An Ontology-based Representation for Shaping Product Evolution in Regulated Industries*

Barbara Gallina[1], Henrik Dibowski[2], and Markus Schweizer[2]

[1] Mälardalen University, Västerås, Sweden, barbara.gallina@mdu.se
[2] Robert Bosch GmbH, Renningen, Germany, {name.surname}@de.bosch.com

Abstract. Compliance management is a challenging activity in regulated industries. This is due to the need of navigating rapidly shifting requirements. In the automotive domain products (items) evolve rapidly creating a product line in time in addition to the one in space. The product variability is constrained by legislation, standards, etc. whose applicability may vary depending on the jurisdiction. In this paper, we focus on the legislation and its impact on the product in terms of inclusion/exclusion of product features. We exploit the Semantic Web technologies and we propose an ontology-based representation for managing product variability in compliance with legislation. Specifically, we provide the representation in Resource Description Framework (RDF) and we introduce reusable SHACL (Shapes Constraint Language) constraints to shape the RDF graphs. We illustrate our proposal by considering US and UNECE regulations and their impact on the window lifter.

Keywords: Compliance Management · Product Evolution · Feature Diagrams · Ontologies · Reusable SHACL Constraints · Variability.

1 Introduction

Compliance management is a challenging activity in (highly) regulated industries. This is due to the need of handling high volumes of data and navigating rapidly shifting requirements. In the automotive domain, for instance, products (items) evolve rapidly creating a product line in time in addition to the one in space. The product variability is constrained by legislations, standards, guidelines, etc. whose applicability may vary depending on the jurisdiction. Hence, the variability to be managed is not only at the technical product level but it also comprises the management of the variability related to potentially applicable and overlapping legislations, standards, guidelines, etc. This implies that activities focused on modelling these different types of features need to be supported. All these pieces of variable information contribute to building the product's pedigree [2]. In [4], authors have elaborated a layered structure where the inter-dependencies of different variable artifacts (applicable regulations, applicable standards, etc. and consequent implication on product evolution) are visualized. Via this layered

* Supported by 4DSafeOps #49 project, https://www.software-center.se

structure, different departments within a highly regulated industry are expected to identify features and characterise what varies and what remains a commonality in their domain of expertise. We recognize that that layered structure is actually present within industrial settings. However, based on experience and as emerged in a recently conducted survey [1], we also recognize that to manage the variability at different levels, there is a lack of suitable modeling approaches that enable a more comprehensive and systematic modeling as well as lack of tools allowing for interoperability. Specifically, authors in [1] foresee a new generation of software variability tools with better visualization capabilities where feature models can be shown not as a whole, but separated into functional areas and easily to be managed. In [6], authors elaborate a set of principles for feature modelling, including the principle "split large models and facilitate consistency with interface models", as done in [5]. Interface models are, however, an ad-hoc solution and not a systematic one. To enable different departments to interoperate while managing the variability at their own level (i.e., to split large models while managing the interfaces), we propose an ontology-based representation for managing product variability and re-configuration at different levels. To do that, first, we provide a feature model metamodel, formalized as an ontology, given in RDF, augmented with SHACL constraints to shape/validate the RDF graphs. We also provide reusable SHACL constraints for characterizing the nature of the features (in terms of their variability type). To illustrate our proposal, we focus our attention on two layers, i.e., the legislations and their impact on the product in terms of inclusion/exclusion of product features. Specifically, we consider US and UNECE regulations and their impact on the window lifter. We show how to represent and validate interrelated feature diagrams representing legislations and window lifters. The rest of the paper is organised as follows. In Section 2, we recall essential background information. In Section 3, we present and illustrate our ontology-based representation. In Section 4, we discuss related work. Finally, in Section 5, we summarise and sketch future work.

2 Background

Feature diagrams are a graphical representation of a hierarchically arranged set of **features**, i.e., system properties that capture commonalities or discriminate among systems in a family. From a semantics perspective, feature diagrams are trees that are composed of nodes and directed edges. Hence, we sometimes call them feature trees. The tree root represents a feature that is progressively decomposed using mandatory, optional, alternative (exclusive-OR features) and OR-features. In feature diagrams, assuming (during configuration) a top-down selection of features, mandatory features are features that are always included in every product. Features that are not necessarily included in every product may vary. Variation points are features that have at least one direct variable sub-feature (i.e. as one of its children). The tree can be constrained by formulating cross-cutting constraints (requires/excludes) that impose presence/absence of

features based on presence/absence of other features. In this paper, we call feature tree variants the feature tree configurations.

Power-operated window lifters (WLs) regulate the movement of a side car window pane. WLs represent a standard feature mounted on every modern car. A WL is always composed of mechanics, electronics, and the connector to receive the power input. The mechanics is composed of a set of components, including a drive gear (mandatory component), composed by a direct current (DC) motor and toothed and worm gears. The DC motor is responsible for converting the electrical energy into mechanical energy, needed to actuate the movement (lifting or lowering) of the window pane. The electronics is also composed of a set of components, including a microcontroller (MC) unit, which, based on the sensed information and on the received input, controls the speed and torque of the DC motor. The MC is composed of hardware and software, which may vary. Different implementations can be selected depending on the to-be-satisfied needs.

WL-related and self-reversal/anti pinch systems-focused legal requirements applicable in USA and EU. Both, the USA and EU legislations, specify the same maximum pressure of 100N in their standards. Hence, this is a mandatory feature. However, in the USA, as part of the Code of Federal Regulations, specifically the part on Federal Motor Vehicle Safety Standards (FMVSS), Section 571.118 (Standard No. 118; Power-operated window, partition, and roof panel systems), it is stated that to detect a pinch situation, tests must be conducted with a specific test rod stiffness of 65N/mm (S8.1.(b)). In Europe, according to the UNECE regulation 21 (R21) [7], the required stiffness for the test rods is 10N/mm (Paragraph 5.8.3.1.3.). These different requirements reveal a variability of type XOR, specifying that UNECE and US legislations are in XOR and that depending on which one is applicable, corresponding requirements (features) have to be satisfied. These different requirements have an implication on the design of the system (i.e., call for different algorithmic solutions at software level), since a stiffer test rod requires faster reaction times as the pressure increases faster.

Resource Description Framework(RDF) [10] allows users to describe resources on the Web in a machine-understandable format. The abstract syntax of RDF has two key data structures: RDF graphs as sets of subject-predicate-object triples, and RDF datasets for organizing several RDF graphs.**Web Ontology Language (OWL)** [8] extends RDFS by adding advanced constructs for defining the semantics of RDF statements. This enables the formulation of constraints related to e.g., cardinality, restrictions of values. The main building blocks of an OWL ontology are classes. A *class* defines a group of individuals that belong together because they share the same properties. Classes can be organised in a specialisation hierarchy using *subClassOf*. **Shapes Constraint Language (SHACL)** [11] is a language for defining constraints, called "SHACL shapes", against which RDF graphs can be validated. SHACL constraints can be interpreted and processed by a SHACL engine, which checks and validates the SHACL constraints. In case there are facts in an ontology model or knowledge graph (KG) which violate any SHACL constraints, the SHACL engine reports a constraint

violation, along with a description. This is then shown to the user. SHACL has its own vocabulary (e.g., a shape is specified with the construct sh:shape), but it uses RDF and RDFS vocabulary to define types, classes, subclasses, properties, lists and resources. **SPARQL** [**9**] stands for SPARQL Protocol and RDF Query Language. SPARQL can be used to formulate queries across diverse data sources (which can be stored natively as RDF graphs).

3 Ontology-based Representation

In this section, we present our three contributions. **Contribution-1** consists of a feature tree metamodel, formalised as ontology. The metamodel allows for modeling individual feature trees of any domain and of arbitrary size. The feature tree metamodel is defined as UML (Unified Modeling Language) class diagram, shown in Fig. 1. It comprises two classes and five relationships in total.

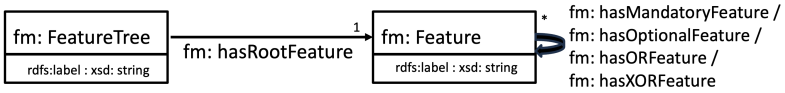

Fig. 1. Feature tree metamodel

`fm:FeatureTree` is the class that represents feature trees. Each feature tree has exactly one root node, which is defined via the relationship `fm:hasRootFeature`. The feature it refers to is the root feature, i.e. the top feature in the feature tree. Each feature can comprise several sub-features, which can be either mandatory features (relationship `fm:hasMandatoryFeature`), optional features (relationship `fm:hasOptionalFeature`), alternative features (relationship `fm:hasORFeature`) or exclusive features (relationship `fm:hasXORFeature`). With this feature tree metamodel, arbitrary feature trees can be expressed. This metamodel can be easily formalized as OWL ontology, by defining the classes as OWL class, and the relationships as OWL object property. Fig. 2 shows an excerpt of the feature tree ontology in RDF syntax.

```
fm:FeatureTree
 a owl:Class ;
 rdfs:label "Feature Tree" ;
.

fm:Feature
 a owl:Class ;
 rdfs:label "Feature" ;
.

fm:hasRootFeature
 a owl:ObjectProperty ;
 rdfs:label "has root feature" ;
 rdfs:domain fm:FeatureTree ;
 rdfs:range fm:Feature ;
.

fm:hasMandatoryFeature
 a owl:ObjectProperty ;
 rdfs:label "has mandatory feature" ;
 rdfs:domain fm:Feature ;
 rdfs:range fm:Feature ;
.
```

Fig. 2. Excerpt of the feature tree ontology in RDF syntax.

The metamodel ontology constitutes a reusable upper ontology, which can be imported and refined by a dedicated feature ontology, specialized for a certain

domain and use case. Therein, the specific feature classes and relationships of the domain and use case can be defined. To link it with the metamodel, the feature classes can be defined as subclasses of class `fm:Feature`, and the relationships can be defined as subproperties of the object property `fm:hasMandatoryFeature`, `fm:hasOptionalFeature`, `fm:hasORFeature` or `fm:hasXORFeature`. This is shown for the class `sa:LegalRequirements` and the object property `sa:comprises Jurisdiction` in Fig. 3.

```
sa:LegalRequirements
  a owl:Class ;
  rdfs:label "Legal Requirements" ;
  rdfs:subClassOf fm:Feature ;
  sh:rule
  co:SPARQLRule_AutoCreateMandatoryFeatures ;
  .

sa:comprisesJurisdiction
  a owl:ObjectProperty ;
  rdfs:subPropertyOf fm:hasMandatoryFeature ;
  .
```

Fig. 3. LegalRequirements Class.

A feature ontology representing and organizing the features of the regulations is shown in Fig. 4 as UML class diagram. This diagram translates the information, which was provided in Subsection 2. The depicted classes represent OWL classes, and the associations between them OWL object properties defined in the feature ontology. The superclass of each class, and the superproperty of each object property from the metamodel are shown as UML stereotypes using the "`<<...>>`" notation. This feature ontology will be used in the following for introducing and demonstrating the reusable SHACL constraints.

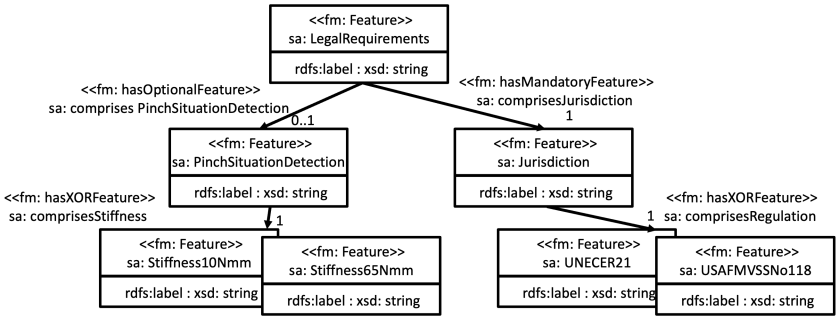

Fig. 4. A feature ontology representing the overlapping legislations.

A feature ontology representing and organizing the features of the set of window lifters is shown in Fig. 5 as UML class diagram.

Contribution-2 consists of the validation of Feature Tree Variants with SHACL. Feature tree variants are to be modeled via instances of the specific feature ontology and relationships between them. Instances of a specific `fm:Feature`

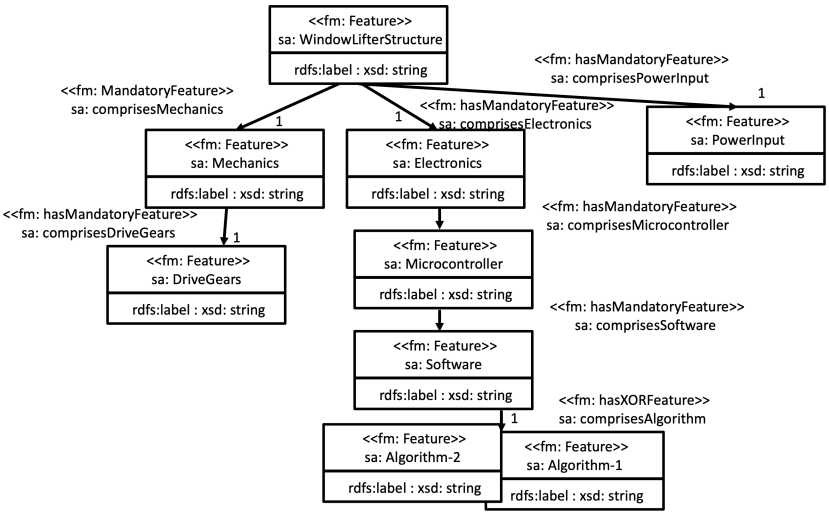

Fig. 5. A feature ontology representing a set of window lifters.

subclass represent a chosen feature. The relationships defined in the feature ontology are to be used for inter-relating the instances, so that the instance model represents a valid instantiation of the feature tree. The following RDF snippet demonstrates that on the example of a feature instance `sd:LegalRequirements_1` and its two relationships to (sub-)feature instances:

```
sd:LegalRequirements_1
  a sa:LegalRequirements ;
  sa:comprisesJurisdiction sd:Jurisdiction_1 ;
  sa:comprisesPinchSituationDetection sd:PinchSituationDetection_1:
.
```

For validating a feature tree variant, we propose to use SHACL, which means that a feature ontology needs to be augmented with several SHACL statements. To realise that, the OWL classes in an ontology need to be extended to SHACL node shapes, and be related to the SHACL property shapes. This is shown in the following example for the class `sa:LegalRequirements` in RDF:

```
sa:LegalRequirements
  a owl:Class , a sh:NodeShape ;
  rdfs:label "Legal Requirements" ;
  rdfs:subClassOf fm:Feature ;
  sh:property sa:LegalRequirements-comprisesJurisdiction ;
  sh:property sa:LegalRequirements-comprisesMaxSqueezingForce ;
  sh:property sa:LegalRequirements-comprisesPinchSituationDetection ;
  sh:sparql co:SPARQLConstraint_RequiresDependency ;
.
```

In this example, `sa:LegalRequirements` is declared to be a SHACL node shape via the statement "`sa:LegalRequirements a sh:NodeShape`", besides being a `owl:Class`. Furthermore, via the SHACL property `sh:property`, several SHACL property shapes are attached to it. Each SHACL property shape is specific for one relationship of the class. To a SHACL property shape, certain constraints can be attached, as shown in what follows.

Contribution-3 consists of reusable SHACL constraints for mandatory, optional, OR, XOR, and requires feature relations, as explained in the following using examples, taken from the feature ontology shown in Fig. 4. The constraints are defined on the level of SHACL property shapes with SHACL core constraint components and are specifically designed to support a top-down based creation of feature tree variants. This means that a feature tree configuration process starts with the instantiation of the root feature, followed by the features on the second, third, fourth level and so on. At each (intermediate) step of this top-down process, the SHACL constraints are capable of validating the correctness and completeness of the current feature tree variant.

SHACL Constraint Pattern for Mandatory Features. A mandatory feature can be realized by a SHACL property shape that constraints the cardinality of the object property to have exactly one value, accomplished with the SHACL cardinality constraint components `minCount=1` and `maxCount=1`. This can be seen in the following RDF snippet:

```
sa:LegalRequirements-comprisesJurisdiction
  a sh:PropertyShape ;
  sh:path sa:comprisesJurisdiction ;
  sh:class sa:Jurisdiction ;
  sh:name "comprises jurisdiction" ;
  sh:maxCount 1 ;
  sh:minCount 1 ;
.
```

This SHACL property shape requires each instance of the class `sa:Legal Requirements` to have exactly one `sa:comprisesJurisdiction` relation to an instance of class `sa:Jurisdiction`, which represents the mandatory feature.

SHACL Constraint Pattern for Optional Features. An optional feature can be realized by a SHACL property shape that constraints the cardinality of the object property to zero or one value, accomplished with the SHACL cardinality constraint component `maxCount=1`. The absence of a `sh:minCount cardinality` constraint component makes the relationship optional. This can be seen in Fig. 6.

```
sa:LegalRequirements-comprisesPinchSituationDetection   sh:class sa:PinchSituationDetection ;
  a sh:PropertyShape ;                                  sh:name "comprises pinch situation detection" ;
  sh:path sa:comprisesPinchSituationDetection ;         sh:maxCount 1 ;
                                                        .
```

Fig. 6. RDF snippet representing the optional feature.

This SHACL property shape defines that each instance of the class `sa:Legal Requirements` may have one (optional) `sa:comprisesPinchSituationDetection` relation to an instance of class `sa:PinchSituationDetection`, which represents the optional feature.

SHACL Constraint Pattern for XOR Features. An "exclusive or" feature (XOR) can be realized by a SHACL property shape that constraints the cardinality of the object property to have exactly one value, accomplished with the SHACL cardinality constraint components `minCount=1` and `maxCount=1`, and that defines the XOR features via SHACL `sh:or` as alternative range classes. This SHACL property shape requires each instance of class `sa:PinchSituation Detection` to have exactly one `sa:comprisesStiffness` relation to an instance of either class `sa:Stiffness10N` or `sa:Stiffness65N`, which represent the XOR features. This can be seen in Fig. 7.

```
sa:PinchSituationDetection-comprisesStiffness
  a sh:PropertyShape ;
  sh:path sa:comprisesStiffness ;
  sh:maxCount 1 ;
  sh:minCount 1 ;
  sh:name "comprises stiffness" ;
  sh:nodeKind sh:IRI ;
  sh:or ([sh:class sa:Stiffness10N;]
         [sh:class sa:Stiffness65N;]);
  .
```

Fig. 7. RDF snippet representing the XOR feature.

SHACL Constraint Pattern for OR Features. Alternative features (OR) can be realized by a SHACL property shape that constraints the cardinality of the object property to one or more values, accomplished with the SHACL cardinality constraint component `minCount=1`, and that defines the OR features via SHACL `sh:or` as alternative range classes.

SHACL Constraints for Requires. Constraints that express comprehensive interdependencies between different features can be modelled in SHACL as SPARQL-based constraints. We propose to specify a "requires" relationship between one feature of one feature model, and another feature of another feature model as SPARQL-based constraint. This constraint detects requires dependencies between features of different feature models. It requires a relationship of type `fm:requiresFeature` to be defined in the feature ontology between the requiring class (subject) and the required class (object). As an example, we consider that this relationship exists between the feature `sa:USAFMVSSNo118` in Fig. 4 and the feature `sa:Algorithm-1` in Fig. 5. If the feature `sa:USAFMVSSNo118` is selected by the legal requirements configuration, `sa:Algorithm-1` is required in the window lifter configuration. The constraint detects violations of specific requires dependencies, i.e. two features where one requires the other and that not both appear at the same time in different feature models. The requires constraint is generic and reusable, i.e. it can be associated with any root feature class of a feature model, and does not require adaptations when doing so. The requires constraint, realized as SPARQL-based constraint, is shown in the following:

`co:SPARQLConstraint_RequiresDependency`

```
  a sh:SPARQLConstraint ;
  sh:message "The feature {?instanceRequiringFeatureClass} of type
  {?requiringFeatureClass} requires the feature {?featureClass}" ;
  sh:select """PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT $this ?requiringFeatureClass ?instanceRequiringFeatureClass
    ?featureClass ?instanceFeatureClass
WHERE { $this a ?rootFeatureClass.
  ?rootFeatureClass ((sh:property/sh:class)|
      (sh:property/sh:or/rdf:rest*/rdf:first/sh:class))+
      ?featureClass.
  ?requiringFeatureClass fm:requiresFeature ?featureClass.
  ?instanceRequiringFeatureClass a ?requiringFeatureClass.
  FILTER NOT EXISTS { ?instanceFeatureClass a ?featureClass} }""";
.
```

The functional principle of the constraint is the following: The constraint iterates over all instances of the root feature class it is associated with (variable `$this`). From that feature class it recursively explores the entire feature tree to all contained feature classes. That is realized by a comprehensive recursive SPARQL path (operator "`+`") comprising two alternative subpaths (operator "|"). The first alternative subpath covers mandatory and optional relationships, which are defined in SHACL via `sh:class` property. The second subpath covers OR and XOR relationships, which make use of the SHACL `sh:or`. Here a comprehensive iterating through the RDF list comprising the `sh:or`-related classes is required via `rdf:rest` and `rdf:first` properties. For each feature class of the feature tree found via the SPARQL path, the existence of an incoming `fm:requiresFeature` relation from another feature class is checked. Then, it is checked if the requiring class has an instance, but the required class not. If so, this means that the requires constraint is violated, and the requiring instance, along with its class, is reported back to the user. The SHACL engine therefore constructs (and reports about) a constraint violation message from the string defined in `sh:message`. At runtime the SHACL engine inserts data from the KG via SPARQL variables into the placeholders, e.g. `{?featureClass}`. The respective variable binding when executing the SPARQL-based constraint will be inserted into the message.

4 Related work

In [12], authors propose an ontology to formally specify feature models. Their ontology, however, is mainly constituted of a set of OWL classes with limited expressiveness. Despite the intention of proposing an OWL representation for the knowledge contained in feature models, representing all possible feature model semantics and despite the intention of formulating SWRL (Semantic Web Rule Language) rules to ensure the consistency of the feature model and detect contradicting or conflicting knowledge in the model, the expressiveness

is rather limited and allows limited checks. In addition, rules are not conceived in an explicit reusable manner. In [3], the author proposes a mapping between feature trees (given in a domain-specific language) and RDF graphs enriched with SHACL constraints. The SHACL constraints formulated in his work are defined specifically for instances of features, i.e. the instance IRIs and literal values appear in the constraints. This makes the constraints very dependent on the respective instances, hence the potential for reusing them is low. In our work, instead, we define constraints on the level of classes. Our constraints are highly reusable. In addition, in his work, no general feature tree ontology is specified. In our work, instead, we propose one.

5 Conclusion and Future Work

To tackle the problem of compliance management in regulated industries characterized by demanding variability management needs, we have proposed an ontology-based representation for representing feature trees as RDF graphs and reusable SHACL constraints to shape the RDF graphs. In future, we aim at conducting case study-based research considering the entire layered structure [4].

References

1. Allian, A.P., OliveiraJr, E., Capilla, R., Nakagawa, E.Y.: Have Variability Tools Fulfilled the Needs of the Software Industry? Journal of Universal Computer Science **26**(10), 1282–1311 (2020)
2. Capilla, R., Gallina, B., Cetina, C., Favaro, J.: Opportunities for software reuse in an uncertain world: From past to emerging trends. Journal of Software Evolution and Process **31**(8), 1–22 (August 2019)
3. Fleischhacker, P.: Validation of Feature Models using Semantic Web technologies. Master's thesis, Graz University of Technology, Graz, Austria (2021)
4. Gallina, B., Munk, P., Schweizer, M.: An Extension of the Rasmussen Socio-technical System for Continuous Safety Assurance. In: Roy, M. (ed.) CARS 2024 - Critical Automotive applications: Robustness & Safety. Leuven, Belgium (2024)
5. Hofman, P., Stenzel, T., Pohley, T., Kircher, M., Bermann, A.: Domain specific feature modeling for software product lines. In: Proc. of the 16th Int'l Software Product Line Conference (SPLC)- Volume 1. p. 229–238. Association for Computing Machinery, New York, NY, USA (2012)
6. Nešić, D., Krüger, J., Stănciulescu, u., Berger, T.: Principles of feature modeling. In: Proc. of the 27th ACM Joint Meeting on ESEC/FSE. p. 62–73. New York, NY, USA (2019)
7. UNECE: *REGULATION 21 - Uniform Provisions Concerning the Approval of vehicles with regard to their interior fittings* (April 2003)
8. W3C: OWL 2 Web Ontology Language Manchester Syntax (2012)
9. W3C: SPARQL 1.1 Query Language (2013)
10. W3C: RDF 1.2 Concepts and Abstract Syntax (2014)
11. W3C: Shapes Constraint Language (SHACL), W3C Recommendation (2017)
12. Zaid, L.A., Kleinermann, F., De Troyer, O.: Applying semantic web technology to feature modeling. In: Proc. of the ACM Symposium on Applied Computing (SAC). p. 1252–1256. Association for Computing Machinery, New York, NY, USA (2009)