACL shapes*, some might have been eliminated during the post-adjustment.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a translation from XSD to SHACL and provide an algorithm which is implemented in the XSD2SHACL system. The comparative experiments demonstrate that our translation covers more XSD components than XMLSchema2ShEx. The validation experiments showcase our efficiency and effectiveness in handling intricate real-world scenarios, outperforming the state-of-the-art systems. We generate SHACL shapes that are comparable to the human-created SHACL shapes, and delineate the disparities of the two SHACL shapes. While there are disparities, the real-world use cases still reveal the efficacy of our system. In conclusion, using legacy XSD, we could generate a draft valid version of SHACL shapes that helps humans enhance and update them rather than beginning anew. In the future, we will investigate generalizing the post-adjustment method to more shapes generation approaches.

ACKNOWLEDGMENTS

This research was partially supported by Flanders Make via the REXPEK project, the strategic research centre for the manufacturing industry and the Flanders innovation and entrepreneurship (VLAIO) via the KG3D project. David Chaves-Fraga is funded by the Galician Ministry of Education, University and Professional Training and the European Regional Development Fund (ERDF/FEDER program) through grants ED431C2018/29 and ED431G2019/04).

REFERENCES

[1] Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux, and Juan Sequeda. 2012. *A Direct Mapping of Relational Data to RDF.* Recommendation. World Wide Web Consortium (W3C). http://www.w3.org/TR/rdb-direct-mapping/

[2] Andrea Cimmino, Alba Fernández-Izquierdo, and Raúl García-Castro. 2020. Astrea: Automatic Generation of SHACL Shapes from Ontologies. In *European Semantic Web Conference (ESWC)*. Springer, 497–513. https://doi.org/10.1007/978-3-030-49461-2_29

[3] Souripriya Das, Seema Sundara, and Richard Cyganiak. 2012. *R2RML: RDB to RDF Mapping Language.* Recommendation. W3C. http://www.w3.org/TR/r2rml/

[4] Thomas Delva, Birte De Smedt, Sitt Min Oo, Dylan Van Assche, Sven Lieber, and Anastasia Dimou. 2021. RML2SHACL: RDF Generation Taking Shape. In *Proceedings of the 11th on Knowledge Capture Conference.* ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/3460210.3493562

[5] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the 7th Workshop on Linked Data on the Web*, Vol. 1184. CEUR Workshop Proceedings. http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf

[6] David Fallside and Priscilla Walmsley. 2004. *XML Schema Part 0: Primer Second Edition.* Recommendation. W3C. https://www.w3.org/TR/xmlschema-0/

[7] Rémi Felin, Catherine Faron, and Andrea G. B. Tettamanzi. 2023. A Framework to Include and Exploit Probabilistic Information in SHACL Validation Reports. In *The Semantic Web*, Vol. 13870. Springer, 91–104. https://doi.org/10.1007/978-3-031-33455-9_6

[8] Daniel Fernández-Álvarez, H. García-González, Johannes Frey, S. Hellmann, and Jose Emilio Labra Gayo. 2018. Inference of Latent Shape Expressions Associated to DBpedia Ontology. In *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018)*, Vol. 2195. CEUR Workshop Proceedings, 52–66. https://ceur-ws.org/Vol-2195/research_paper_2.pdf

[9] Daniel Fernandez-Álvarez, Jose Emilio Labra-Gayo, and Daniel Gayo-Avello. 2022. Automatic extraction of shapes using sheXer. *Knowledge-Based Systems* 238, C (Feb. 2022), 107975. https://doi.org/10.1016/j.knosys.2021.107975

[10] Herminio Garcia-Gonzalez and Jose Emilio Labra-Gayo. 2020. XMLSchema2ShEx: Converting XML validation to RDF validation. *Semantic Web* 11, 2 (2020), 235–253. https://doi.org/10.3233/SW-180329

[11] Dave Lewis Harshvardhan J. Pandit, Declan O'Sullivan. 2018. Using Ontology Design Patterns to Define SHACL Shapes. In *9th Workshop on Ontology Design and Patterns (WOP2018)*, Vol. 2195. CEUR Workshop Proceedings, Monterey California, USA, 67–71.

[12] ISO/IEC 9075-1:2023 2023. *Information technology — Database languages SQL — Part 1: Framework (SQL/Framework).* Standard. International Organization for Standardization. https://www.iso.org/standard/76583.html

[13] Holger Knublauch and Dimitris Kontokostas. 2017. *Shapes Constraint Language (SHACL).* Recommendation. W3C. https://www.w3.org/TR/shacl/

[14] Jose Emilio Labra Gayo, Herminio Garciía González, Daniel Fernández Álvarez, and Eric Prud'hommeaux. 2019. *Challenges in RDF Validation.* Studies in Computational Intelligence, Vol. 815. Springer International Ablex Publishing Co, Chapter 6, 121–151. https://doi.org/10.1007/978-3-030-06149-4_6

[15] Nandana Mihindukulasooriya, Mohammad Rifat Ahmmad Rashid, Giuseppe Rizzo, Raul Garcia-Castro, Oscar Corcho, and Marco Torchiano. 2018. RDF Shape Induction using Knowledge Base Profiling. In *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing (SAC '18).* 1952–1959. https://doi.org/10.1145/3167132.3167341

[16] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2023. Extraction of Validating Shapes from Very Large Knowledge Graphs. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1023–1032. https://doi.org/10.14778/3579075.3579078

[17] Edna Ruckhaus, Oscar Corcho, Julián Rojas, Dylan van Assche, Ivo Velitchkov, Pieter Colpaert, and Wouter Beek. 2023. *ERA Vocabulary.* https://doi.org/10.5281/zenodo.7775344

[18] Blerina Spahiu, A. Maurino, and M. Palmonari. 2018. Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL. In *Workshop on Ontology Design Patterns (WOP) at ISWC (Best Workshop Papers)*, Vol. 2195. CEUR Workshop Proceedings, 67–71.

[19] Ratan Bahadur Thapa and Martin Giese. 2021. A Source-to-Target Constraint Rewriting for Direct Mapping. In *The Semantic Web – ISWC 2021.* Springer, 21–38. https://doi.org/10.1007/978-3-030-88361-4_2

[20] Ratan Bahadur Thapa and Martin Giese. 2022. Mapping Relational Database Constraints to SHACL. In *The Semantic Web – ISWC 2022: 21st International*

Semantic Web Conference, Virtual Event, October 23–27, 2022, Proceedings. Springer, 214–230. https://doi.org/10.1007/978-3-031-19433-7_13

[21] Dylan Van Assche, Thomas Delva, Gerald Haesendonck, Pieter Heyvaert, Ben De Meester, and Anastasia Dimou. 2023. Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review. *Journal of Web Semantics* 75 (2023), 100753. https://doi.org/10.1016/j.websem.2022.100753

[22] Priscilla Walmsley. 2012. *Definitive XML Schema, 2nd Edition*. Pearson Education.

[23] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. 2022. *JSON Schema: A Media Type for Describing JSON Documents*. Internet-Draft draft-bhutton-json-schema-01. IETF Secretariat.