KG seminar - Validation of Knowledge Graphs: SHACL, ShEx and Friends

Merlin Bögershausen - 317962 - merlin.boegershausen@rwth-aachen.de

RWTH Aachen University
Templergraben 55
52062 Aachen

Abstract. Knowledge graphs are used in all kind of industrial and academical environments to store and work with knowledge. Knowledge and RDF are diversified techniques, to gain better knowledge from them, we need merging and validation as first steps in our workflows.
RDF does not have a default or recommended way to do such validation, therefore some promising approaches are appearing. But none of them is familiar to the widely know Object Constraint Language, which is the quasi-standard tool for constraint definition.
We will show how transformation from OCL to SPARQL Inference Modeling, Shape Constraints and Shape Expressions can be done, to open this technique to the UML trained software developer and architects.

Keywords: Knowledge Graph · Data Validation · Automation

1 Introduction

Knowledge graphs gaining more and more attention in the last few years, due to the growing amount of information spread around the world wide web. With the recommendation of the Resource Description Framework (RDF) by the W3C, the way to receive and populate such information is getting standardized.

RDF stores information in a describing way, by paring an object with property and value. The objects and properties are IRIs and the value can be an IRI or literal, usually out of the standard XSD namespace. This pair is named a *Triple* and describes an arc in a labelled directed graph, the *RDF-Graph*.

Using RDF as the base of data exchange in a workflow leads us to various problems [5]. Miss matched data and missing data are likely to be present if the source graph uses another or unexpected structure. In this case, no information may be found and due to the open-world assumption, no errors will be raised. Therefore, validation is needed to prevent the workflow from unknown behaviour.

The classic framework for modelling and validation in software development provides us with all features to tackle and solve this problem, by using the Object Constraint Language (OCL). The validation with such constraints is normally done by adding assertions/checks at code generation, but RDF does not provide such feature natively.

In this paper, we will show how the translation from OCL into some of the upraising RDF validation languages, can be done. For this, we give a short overview of OCL and find some metric in Section 2. In Section 3 we show how OCL can be translated into SPARQL Inference Notation (SPIN), Shape Constraint Language (SHACL) and Shape Expressions (ShEx) by defining such translation functions and combining them to one. In Section 4 we will see the differences of the three approaches.

2 Basic Knowledge

2.1 Object Constraint Language

The Object Management Group is a consortium that produces and maintain specifications for enterprise applications, like the widely known and used Unified Modeling Language (UML). As part of UML, the Object Constraint Language was developed to provide the ability to formulate statements about the structure of data and to specify the input and output of functions [2]. Therefore OCL defines various expression types, but not all are needed to validate the data integrity.

In this paper, we will focus on the invariant feature of OCL, with which conditions can be defined that have to hold for the whole life span of the data.

2.2 OCL Expressions

Invariants are OCL Expressions, so we will use a sub grammar of the OCL grammar which meets our requirements, the full grammar can be seen in [2].

Primitive Type Expressions are used to define which range or specific type value must-have. All basic datatypes are covered, like Integer, Real and UnlimitedNatural for all numerical values, Boolean and String as usual. The Null expression is used to cover all invariants containing a null statement.

Variable Expressions are used for declaring on which variable the invariant must hold. Therefore, a self-pointer or every other String, except the OCL keywords are allowed. This expression is used in other expressions.

Let Expressions define a variable with which further, more complex validation can be done. This expression does not influence any value in the data.

Collection Literal Expressions are used to write invariants about a collection of values.

The expressions mentioned above can be represented with the union of the sets below.

1. IfEXP := OCLExpression × OCLExpression × OCLExpression
2. PrimitivTypeEXP := IntegerEXP∪RealEXP∪BooleanEXP∪UnlimitedNatrualEXP∪ NullEXP
3. VariableEXP := "$self$" ∪ (String \ OCLKeywords)
4. LetEXP := DeclareEXP × OCLExpression
5. DeclareEXP := String × DataType × OCLExpression
6. DataType := OCLType ∪ SelfDefinedType
7. CollectionLiteralEXP := CollectionTyp × {OCLExpression*}

2.3 Metrics for RDF Validation Frameworks

The ongoing research about the RDF validation topic gave us some requirements [6] for a validation language, which are not suitable for our purpose.

We need some other metrics to validate the given approaches, these will be the following three.

1. feature coverage for data value validation
2. feature coverage for data structure validation
3. human readability of the resulting constraints

3 From OCL to RDF validation framework

Now we give the translation relations from OCL to SPIN, SHACL and ShEx by combining sub-feature translation relations.

3.1 Translation from OCL to SPIN

The RDF SPIN vocabulary provides RDF properties to represent SPARQL queries with the RDF triple notation, this enables storing and reusing of queries and rule propagation to other endpoints [3].

The whole validation magic will take place in the where clause of a SPIN ASK Query. In this section, we only give the SPARQL queries and not the SPIN vocabulary notation.

We will see SPINWhere which is the set of all valid SPIN where clauses.

PrimitivTypeEXP will be mapped to a pair of a triple clause and a FILTER clause, where the compare operator mapping σ_{op} is used. For this, we bind the concrete value to a variable, which later will be compared.

Definition 1 (PrimitivTypeEXP to SPIN Transformation).

$$\sigma_{PT}\colon \text{PrimitivTypeEXP} \to \text{SPINWhere}$$
$$\sigma_{PT}((n, o, v)) \mapsto \$this\ ?n\ ?val.\ FILTER(\sigma_{op}(val, o)\ v).$$

Basic Operators need to be mapped for each datatype individually. This makes the relation very huge, we choose a table for visualization.

Definition 2 (OCL to SPIN Transformation of basic Operators).

$$\sigma_{op}\colon \text{OCLOperator} \times \text{OCLObjects} \to \text{SPINOperator}$$
$$OP_{OCL} \mapsto OP_{SPIN}$$

Where OP_{OCL} and OP_{SPIN} have the non trivial mapping seen in Tables 1, 2, 3.

Integer		Real	
OP_{OCL}	OP_{SPIN}	OP_{OCL}	OP_{SPIN}
a.div(b)	a/b	a/b	a/b
a.mod(b)	FLOOR(a-(b*FLOOR(a/b))	a.floor()	FLOOR(a)
a.abs()	ABS(a)	a.round()	ROUND(a)
a.max(b)	MAX(a,b)	a.max(b)	MX(a,b)
a.min(b)	MIN(a,b)	a.min(b)	MIN(a,b)

Table 1. Operator mapping for numerical values

Boolean		String	
OP_{OCL}	OP_{SPIN}	OP_{OCL}	OP_{SPIN}
not a	!a	a.size()	STRLEN(a)
a and b	a && b	a.concat(b)	CONCAT(a,b)
a or b	a || b	a.substring(i,j)	SUBSTR(a,i,j)
a xor b	(!a && b)||(a && !b)	a.toUpper()	UCASE(a)
a implies b	a && !b	a.toLower()	LCASE(a)
		a.toInteger()	xsd:integer(a)
		a.toReal()	xsd:decimal(a)

Table 2. Operator mapping for Boolean and string values

Object	
x.oclTypeOf(c)	$this x/rdf:type c.
x.oclKindOf(c)	$this x/rdf:type/rdfsSubClassOf* c

Table 3. Operator mapping for all objects

IfEXP cannot be covert native with SPIN, so we use the logical or/and approach. This conditioning then will be parsed into an equivalent FILTER clause.

Definition 3 (IfEXP to SPIN Transformation).

$$\sigma_{IF}\colon \text{IfEXP} \to \text{SPINWhere}$$
$$(e_1, e_2, e_3) \mapsto \textit{FILTER((e\_1 \&\& e\_2) || (!e1 \&\& e\_2))}.$$

CollectionLiteralEXP are all needed to be mapped, so this produces a huge mapping, so we choose a table for non the trivial mappings.

Definition 4 (OCL to SPIN Transformation of CollectionLiteralEXP).

$$\sigma_{COL}\colon \text{CollectionLiteralEXP} \to \text{SPINExpression}$$
$$\sigma_{COL}(e_{ocl}) \mapsto e_{SPIN}$$

The mapping can be seen in Table 4.

OCLExpression translation are a combination of the defined translation relations we seen above, with which we can map all OCL Constraints from Section 2.2.

e_OCL	e_SPIN
x.size()	SELECT COUNT(?x) AS ?count {$this ?x ?v}
a.includes(x)	ASK {$this ?a ?x}
a.excludes(x)	ASK NOT EXIST {$this ?a ?x}
count x	SELECT COUNT(?v) AS ?count {$this ?x ?v}
a.includeAll($x_1..x_n$	ASK {$this ?a ?x_1; .. ?a ?x_n}
a.excludeAll($x_1..x_n$	
a.isEmpty	*empty set is not present in RDF*
a.notEmpty	ASK {$this ?a ?x}
max x	BIND(MAX(?v) AS max)
min x	BIND(MIN(?v) AS min)
sum x	BIND(SUM(?v) AS sum)
product	ERROR
selectByKind(c)	SELECT ?s {?s rdfs:subClassOf* ?c}
selectByType(c)	SELECT ?s {?s rdf:type ?c}
flattern	ERROR

Table 4. OCL Collection based operation to SPIN

Definition 5 (OCL to SPIN translation).

$$\sigma_{SPIN}\colon \text{OCLExpression} \rightarrow \text{SPINExpression}$$

$$\sigma_{SPIN}(e) \mapsto \begin{cases} \sigma_{IF}(e) & , e \in \text{IfExpression} \\ \sigma_{Let}(e) & , e \in \text{LetExpression} \\ \sigma_{PT}(e) & , e \in \text{PrimitivTypeExpression} \\ \sigma_{Col}(e) & , e \in \text{CollectionLiteralEXP} \end{cases}$$

3.2 Translation from OCL to SHACL

SHACL is meant to be the successor of SPIN and like SPIN it is a language for validating RDF graphs against a set of constraints. But SHACL additionally has some goals like user interface building and code generation [4].

Basic Operators are mapped for each datatype individual.

Definition 6 (OCL to SHACL Transformation of basic Operators).

$$\sigma_{op}\colon \text{OCLOperator} \times \text{OCLObjects} \rightarrow \text{SHCALOperator}$$
$$OP_{OCL} \mapsto OP_{SHACL}$$

Where OP_{OCL} and OP_{SHACL} have the following non trivial mapping relations in Tables 5, 6.

Object		Integer		Real	
OP_{OCL}	OP_{SHACL}	OP_{OCL}	OP_{SHACL}	OP_{OCL}	OP_{SHACL}
x.oclTypeOf(c)	sh:datatype c	a.mod(b)		a.floor()	
x.oclKindOf(c)	sh:datatype c	a.abs()		a.round()	
		a.max(b)		a.max(b)	
		a.min(b)		a.min(b)	
		<	sh:minExclusive	<	sh:minExclusive
		>	sh:maxExclusive	>	sh:maxExclusive
		<=	sh:minInclusive	<=	sh:minInclusive
		<=	sh:minInlusive	<=	sh:minInclusive

Table 5. Object, Integer and Real operator mapping for SHACL

Boolean		String	
OP_{OCL}	OP_{SHACL}	OP_{OCL}	OP_{SHACL}
not a	sh:not	a.size()	sh:minLenght and sh:maxLenght
a and b	sh:and	a.concat(b)	sh:pattern: "ab"
a or b	sh:or	a.substring(i,j)	sh:pattern: (RegEX)
a xor b	*combine and/or with not*	a.toUpper()	sh:pattern: (RegEX)
a implies b	*combine and/not*	a.toLower()	sh:pattern: (RegEX)
		a.toInteger()	type conversation not supported
		a.toReal()	type conversation not supported

Table 6. String and Boolean operator mapping for SHACL

PrimitivTypeEXP can be transfomed with σ_{PT} into a SHCALPropertyConstraint instance using *sh:property* relation.

Definition 7 (PrimitivTypeEXP-Transformation).

$$\sigma_{PT}\colon \textsc{PrimitivTypeEXP} \rightarrow \textsc{SHCALPropertyConstraint}$$
$$(n, o, v) \mapsto \textit{sh:property [sh:path n; } \sigma_{op}\textit{(v,o);]}$$

IfEXP can be transformed with $\sigma_(IF)$, the translation relation from OCLs IfEXP to SHACL equivalent.

Definition 8 (IfEXP-Transformation).

$$\sigma_{IF}\colon \textsc{IfEXP} \rightarrow \textsc{SHACLExpression}$$
$$(e_1, e_2, e_3) \mapsto sh : or(sh : and(\sigma_{SHACL}(e_1), \sigma_{SHACL}(e_2)),$$
$$sh : and(sh : not(\sigma_{SHACL}(e_1)), \sigma_{SHACL}(e_3)))$$

VariableEXP are used to produce a variable addressing. We now define σ_{VAR} which transform an instance of OCL into a SHACLExpression.

Definition 9 (VariableEXP-Transformation).

$$\begin{aligned} \sigma_{VAR}\colon \textsc{VarEXP} \rightarrow \textsc{SHACLExpression} & \\ (v) \mapsto' \$this' & \quad ,iff v =' self' \\ v\hat{}\hat{}xsd : string & \quad ,else \end{aligned}$$

LetEXP cannot be covered, because SHACL does not support variables at validation level. With sh:sparql we maybe can bypass this restriction.

CollectionLiteralEXP are various expressions on collections combined in the CollectionLiteralEXP. They enable most of the cardinality features.

Definition 10 (CollectionLiteralEXP-Transformation).

$$\begin{aligned} \sigma_{COL}\colon \textsc{CollectionLiteralEXP} &\rightarrow \textsc{SHACLExpression} \\ (e_{OCL}) &\mapsto e_{SHACL} \end{aligned}$$

The mapping can be seen in 3.2.

e_OCL	**e_SHACL**
=	sh:eq
<>	sh:not(sh:eq)
size()	sh:and(sh:minCount, sh:maxCount)
includes(x)	sh:hasValue x
excludes(x)	sh:not(sh:hasValue x)
count x	sh:sparql(SELECT COUNT(?v) AS count {$this x ?v})
includeAll($x_1..x_n$	sh:in [$x_1..x_n$]
excludeAll($x_1..x_n$	sh:noz(sh:in [$x_1..x_n$])
isEmpty	sh:eq ()
notEmpty	sh:not(sh:eq ())
max x	sh:sparql(SELECT MAX(?v) AS max {$this x ?v})
min x	sh:sparql(SELECT MIN(?v) AS min {$this x ?v})
sum x	sh:sparql(SELECT SUM(?v) AS sum {$this x ?v})
product	ERROR
selectByKind(c)	sh:sparql(SELECT ?s {?s rdfs:subClassOf* ?c})
selectByType(c)	sh:sparql(SELECT ?s {?s rdf:type ?c})
flattern	ERROR

Table 7. OCL Collection based operation to SHACL

OCLExpression translation combine σ_{SHACL} and get a translation function which covers nearby every OCLExpression we allowed in Section 2.2.

Definition 11 (OCL to SHACL translation).

$$\sigma_{SHACL}\colon \text{OCLExpression} \to \text{SPINExpression}$$

$$\sigma_{SHACL}(e) \mapsto \begin{cases} \sigma_{IF}(e) & , e \in \text{IfExpression} \\ \sigma_{Let}(e) & , e \in \text{LetExpression} \\ \sigma_{PT}(e) & , e \in \text{PrimitivTypeExpression} \\ \sigma_{Col}(e) & , e \in \text{CollectionLiteralEXP} \end{cases}$$

3.3 Translation from OCL to ShEx

ShEx is very different from the other two approaches, it follows a more structure orientated idea[1]. It is meant to be used for APIs and UIs.

We will give the JSON notation of ShEx because it is the shortest and most common notation.

CollectionLiteralExp can be translated to whether all or specific elements within this collection match a defined shape. So, a collection validation is a validation of all outgoing arc with the given property, which is covered by σ_{PL}.

IfEXP are used for conditional validation. This feature can be archived with ShEx *ShapeAnd* and *ShapeOr* statements.

Definition 12 (IfEXP to ShEx Transformation).

$$\begin{aligned} \sigma_{IF}\colon \text{IfEXP} &\to \text{ShExShape} \\ (e_1, e_{2,3}) &\mapsto \{\text{"}type\text{"} : \text{"}ShapeOr\text{"}, \text{"}shapeExprs\text{"} : [\\ &\quad \{\text{"}type\text{"} : \text{"}ShapeAnd\text{"}, \text{"}shapeExprs\text{"} : [\\ &\quad\quad \sigma_{ShEx}(e_1), \sigma_{ShEx}(e_2)]\}, \\ &\quad \{\text{"}type\text{"} : \text{"}ShapeAnd\text{"}, \text{"}shapeExprs\text{"} : [\\ &\quad\quad \{\text{"}type\text{"} : \text{"}ShapeNot\text{"}, \text{"}shapeExpr\text{"} : \sigma_{ShEx}(e_1)\} \\ &\quad\quad \sigma_{ShEx}(e_2)]\}]\} \end{aligned}$$

PrimitivTypeEXP are the main magic for ShEx translation. But not all features are supported with the current ShEx version. To enable the basic features we map numericals to ShEx *Numeric* datatype and other to *String* datatype.

Definition 13 (PrimitivTypeEXP to ShEx Transformation).

$$\begin{aligned} \sigma_{PL}\colon \text{PrimitivTypeEXP} &\to \text{ShExShape} \\ (n, o, v) &\mapsto \{\text{"}type\text{"} : \text{"}tripleConstraint\text{"}, \\ &\qquad \text{"}predicate\text{"} : n, \\ &\qquad \text{"}valueExpr\text{"} : \sigma_{op}(o, v)\} \end{aligned}$$

Basic Operator are not fully mappable to ShEx, here we only give the few possibilities.

Definition 14 (OCL to ShEx Transformation of basic Operators).

$$\begin{aligned}
\sigma_{op}\colon \text{OCLOperator} \times \text{OCLObjects} &\to \text{ShExNodeCinstrint} \\
(=, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}values\text{"} : [\{\text{"}value\text{"} : v\}]\} \\
(<>, v) &\mapsto \{\text{"}type\text{"} : \text{"}ShapeNot\text{"}, \text{"}shapeExpr\text{"} : \sigma_{op}(=, v)\} \\
(.size() =, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}lenght\text{"} : v\} \\
(.size() <, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}minLenght\text{"} : v\} \\
(.size() >, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}maxLenght\text{"} : v\} \\
(.size() <=, v) &\mapsto \sigma_{op}(.size() <, v+1) \\
(.size() >=, v) &\mapsto \sigma_{op}(.size() >, v-1) \\
(>, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}maxexlusive\text{"} : v\} \\
(<, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}minxlusive\text{"} : v\} \\
(>=, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}maxinclusive\text{"} : v\} \\
(<=, v) &\mapsto \{\text{"}type\text{"} : \text{"}nodeConstraint\text{"}, \text{"}maxinclusive\text{"} : v\} \\
(o, v) &\mapsto \epsilon
\end{aligned}$$

VarEXP are handled with σ_{PL}, because ShEx does not provide any kind of this or self-structure. All rules are handled with an indirect assignment of this/self.

LetEXP are used to declare a variable for later use. Variables are not supported by ShEx, therefore all let-statements must force an error.

OCLExpression translation are now defined σ_{ShEx} as combination of the translation relations given in this section.

Definition 15 (OCL to ShEx translation).

$$\begin{aligned}
\sigma_{SPIN}\colon \text{OCLExpression} &\to \text{SPINExpression} \\
\sigma_{SPIN}(e) &\mapsto \begin{cases} \sigma_{IF}(e) & , e \in \text{IfExpression} \\ \sigma_{PT}(e) & , e \in \text{PrimitivTypeExpression} \end{cases}
\end{aligned}$$

4 Comparision of the approaches

We have seen that it is possible to translate a $e \in$ OCLExpression to Spin, SHACL or ShEx by applying their translation relation. But the different approaches have different complexities and coverages of OCL features.

4.1 Coverate of validation features

We can see that SPIN and SHACL are very similar at coverage. But it still misses two collection expression, as seen in Table (3.2). In Table 8 the feature coverage of the approaches is compared. One can see that basic structural validation is possible with all approaches, but the ability to validate the values is missing in ShEx.

OCLFeature	**SPIN**	**SHACL**	**ShEx**
Primitiv Type Expression	✓	✓	✓
TypeContraints	✓	✓	✓
If Expression	✓	✓	✓
Variable Expression	✓	✓	(x)
Collection Literal Expression	(✓)	✓	(x)
Let Expression	(✓)	(✓)	x

Table 8. Coverate of OCl features: ✓-covers, x-covers not, ( )-with restriction

4.2 Human Readability

For the value-based validation, we can see that SPIN needs less lines for a simple length check than SHACL or even ShEx. With SHACL it is very clear what is checked. The translation into SHACL would have the most advantages.

In the case of structure validation, we can see that SPIN and SHACL grow heavily with more invariants. The ShEx approach is very easy to read because this is one of the ShEx goals. But ShEx lacks a subType feature (as SHACL) so checking a class hierarchy is impossible.

5 Conclusion

In Section 3 we have seen that with an increasing amount of features the translation function is growing. We also have seen in Sections 4.1 and 4.2 that growing feature coverage implies growing complexity. Currently one can see a process of losing features with newer developments.

That leads us to the conclusion that we should use ShEx over SHACL as long as possible. If they meet their bounds the step back to SPIN is needed.

References

1. Eric Prud'hommeaux, I.B.e.: Shape expressions language 2.1. draft community group report (2018)
2. Group, O.O.M.: Object constraint language, v2.4 (2014)
3. Knublauch, H.: Spin-modeling vocabulary. w3c member submission (2014)

4. Knublauch, H., Kontokostas, D.: Shapes constraint language (shacl). w3c recommendation, w3c, july 2017
5. Sadiq, S., Orlowska, M., Sadiq, W., Foulger, C.: Data flow and validation in workflow modelling. In: Proceedings of the 15th Australasian database conference-Volume 27. pp. 207–214. Australian Computer Society, Inc. (2004)
6. Tomaszuk, D.: Rdf validation: A brief survey. In: International Conference: Beyond Databases, Architectures and Structures. pp. 344–355. Springer (2017)