Philipp Fleischhacker, BSc

Validation of Feature Models using Semantic Web technologies

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach

Institut für Technische Informatik
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Graz, April 2021

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

Abstract

With the introduction of Industry 4.0, a push for automation started in manufacturing. Companies need to adapt to this shift, which influences the way new products are developed. Often, products share similar traits to other products. To reduce development time, companies have started to reuse components of previous products to create new ones. In order to do so, these products need to be designed in a way to support reuse.

The field of Software Product Lines (SPL) aims to increase the reuse of software components. One of the most used tools of SPL are feature models. A feature model is a hierarchical model of all components in a product family, and the dependencies between them. Valid combinations of features from the model form a product.

The Semantic Web has been gaining popularity both academically and in industry. With the recent introduction of the Shapes Constraint Language (SHACL) standard for validating Resource Description Frame (RDF) models, we want to investigate how well suited it is for feature modelling.

In this thesis, we build a feature model system using the Semantic Web technologies RDF and SHACL. This system allows a user to configure a product, and validate whether the configuration is a valid product. As RDF is a graph based way to structure data, we directly map a feature tree to an RDF model. We model the constraints of the feature model as SHACL shapes. SHACL allows us to specify advanced constraints, such as attribute constraints over aggregates of attributes, in addition to the standard constraints of feature models (require, exclude, cardinality). We use the features built into SHACL to describe complex constraints on the feature model. Using Semantic Web technologies has the benefit making the system platform independent. To facilitate the definition of these constraints, a simple textual Domain Specific Language (DSL) was created. Writing RDF by hand is cumbersome, so this gives users a way to define constraints in a concise and readable way.

Testing shows that the system performs well, even with large models consisting of only boolean constraints. When using models containing a large amount of advanced constraints, the system still performs well with medium size models of about 500 features and an equal number of constraints.

Contents

1 Introduction

For the last decade, industrial manufacturing has been undergoing a revolution, called ”Industry 4.0” [1]. Industry 4.0 has spurred the development of new manufacturing techniques caused by rapid changes and increasing competitiveness in the manufacturing market, as well as changes of customer demands.

Companies are required to adapt to this change of markets in order to stay competitive [2]. This causes a paradigm shift in how products are developed. Different products often share similar components among them. Companies exploit these similarities in order reuse components of older products to develop new ones. This requires products to be developed in a way to enable this reuse with the least effort possible.

A software product line is defined as “a set of products that share a common core technology and address a related set of market applications”[3]. Software Product Lines (SPL) are the methods and technologies used to create product lines. Fields like SPL continue to gain importance. While SPL often concerns itself with modelling software products, the underlying principles can be applied to any type of product.

One of the tools from SPL are feature models. Feature models are hierarchical models of all the components of a product family. They can be used to model the variability of any product. Using feature models, it is possible to build a structure to model the hierarchy of components of a product family, and model dependencies between these components. By combining components that do not violate constraints, new products can be derived [4]. By being able to reuse components in this way, it is possible to develop new products that cater to a customer's need more efficiently.

In this thesis, we develop a new variant management system, that is able to handle complex constraints between components. In addition to building a feature model system, we will, we will also develop a Domain Specific Language (DSL) to define constraints for this feature model in a user friendly manner. To do so, we will first look at the broader background of feature modelling. We will look at the different ways how feature models and systems that use them are built. We will build our feature modelling system using Semantic Web technologies. Using the Semantic Web gives us a platform independent way to both structure and validate data. In particular, we will use the Resource Description Framework (RDF) [5] and the Shapes Constraint Language (SHACL) [6].

RDF describes a way to structure data, while SHACL describes a way to validate this data according to defined rules. We will take a more detailed look at these standards in Chapter 2. The Semantic Web has been gaining a lot of interest in both industry and academia due to its applicability in a multitude of different fields.

One major application of the Semantic Web is to describe data on the internet. Google uses the Semantic Web, in particular RDF to improve search results [7]. The way this works is that google will look for structured data that the website owner defined on the site. The data has to adhere to guidelines for google to understand it. It can for example be a JSON-LD object with properties such as *datePublished* or *rating*. Adding semantic data to a website not only improves search results, but also enables special search features, like embedding the content in a certain way in search results.

Apart from describing meta-data on the internet, RDF has found numerous other applications. One such application is in digital libraries [8]. Digital libary data lends itself well to the ways semantic web data is structured, as often information can be uniquely identified and we often want to categorize it by certain properties. Using Semantic Web technologies was found to be a good way to make data more searchable on the web.

Semantic Web technologies also increasingly find usage in architecture, engineering and construction [9]. In these fields Semantic Web technologies are often used complementary to existing software. The use of Semantic Web tools is motivated by the desire to improve interoperability between software tools and the desire to connect multiple domains of applications.

The Semantic Web has also found applications in biology. DisGenNET-RDF [10] has provided knowledge about the genetic basis of diseases in the RDF format. The information is linked to other biomedical databases to support development and research in bioinformatics.

Life science researchers use semantically annotated data to create models more efficiently [11]. The semantic web allows the reusability and interoperability of models across different model repositories.

1.1 Structure of the Thesis

Chapter 2 gives an overview of feature models and the Semantic Web. We will introduce the history and basic concepts of variant management and software product lines. We will also state the goals of the thesis.

In Chapter 3 we will give an overview of works related to this thesis. We will give an in depth look into different implementations of feature models, and what their supported features are. We will also look at different types of variant management systems that have been implemented.

Having summarized the current state of the art, in Chapter 4, we will introduce the design for the system we will build. This includes the feature requirements for the system and an example of how the features will be used.

Chapter 5 describes the implementation details of the system. We give an overview of the architecture of the system and explain which technologies were used for the implementation and why.

Chapter 6 gives an overview of how the system can be used from a user's perspective. We give an overview of how to define a feature model and how to formulate constraints for the feature model.

In Chapter 7 we will give an evaluation on the system performance. The evaluation is split into an evaluation of the performance of the basic functionality of a feature model and an evaluation of the advanced features.

Finally, Chapter 8 will give an outlook on what could be improved in the future.

2 Background and Motivation

In this chapter we will explain the background and motivation for this thesis. We are first going to give an overview over how feature models are structured, then we will give an introduction to the Semantic Web, and finally state the motivation for this thesis.

2.1 Feature Modelling

One way efficiency in product development can be improved, is by reusing components of other products. SPL is a field that aims to do just that.

One of the most popular approaches to modelling variability of systems that developed from SPL is Feature Modelling. Feature models are models that contain all the components of a system and describe the relationships between them. Feature models are built as a tree with a parent-child relationship between features. According to [3], a general definition of feature models is: "A feature model is a description of the commonalities and differences between the individual software systems in a software product family". There are many different kinds of feature models with different use cases and features.

2.1.1 Structure of Feature Models

Figure 2.1 shows a graphical representation of a simple feature model for a mobile phone.

As can be seen, features are structured in a hierarchical way, forming a tree. For example, the feature *Media* has the child features *Camera* and *MP3*. In the

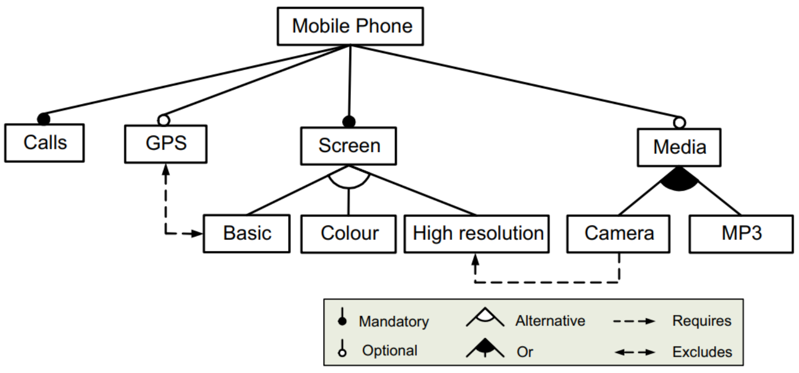


Figure 2.1: An example for a feature model tree, containing a feature hierarchy for a mobile phone [12]

figure we can see all the types of relationships that are are usually supported by feature models. There are multiple types of hierarchical relationships. According to [13], the basic set of relationships are: Mandatory, Optional, Alternative, and Or. Mandatory and optional denote when a child feature has to be included, while Alternative and Or constraint groups of child features.

Cross Tree Constraints

Given the basic relationships, we can model constraints between parent features and direct child features. Additionally, it is often needed to also model constraints between features that are not in a direct relationship. We call these types of constraints cross-tree-constraints. They are used to require or exclude a feature, based on the selection of another feature.

An example for a cross tree constraint can be seen in Figure 2.1. The dotted line between the features *Camera* and *High Resolution* represents a requirement constraint. It means that if *Camera* is included then *High Resolution* must also be selected.

Attributed Feature Models

With the relationships described above we can create rules that constrain which features can or must be selected. However, we can only make rules that restrict the selection of a feature based on the selection of another feature. In order to increase the expressiveness of the feature model, many propositions include attributes to more closely describe the features. [14, 15, 16]. Feature models that support the use of attributes are called attributed feature models [12].

While there is currently no standardized way how attributes are supposed to be defined, they usually include a name, a value and a domain for that value. The use of attributes allows more powerful constraints to be defined. It allows for features to be restricted or required based on an attribute value they have. For example, consider a feature having the attribute *PowerConsumption*. We can define a constraint saying that if a specific feature is included, then only features with a *PowerConsumption* of less than a certain threshold are allowed to be selected.

Logical Operators

Many feature models have support for building logical formulas for constraints. This allows the combination of selections using logical operators. This greatly increases the expressiveness of constraints.

```
if A && B then restrict C
```

These rules can often not feasibly included in the feature tree itself. Therefore, there is often an extra way to specify these constraints.

2.2 Semantic Web

As mentioned in Chapter 1, we will use Semantic Web technologies as the core for our feature model due to its applicability to feature modelling and its platform independence. This section is meant to explain the background to the Semantic Web.

The idea for the Semantic Web was first described in 2001, with the goal of making data on the internet machine readable [17]. For this purpose, several specifications were developed to represent semantic data and to query this data. One of the specifications is the Resource Description Framework (RDF) [5]. RDF is a format for storing data. Another format to store semantic data is the Web Ontology Language (OWL) [18]. While support for Semantic Web meta data on the internet remains somewhat low (in 2013 there were 3 million domains containing semantic markup [19]), there are useful applications in other domains such as library science ([8]) and biology ([11]), [10]). We will describe the technologies and some of their applications in more detail.

2.2.1 Resource Description Framework - RDF

RDF is a specification for representing semantic data from the World Wide Web Consortium (W3C) [5]. It was first specified in 1997 and remains one of the most common ways to represent data for the Semantic Web. RDF allows the encoding and reuse of metadata. The language itself makes no assumptions on the semantics of the metadata. Every resource in an RDF model is described by an International Resource Identifier (IRI).

While originally RDF was an XML language [20], there are many different formats in which RDF graphs can be saved. Two common ones are the Turtle format, and JSON-LD. Turtle stores triples by seperating the IRI of the resources by a space and ending each triple with a period. It also groups triples by resources so that each subject only has to be stored once. JSON-LD stores triples in the JSON format. Each JSON object has a subject IRI and key value pairs for each property of the subject.

The IRI for a resource can be shortened by using namespace prefixes. It is possible to define a prefix (e.g. *ex:*) which replaces the domain part of the IRI. For example say we have the resource *http://www.example.org/myresource*. With a namespace prefix we can define *ex:* to be *http://www.example.org/*. This shortens it to just *ex:myresource*. Below is an example for an RDF model

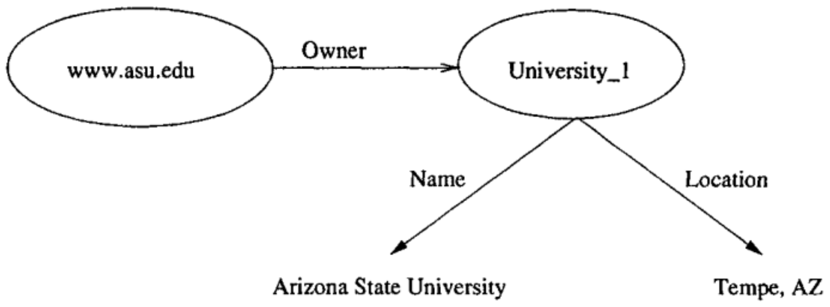


Figure 2.2: An example graph representation of an RDF model [5]

in the turtle format. The model consists of a university resource that has a Name and location property, and another resource that has an owner property pointing to the university.

```
<www.asu.edu> ex:Owner ex:University_1.
ex:University1 ex:Name "Arizona State University";
               ex:Location "Tempe, AZ".
```

Resources can have properties with their own IRI. These properties point to values, which can be other Resources in the model. We can view RDF models as a graph, which is why they are also called RDF graphs. The graph consists of the resources as nodes and properties as edges between nodes. Figure 2.2 shows the example described above in a graph representation. Since each identifier for a resource should be unique, properties can also refer to resources that are not part of the same RDF graph. Each of these structures of resource-property-value is called a triple, consisting of a subject a predicate and an object. Because triples can have resources as objects, which means that resources can be connected to other resources via multiple property paths.

There are numerous applications that support the RDF specification and offer functionality for creating and querying models. Examples are Apache Jena, an implementation as a Java library, DotnetRDF, an open source implementation for C# and TopBraid Composer by TopQuadrant.

RDF Schema (RDFS) [21] is a specification that provides semantic extensions to RDF. It allows resources to be grouped together and more closely described via a group of properties. RDFS allows RDF data to be structured in an object oriented way. To do so it provides properties such as *rdf:class* to determine that a resource is a class. Accordingly, there is also an *rdf:type* property to declare that a resource is an instance of a class. Class hierarchies are possible via the *rdfs:subclassOf* property.

2.2.2 Ontology Web Language - OWL

OWL is another way to represent data in the Semantic Web. It's a declarative language that aims to make it possible to reason about the model.

OWL is a language of the Semantic Web stack [18]. The language is designed to represent complex knowledge about things, and relationships between things. Being a Semantic Web language, the goal is to represent data in a way such that it can be reasoned about by software. OWL models are called ontologies. It defines a set of precise statements about a part of the world (the domain of interest). It is necessary to have precise descriptions in order to be able to reason about these statements. In order to precisely describe the domain, OWL documents usually contain a vocabulary of terms with defined meanings.

OWL is a declarative language, describing the domain in a logical way. This means tools called reasoners can be used to infer new information from the ontology. OWL follows the open world assumption. Usually if a fact is missing in a data base it is assumed to be false. This is called the closed world assumption. In the case of OWL it is assumed that the fact might simply be missing, but still true, the open world assumption.

OWL has been long established and is supported by many tools.

2.2.3 SPARQL

RDF and OWL only define how the data is represented. To work with it we need other tools. SPARQL is a query language for querying RDF graphs

[22]. The syntax of SPARQL is similar to that of SQL. There are different types of SPARQL queries:

- *SELECT* queries select nodes according to the conditions specified in the query
- *ASK* queries return true if a result can be found using the specified query or false otherwise
- *CONSTRUCT* queries can be used to insert new triples into the graph

Below is a simple example for a SPARQL *SELECT* query. The examples are from the SPARQL documentation [22].

Data graph:

```
ex:book1 ex:title "SPARQL Tutorial" .
```

Query:

```
SELECT ?title
WHERE
{
  ex:book1 ex:title ?title .
}
```

This query will return a set of results corresponding to the graph pattern in the query body. The body selects all triples in the graph that have *ex:book1* as subject and *ex:title* as a predicate. The objects of these triples get bound to the variable *?title*. the results that are bound to the variables in the *SELECT* clause form the result set for the query. In this example the nodes bound to the *?title* variable will be returned. Because the triple in the data graph matches the pattern in the query, the result for this query is simply:

```
    title: "SPARQL Tutorial"
```

Looking at a more advanced example:

```
_:a  foaf:name   "Johnny Lee Outlaw" .
_:a  foaf:mbox   <mailto:jlow@example.com> .
_:b  foaf:name   "Peter Goodguy" .
_:b  foaf:mbox   <mailto:peter@example.org> .
_:c  foaf:mbox   <mailto:carol@example.org> .
```

```
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

Here the result is:

```
name: "Johnny Lee Outlaw" mbox: <mailto:jlow@example.com>
name: "Peter Goodguy" mbox: <mailto:peter@example.org>
```

Each result represents one way the variables can be bound so that they match the data. In this example if *?x* is bound to *_:a* then *?name* must be bound to *"Johnny Lee Outlaw"* and *?mbox* must be bound to *<mailto:jlow@example.com>*.

SPARQL supports several more advanced features. One of the features is the ability to nest queries. This means a *SELECT* query can be specified in the body of another query, and results of that query can be used like any other bound variable.

Another feature is the *FILTER* keyword. The keyword allows filtering of the results for a triple pattern.

```
SELECT  ?title ?price
WHERE   { ?x ns:price ?price .
          FILTER (?price < 30.5)
          ?x dc:title ?title . }
```

The example above will filter out any match for the triple *?x ns:price ?price* where the binding for the variable *?price* is ≥ 30.5.

We can group the results of the query by one of the returned bindings. This allows us to return an aggregate over all bindings for one variable.

```
SELECT  ?title (SUM(?price) as ?sum)
WHERE   { ?x ns:price ?price .
          ?x dc:title ?title . }
GROUP BY ?title
HAVING (SUM(?price) < 500)
```

2.2.4 Shapes Constraint Language - SHACL

Data validation is an important use case of the Semantic Web. However, there was no standardized way for it until the SPARQL Inference Notation (SPIN) [23] was introduced. SPIN is a language with the purpose of inferring knowledge for RDF models and OWL ontologies. It is also possible to define constraints, which makes it possible to use for data validation. Data validation was however not it's main purpose.

With the Shapes Constraint Language (SHACL) [6], the Semantic Web received its first specification with data validation as its main purpose. Specifically, SHACL is a specification to validate RDF graphs. The rules for validating RDF graphs are part of another valid RDF graph themselves. The first version of the specification was released in 2017 by the W3C making it a relatively new standard. SHACL provides a wide array of properties to validate RDF graphs. SHACL rules are defined as so called shapes which are RDF nodes with properties that define which nodes are affected and SHACL properties that these nodes must adhere to. We call the graph that contains the model we want to validate the data graph and the graph that has the SHACL rules the shapes graph.

The resources that are validated by a shape are defined by the target node properties in the shape. `sh:targetNode` is the simplest node target property. It is used to specify a specific node of the data graph as the target for the shape. Another way to define target nodes is the `sh:targetClass` property which takes all instances of a target class as target for the shape. There are some more ways to define the targets for a shape. For example, it is possible to use all nodes that are the subject of specific triples as target nodes.

Shapes that define target nodes are called node shapes. During the validation process, every node shape in the shapes graph is evaluated. We can also define shapes that can be referred to by node shapes, so called property shapes. Property shapes are shapes that have a path property that specifies a path from the focus node to a property. A property shape also has an arbitrary number of SHACL properties that are validated against the nodes found via the property path. A property shape is referenced in a node shape by using the SHACL property `sh:property` These property shapes are only evaluated when a node shape is referring to them. The limitation for property shapes is that they cannot define node targets themselves, and they cannot be nested.

SHACL shapes can use SPARQL queries using the `sh:sparql` predicate, for validation purposes with some restrictions. It is possible to specify either an *ASK* or a *SELECT* query. For select queries, the focus node that is currently evaluated is pre-bound to a variable called `$this` and has to be the first value returned by the query. Any returned set of notes produces a rule violation in the report. *ASK* queries produce a violation if the query evaluates to *false*.

The SHACL validator will go through all node shapes in the shapes graph in order. For every node shape the validator first produces a set of nodes using the node target predicates that are defined. It then tries to validate all rules defined on the shape for every target node. If any of the rules failes for a node, then a violation is produced and added to a report which is returned after validation is finished. The report contains which node failed, which rule produced the failure and a message explaining it.

Below is an example for a shape together with a corresponding data graph. The shape validates that the given nodes have an *ex:age* property of type integer.

```
Data:

ex:Alice ex:age 25 .
ex:Bob ex:age "seventeen" .

Shapes:

ex:ExampleShape
a sh:NodeShape ;
sh:targetNode ex:Alice, ex:Bob ;
sh:property [
sh:path ex:age ;
sh:datatype xsd:integer ;
] .
```

When validating, SHACL will report a violation for ex:Bob, because his age property is of type string.

SHACL has been implemented in most major RDF implementations including DotnetRDF and Apache Jena, making it a tool that can be used on most platforms.

2.3 Domain Specific Languages

A Domain Specific Language (DSL) is a language tailored for a specific application domain [24]. As such a DSL only includes constructs and features needed for its respective domain. The goal in making a DSL is to provide a high level abstraction to create programs within a domain without the need for general programming knowledge. A DSL has a formally defined syntax. The interface to the language can be either textual or graphical.

There are several different types of approaches to implement a DSL. Common approaches are:

Embedding, where the where mechanisms in a host language are used to express the domain specific requirements. The DSL inherits the host language syntax and adds constructs to express the requirements of the application domain. *Compilers*, where the DSL gets compiled to constructs in the general programming language. *Compiler generators*, which is similar to compiled languages, but uses language development tools to automate some of the compilation stages. Using a generator reduces the effort of developing the language.

We will now take a look at one of these language development tools.

2.3.1 ANTLR

ANTLR (Another Tool for Language Recognition) is an automatic parser generator [25]. It is implemented as a library for Java and C#. ANTLR can be used for reading, processing or translating structured text. It is a widely used library for creating languages. ANTLR generates a parse tree from a given input text, using a user defined grammar.

When working with ANTLR you generally define two things:

- A grammar file that defines the rules for grammar you want to parse.
- A Visitor or Listener class that is used to actually parse text.

Grammar Structure

The rules in ANTLR grammar files are split into two parts. The first part are lexer rules. Lexer rules are used for the first step of parsing, the lexicographic analysis. They are used by taking the text as input and transforming it into tokens. This token stream is then passed as input to the parser. ANTLR automatically generates a lexer and a parser from the rules.

The rules for both parser and lexer are in the format:

```
<rulename> : <rulebody>;
```

Whether a rule is a lexer or a parser rule is determined by the first letter. Lexer rules start with a capital letter, parser rules start with a lower case letter. Lexer rules generally define how each token is structured. They define if something is considered an identifier, a general text, a digit, keywords for the language and the like. Parser rules define higher level constructs of the language, like how a function is structured, how a statement looks like, expressions etc.

```
intDeclaration : INT IDENTIFIER ';';
INT : 'int';
IDENTIFIER : [a-zA-Z]+;
```

Rules can use *zero-or-more*, *alternative*, *one-or-more operators* like it is possible in regular expressions. For example a function body may be zero or more statements inside. An assignment may have either a literal value or an identifier on the right hand side.

```
function : functionheader functionBody;
functionBody : (statement | assignment)*;
```

The parser uses the token stream from the lexer together with the parser rules, defined in the grammar file to build a parse tree. It does so by matching tokens from the token stream to rules from the grammar. The parser returns the root node of the parse tree.

ANTLR automatically generates a base visitor or listener. The methods for visiting can be overridden in a derived class. The visitor has a visit method for each parser rule in the grammar. Usually after parsing, the root node of the parse tree will get passed to the visit method of the visitor. This is the entry point for visiting the parse tree. The passed node has it's child rules as methods returning a context object, which can be used to visit those rules. Each of the methods returns 'object' as type, so it can be used to propagate results from visiting back up.

2.4 Motivation

Approaches using OWL to model feature trees have been becoming more frequent recently [26] [27]. Although SHACL and OWL are very different standards there are some commonalities with how they are used. OWL was made for reasoning over ontologies, but with SPIN it was made possible to also do model validation. SHACL is a relatively new standard, that has been gaining a lot of interest. Being a new standard, many of its possibilities are not yet tested. We want investigate how well suited SHACL is for validating configurations, and in particular feature modelling.

Since SHACL has a large number of features, including SPARQL support built in, we want to not only build a basic feature model with required and optional features and feature groups. We want to take a look at how it is possible to include attributes into this model.

Additionally we try to include more advanced feature model functionality. The inclusion of SPARQL in SHACL makes it possible to define complex constraints like summing over an attribute. We want to see how well this functionality can be integrated with normal feature models, and how performance is impacted by the addition of this functionality.

Apart from building the feature model itself, we also need a way to define constraints. To include support for attributes and and extended features in constraints, we want to develop a textual language including a parser to facilitate construction of constraints in a user friendly manner.

3 Related Work

This chapter gives an overview of works related to feature modelling and variant management.

As a feature model lies at the heart of this thesis, we will give a detailed overview over them and explain the different types of feature models that have been proposed. We will first give an introduction to what feature models are, and how the most basic feature models are structured. Since the first feature models a lot of progress has been made, making feature models more powerful, so we will take a look at what additional features have been proposed. Finally, we will look at some variant management systems that already exist and which features they support.

3.1 Feature Models

Feature models model the commonalities and differences between components in a product line. This section will give an overview over the types of feature modelling approaches that were proposed over time.

3.1.1 Semantic Web Based Analysis of Product Line Variant Model

Feature modelling is all about finding commonalities in a product domain. Ontologies have shown to be well suited for capturing the common vocabularies in any field [28]. An approach to map a feature model into OWL is presented in [26].

For modeling the features, they determined what information is saved about a variant and created a table with the following information:

- Variant name
- Variant type (mandatory / optional)
- Sub domain (applicable area of the variant)
- Relation between sibling variants (alternative / or)
- Dependency (the variants that a variant depends on)

Additionally, the require and exclude relations are also included in the variant model. This feature model now has to be converted to the OWL-DL format. There are six relation types between features supported by the feature model which also have to be mapped to OWL. The relations are: mandatory, optional, alternative, or, optional alternative and optional or. Additionally the require and excludes cross-tree-constraints are modelled.

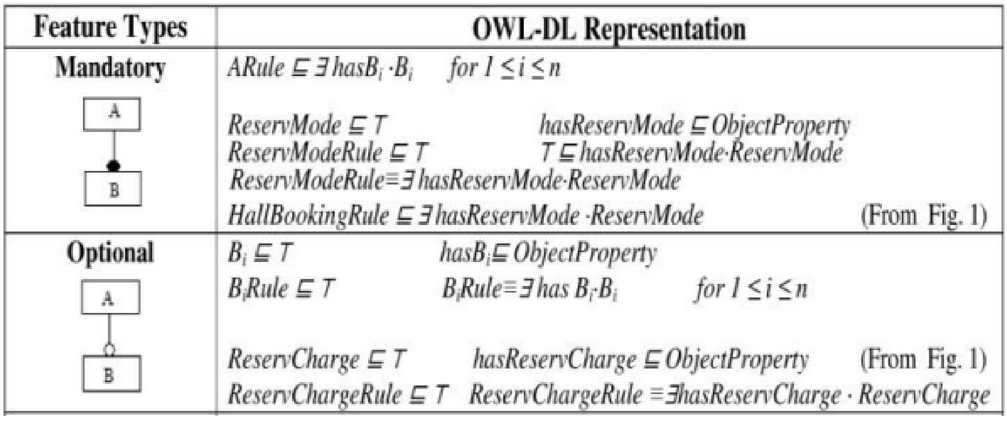
First an OWL ontology is built for the features and nodes of the feature tree, where each node is modelled as an OWL class. A rule class is defined for each of the nodes, defining the constraints for it and an object property is created for each type of edge in the feature model.

Now the feature relations can be translated to OWL. Figure 3.1 shows a part of the table containing OWL definitions for the the mandatory and optional relations. An example ontology was implemented using Protege. The RACER reasoner was used to check for consistency. Protege can not only show if a configuration is consistent, but also which features produce inconsistencies.

Constraint Solving

The second part of [26] is looking at algorithms for constraint solving. Since many constraint solving systems are created for general use, they often perform poorly compared to a domain specific system.

Feature models can be represented as boolean formulas. These formulas have certain properties that general boolean formulas do not necessarily have. One of the properties is that because of the nature of feature models, any boolean formula of the feature tree is necessarily satisfiable. Therefore,



Feature Types	OWL-DL Representation
Mandatory	$ARule \sqsubseteq \exists hasB_i \cdot B_i$ for $1 \le i \le n$ $ReservMode \sqsubseteq T$ $hasReservMode \sqsubseteq ObjectProperty$ $ReservModeRule \sqsubseteq T$ $T \sqsubseteq hasReservMode \cdot ReservMode$ $ReservModeRule \equiv \exists hasReservMode \cdot ReservMode$ $HallBookingRule \sqsubseteq \exists hasReservMode \cdot ReservMode$ (From Fig. 1)
Optional	$B_i \sqsubseteq T$ $hasB_i \sqsubseteq ObjectProperty$ $B_iRule \sqsubseteq T$ $B_iRule \equiv \exists has\ B_i \cdot B_i$ for $1 \le i \le n$ $ReservCharge \sqsubseteq T$ $hasReservCharge \sqsubseteq ObjectProperty$ (From Fig. 1) $ReservChargeRule \sqsubseteq T$ $ReservChargeRule \equiv \exists hasReservCharge \cdot ReservCharge$

Figure 3.1: Part of table containing OWL definitions for feature model relations [26].

a domain specific constraint solver taking these properties into account can perform better than a generic one. The paper presents a new domain specific constraint solver for feature trees called the feature tree constraint system (FTCS). They provide several algorithms to check if a configuration is valid, counting the number of possible configurations and find the minimum configuration.

The FTCS was evaluated against the general purpose constraint solver Choco. Testing showed that FTCS vastly outperformed Choco in counting possible solutions. The time to count solutions was up to 100 times faster for FTCS for feature models with 50 features and it still performed sufficiently for feature models with 10000 features where the Choco solver timed out.

3.1.2 Applying Semantic Web Technology to Feature Modeling

We take a look at another ontology based approach. This approach has additional support for attribute based constraints [29].

The reason for using OWL has multiple reasons. Firstly, because OWL is the standard Semantic Web language, it will make their approach interoperable with many other applications. Secondly, OWL allows the modeling of classes along with constraints on them, which provides a seamless transition

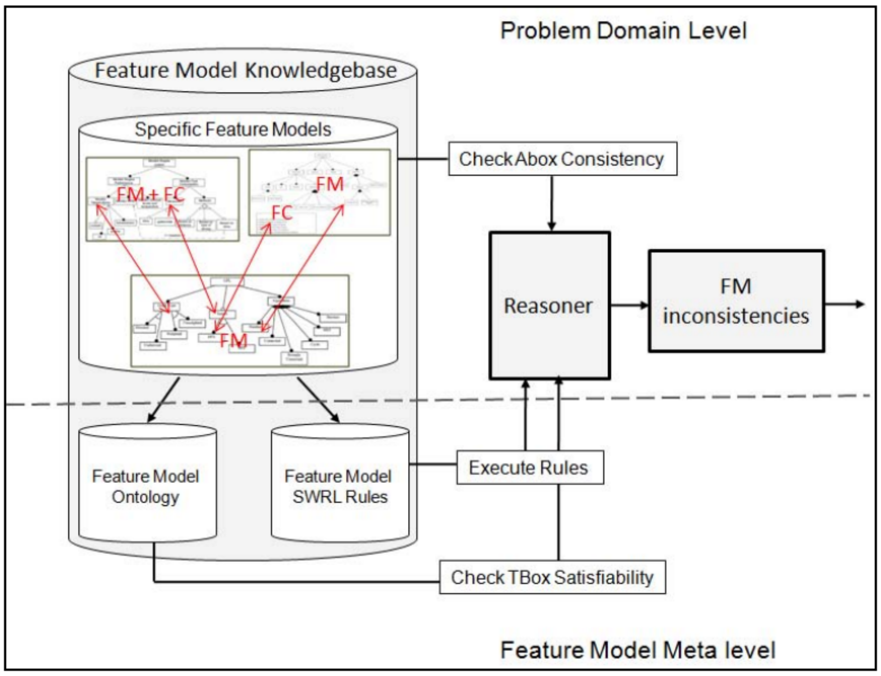


Figure 3.2: Overview of the system architecture [29]

from the real world model view to the ontology view. Lastly, OWL was designed with reasoning on models in mind, making it possible to infer rules. Similarly to [26] they translate feature models into an OWL representation and provide a representation for the needed feature model semantics. In addition to the feature model ontology, they use Semantic Web Rule Language (SWRL)[30] rules to ensure the consistency of the feature model. Figure 3.2 shows an overview of the system.

The basic structure of the feature model is taken from the FODA representation combined with concepts from FORM and FeatuRSEB. Semantics for cardinality constraints and attributes were added. Using an ontology allows the feature model and feature model constraints to be represented in the same model. The ontology contains classes for features, alternative and or relations, attributes and feature relations. The relations between classes are represented via a set of properties. There are several constraint types. Firstly, feature to feature constraints. These are constraints like requires and excludes. Feature value constraints represent constraints on values attached to features. *EqualTo*, *GreaterThan* and *LessThan* are supported. Attribute value constraints constrain the value of a feature attribute. The same operations as for feature value constraints are supported.

The situations that cause the model to become inconsistent, are described using SWRL rules. One example is a feature both requires and excludes another feature. This is represented as the rule:
`Requires(?x,?y) && Excludes(?x,?y) then problem(?x,?y).`

The rules together with the ontology get passed to a reasoner to check the consistency of the model.

3.1.3 Decision-making coordination and efficient reasoning techniques for feature-based configuration

The product configuration process often involves many people from different areas. It is important that the configuration process facilitates the collaboration by multiple involved partes. Here, an approach is presented for a product configuration system that makes it possible to collaborate on the configuration task [31].

Often, there are multiple stakeholders involved in configuring a product. When a local decision (e.g. include a feature or exclude a feature) causes an inconsistency because of another local decision by another stakeholder, it is often difficult to resolve those conflicts. This paper presents an approach to resolve these conflicts that arise as part of collaborating on a configuration task.

Constraints for feature models can often be encoded as a boolean constraint satisfaction problem (B_CSP), which can in turn be translated to a SAT problem.

Collaborative Configuration

The approach is split in 2 parts: Generating a configuration plan and the configuration itself. The goal of the configuration plan is to describe what

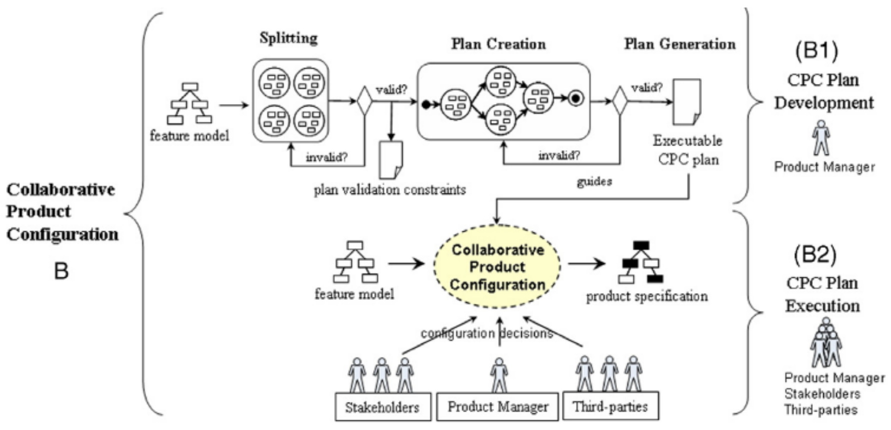


Figure 3.3: An overview of how a product is configured using collaborative methods [31]

the configuration tasks are, and in what order they should be carried out. The configuration part is then carried out by the stakeholders according to the configuration plan.

Generating the configuration plan is done by splitting the configuration decisions into more manageable groups called configuration spaces. The splitting of the feature space has to adhere to rules so that they can be configured by different groups of stakeholders. Once the feature model is split into configuration spaces, the configuration plan can be created. The plan consists of a set of configuration sessions where each session can configure multiple configuration spaces. The order of these sessions is specified in the plan. Sessions also have to adhere to rules so that the configuration cannot lead to an invalid product. These rules should be validated by an automatic system. After the plan is created, an executable plan is generated. The configuration of the product is then carried out by the stakeholders according to this executable plan.

Figure 3.3 depicts a rough overview of how this process is done. We start in the plan development phase with the splitting step. After that, the plan creation is carried out and an executable plan is generated. After that, the plan execution phase is conducted according to the plan.

The plan splitting step is the responsibility of the product manager, as they have an overview of the expertise of the stakeholders. In this phase the configuration problem is split into smaller configuration spaces. Each space contains a subset of the features that have to be configured. Certain features can appear in more than one configuration space. This is only allowed for so called junction points. A junction point is a feature that connects a child configuration space to a parent. Only a single parent space can be connected to a child. There are two types of dependencies between spaces that are relevent. They are strong and weak dependencies. Configuration spaces are called weakly dependent if some decisions in one space influence decisions in the other space. These dependencies can be specified by extra constraints for the feature model. Two spaces are strongly dependent if a decision in one space can affect all decisions in the other space. A child space is always strongly dependent on its parent.

After the plan is split into configuration spaces, the plan creation phase can begin. In this phase configuration sessions are created, and configuration spaces assigned to these sessions. The sessions are also scheduled in sequential and parallel flows. Validation rules are added to the plan to ensure its correctness. To validate plans there are two rules to be considered. If two spaces are weakly dependent, then they have to be scheduled in subsequent sessions. If space B is strongly dependent on space A, then A must be scheduled before B. These rules only apply to spaces that are placed in different configuration sessions, otherwise conflicts can be locally resolved.

The last step is the plan generation step. In this the plan is transformed to an executable representation. In the plan we have we assume that all configuration sessions will be executed and conflicts will arise. In reality, many configuration spaces do not have to be configured, as previous decisions excluded the space (eg: a junction point has been excluded). This workflow is referred to as the executable plan.

3.1.4 Automated Reasoning on Feature Models

We include attributes in our features to enable the construction of more powerful constraints. Here we have a CSP based approach for validation, that included attribute constraints [32]. It also introduces a way to automatically reason over feature models.

The feature model is based on the typical feature model relations *mandatory*, *optional*, *alternative*, *or*. Attributes are defined as a characteristic of a feature that can be measured. For example cost or bandwidth. Extra-functional features are defined as relations between one or more attributes of a feature. The base feature model definition is extended by allowing relations amongst attributes.

To resolve conflicts in the configuration, the feature model is translated into a CSP. The features make up the set of variables for the CSP. The domain for each variable is boolean. Extra functional features are the constraints. The relations of the feature model become constraints in the feature model in the following way:

Mandatory: f is a parent feature with f1 a child of f with a mandatory relation. Then the mandatory constraint is: `f1 = f`.
Optional: f is a parent feature with f1 a child of f with an optional relation. Then the optional constraint is: `f1 => f`.
Or: f is a parent feature with f1, ..., fn children of f in an or relation. Then the or constraint is: `f1 v ... v fn  <=> f`.
Alternative: f is a parent feature with f1, ..., fn children of f with an alternative relation. Then the alternative constraint is:
`(f1 <=> (~f2 ^ ... ^ f)) ^ ... ^ (fn <=> (~f1 ^ ... ^ f)).`

With this mapping, features, constraints and extra functional constraints can be handled in the same CSP. With the feature model mapped to a CSP, reasoning can be applied. We can infer the number of valid products that can be configured from the features model. Filtering allows users to apply rules to the model to filter out a set of products. This is useful because users are often not interested in the entire product set. We can calculate the entire possible product set which are all solutions to the CSP. Lastly it is possible to look for the optimal product according to a specified criterion.

The system was implemented using the OPL Studio CSP solver. A simple GUI prototype was developed to showcase the system. Additionally, a simple XML language was developed to store models and a parser to load models.

3.2 Variant Management Approaches

3.2.1 Integrating Semantic Web Technologies and ASP for Product Configuration

Since we want to use SHACL to validate feature models, we are looking for other approaches that have done something similar before. Here, an approach is presented to validate configurations with SHACL shapes that are similar to feature model constraints [33]. They are modeling a configuration task of different types of hardware modules into RDF and define a shapes graph that contains constraints for how these modules can be used together. With the defined constraints it is possible to check if any given configuration in a data graph is valid and if not, which constraints are not fulfilled.

The approach is illustrated using an example. The example consists of hardware elements which are controlled by modules. Each module must be placed in a frame, and each frame be mounted on a rack. There are different types of elements, modules, racks and frames. Figure 3.4 shows graphical representation for how an RDF data graph for such a configuration task looks like.

A constraint for such a configuration could be that an element needs to be connected to exactly two modules of a certain type. Below is a SHACL representation for how such a constraint looks like. The shape targets all elements of type *ex:ElementB*. Each of these elements has to fulfill the conditions specified by the property shape defined by *sh:property*. The *sh:path* property denotes that the focus nodes are nodes that the value nodes that the properties are validated against are the ones reached via *ex:requiredModule*. In this case the conditions these nodes have to fulfill is that there are exactly 2 of them and the have the type *ex:ModuleII*.

```
ex : ElementBRequiredModuleShape
    a sh:NodeShape;
    sh:targetClass ex:ElementB;
    sh:property [
      sh:path ex:requiredModule;
```

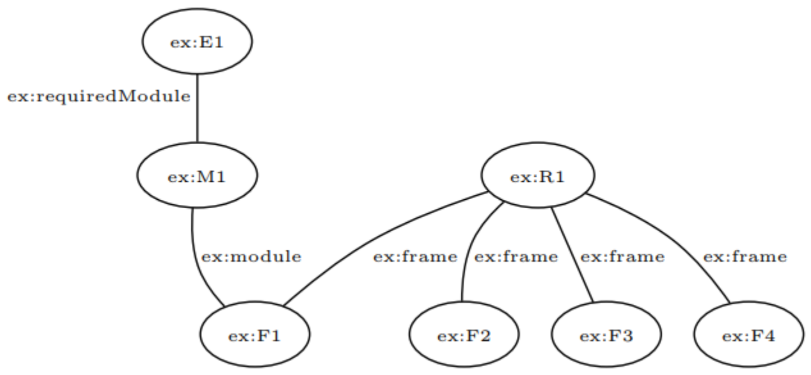


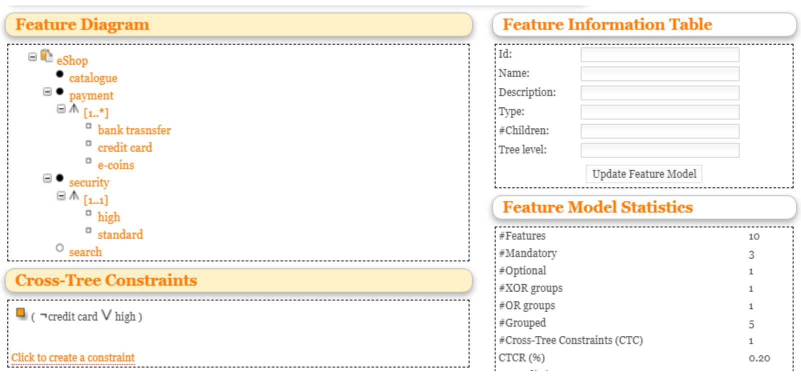
Figure 3.4: An example RDF graph showing a possible configuration of hardware modules [33]

```
        sh:minCount 2;
        sh:maxCount 2;
        sh:class ex:ModuleII;
] .
```

When configuring the system, the SHACL rules tell the user if a selection caused a rule violation. But often there is only one valid choice that can be selected. This is why in addition to only using SHACL for validation, the SHACL rules are translated to ASP rule. These rules can be used to infer these decisions automatically using an ASP solver.

3.2.2 Software Product Lines Online Tools

Software Product Lines Online Tools (S.P.L.O.T) [34] are a collection of web-based tools with the purpose of creating, configuring and validating feature models. The functionality for creating and validating models is similar to what we want to accomplish. SPLOT also has a repository of feature models



Feature Diagram

- eShop
  - catalogue
  - payment
    - [1..*]
      - bank trasnsfer
      - credit card
      - e-coins
  - security
    - [1..1]
      - high
      - standard
  - search

Cross-Tree Constraints

(¬credit card ∨ high)

Click to create a constraint

Feature Information Table

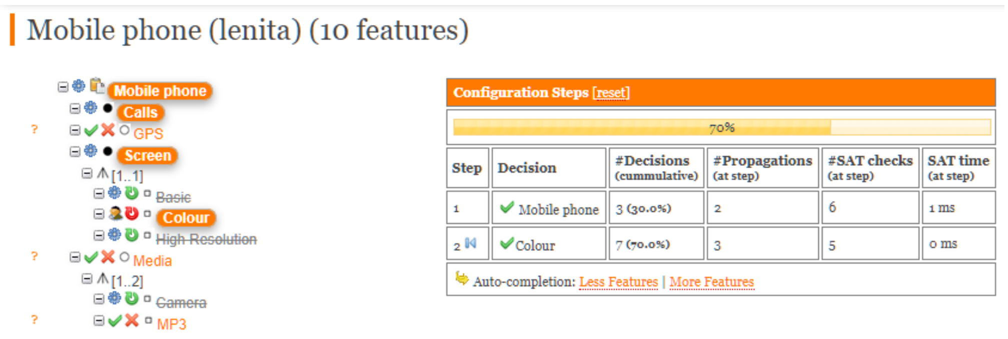
Id:	
Name:	
Description:	
Type:	
#Children:	
Tree level:	

Update Feature Model

Feature Model Statistics

#Features	10
#Mandatory	3
#Optional	1
#XOR groups	1
#OR groups	1
#Grouped	5
#Cross-Tree Constraints (CTC)	1
CTCR (%)	0.20

Figure 3.5: User interface of the S.P.L.O.T feature model editor



Mobile phone (lenita) (10 features)

- Mobile phone
  - Calls
  - ? GPS
  - Screen
    - [1..1]
      - ~~Basic~~
      - Colour
      - ~~High Resolution~~
  - ? Media
    - [1..2]
      - ~~Camera~~
      - ? MP3

Configuration Steps [reset]

70%

Step	Decision	#Decisions (cummulative)	#Propagations (at step)	#SAT checks (at step)	SAT time (at step)
1	✓ Mobile phone	3 (30.0%)	2	6	1 ms
2	✓ Colour	7 (70.0%)	3	5	0 ms

Auto-completion: Less Features | More Features

Figure 3.6: User Interface of the S.P.L.O.T feature model configuration tool with an example feature model

that we can use to benchmark our system. The tools can be found on the SPLOT website [1].

The feature model creator tool lets a user create and export a feature model. Feature models are saved in the SXFM [35] format. SFXM is an XML based format that stores the feature tree and its associated constraints. It is also possible to import an existing model.

The feature model editor tool, allows creating and editing feature models using a graphical interface. Figure 3.5 shows a screenshot of the editor interface. In the feature diagram section, the feature tree is built. Features or feature groups can be created. Feature groups can be *Or* or *Alternative*

[1]http://www.splot-research.org/

groups. Singular child features can be marked either optional or mandatory. The Cross-Tree-Constraints section allows the specification of additional constraints. Constraints use identifiers for feature in the feature tree and have to be in CNF format.

The feature model analysis tool allows analyzing an SXFM feature model based on SAT solvers and Binary Decision Diagrams. It displays information about the feature model (eg. number of features, tree depth, and clause density). It also has the ability to detect dead features and count the number of valid configurations.

The product configurator enables interactive configuration of a product. The user can include or exclude features from the feature tree. Whenever a feature is selected, the constraints of the feature tree are validated. Any features that are no longer possible to be selected are crossed out in the UI. By clicking on an excluded feature, the conflicting features can be displayed. It is possible to automatically infer feature selections where there is no user choice. Figure 3.6 shows the user interface for the configurator. We can see the included and excluded features in orange and grey respectively. The table on the right shows information about every step in the configuration process. The configurator uses a SAT solver for validating configurations and inferring decisions.

SPLOT has a repository of feature models in the SXFM format that can be used for empirical testing of feature models. Most of these models contain constraints consisting of 3-CNF formulas. They also provide a feature model generator with which such feature models can be generated.

4 Design and Concept

Although feature modelling has been long established, even now many feature models only support basic boolean constraints. There are approaches that use Semantic Web technology, most notably OWL, to implement feature models. The Semantic web gives the big benefit that it provides a standardized format to represent models. Semantic Web specifications have been implemented in many different languages, making models platform independent. Recently the SHACL standard for validating RDF models has been introduced.

RDF and SHACL provide an ideal environment to model and validate a feature models, and have the functionality built in to facilitate the construction of complex constraints. There has been an approach already to use SHACL for configuration tasks [33]. This model is specific to a certain domain. Our goal is to build a domain independent model, and a system that can be used to construct constraints. The reason we do not want to use RDF directly is, that while it is human readable, it is cumbersome to write by hand. Therefore we want a more user friendly way to define constraints.

In this chapter we define the requirements for the feature model we use and design a textual language for constructing constraints for the feature model.

4.1 Defining Feature Model Requirements

In this section we will define the requirements that our validation system needs to support in order to properly handle model validation. To do so we will build and example model that will cover a breadth of constraints that appear in product configurations.

4.1.1 System Requirements

The example we will build is the selection of components to build a PC. A PC consists of multiple different components that need to be decided on. There is a large number of possible options for each component, with a multitude of restrictions on their selection. This makes it a good candidate to define what a system like the proposed one needs to be capable of.

We start by defining the overall components a PC needs. The simplest configuration for a working PC is a motherboard, a CPU, RAM, a storage device and a power supply. These will be the component categories from which a user will select. Each of these of course has a long list of variants to choose from. Of course, we cannot just arbitrarily combine whichever component we want of each component category. An example for this is motherboards restricting the type of CPU they support. A motherboard with an Intel CPU socket cannot use an AMD CPU and vice versa. Our PC configuration can have only one motherboard and CPU, but it is possible to use more than one RAM module or hard drive. Concretely, these constraints would be as follows for our example model:

```
if Motherboard == IntelMotherboard then
   restrict "AMD Ryzen 5";
   restrict "AMD Ryzen 7";
   ...

maxcount(CPU) == 1;
maxcount(Motherboard) == 1;
```

This gives us a basis for rules we need to model the restrictions of features. Firstly, we need rules that either restrict or require a certain component, if a specific component is selected. Secondly, we need to be able to define how many of a certain component must be selected.

While this approach works fine for smaller models, it becomes unfeasible when the number of components and restrictions grows. For example: each motherboard will only be compatible with a subset of possible CPUs, therefore we would have to define a rule to explicitly exclude any non-compatible CPU, for a selected motherboard. In reality, the most common restriction for CPU choice is the type of CPU socket on the motherboard. Considering this, it would be a good idea to be able to define attributes as name-value pairs on features. In our example we would then define the attribute CPU-Socket on our motherboard components and give it the specific socket it has as a value. We then need a way to define rules that will handle these attributes. An example for how a rule would look like:

```
if (CPU == SomeAmdCpu) then
  require Motherboard::Socket == "AM4";
```

Components in a PC often consist of a lot of sub-components. For example, a motherboard will have a memory controller which has it's own attributes. Instead of having to define those attributes on the motherboard itself, it would be beneficial to be able to create sub-components with their own attributes. If for example, we want to use an M.2 PCIe SSD, then it is required that the memory controller of the motherboard supports the NVMe specification.

Some restrictions don't just apply at one attribute in isolation. If we think about the requirement for a power supply, then we need to ensure that the sum of power consumption of all the selected components is smaller than the capacity of the chosen power supply. Another example would be that the amount of components that require a PCI slot has to be smaller or equal to the amount of PCI slots on the board. So we need a way to define aggregate functions and compare them to attributes.

Sometimes a component does not specifically require one component. A motherboard with the *mATX* form factor can fit in a case that has the *ATX* form factor. Therefore, it should be possible to state alternatives in rules. In general we want to be able to combine statements using logical operators.

When building a PC, the user will often have a specific budget in mind. This budget varies depending on the user and cannot feasibly be hard coded in the model. We want users to be able to set certain parameters in the model during configuration.

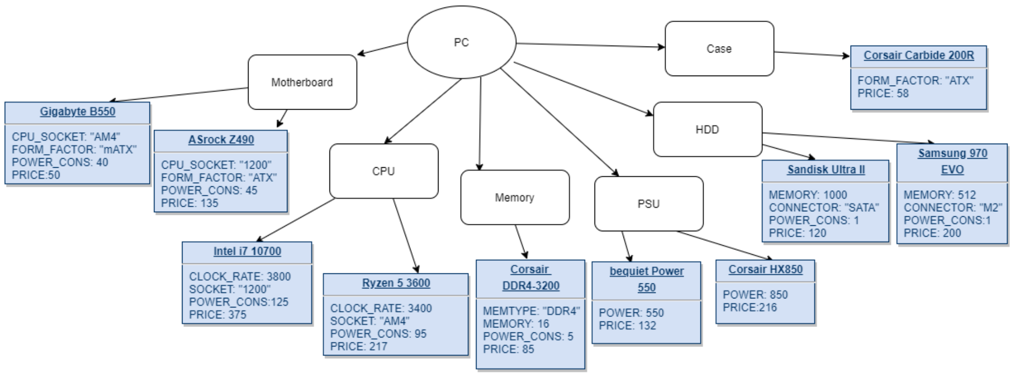


Figure 4.1: Feature tree for PC configuration

A complete version of a feature tree for PC components can be seen in Figure 4.1

4.1.2 Requirements Summary

This section will give a summary of the features our proposed system needs as derived from the example in the previous section. First, there has to be a way to define features and attributes on these features. We also need to be able to define restrictions between the features. It should be able to easily update the features and restrictions.

There should be an intuitive way to select features from the component model. The system then needs to check if this selection is valid. If not there should be feedback as to why it is not valid.

Restrictions are defined in the form of if condition then action. Conditions are component selections that can be arbitrarily combined with *and*, *or* and *not* operators.

Actions consist of an action type and an operation. The action type can be either restrict or require. The operand can be simply be a feature or an attribute in the simplest form. This would either require or restrict this feature or attribute from being included in the configuration. It is also possible to specify comparison operators when the operand is an attribute(<, ==, >). With these it is possible to restrict the possible values that an attribute can have. It is possible to either specify a constant as second operand, or another attribute. Attribute rules can additionally compare to parameters, the value of which can be set by the user during the configuration process. Lastly it should be possible to specify that the sum of an attribute has a certain value.

4.2 Formal Definition

In this section we will formally define the structure of the feature tree and its constraints. We will use the example defined in Chapter 4.1 to illustrate this definition. See Figure 4.1 for a graphical representation of this example model.

4.2.1 Basic Relationships

We can define the structure of the feature model as follows:

```
model := <varpoint>*
varpoint:= <name> <variant>* <cardinality>
variant := <name> <varpoint> <attribute>* <variant>*
cardinality := <min> <max>
attribute := <name> <value> <type>
min := int
max := int
type := string | int
```

We call the features in the top most level of the feature hierarchy variation points and the rest of the feature variants. In the PC example the variation points are the categories of components, while the variants are actual instances of PC parts.

Apart from the parent-child relationships, there are other constraints on variants. According to [13], the basic set of relationships are:

Mandatory

A *Mandatory* relationship denotes that this feature has to be included in any configuration where the parent feature is also included. An example for a mandatory feature from Figure 2.1 are the *Calls* feature and the *Screen* feature.

Optional

An *Optional* relationship denotes that this feature can optionally be included in the configuration if the parent feature is also included. In Figure 2.1, the optional features for the Mobile phone feature are the *GPS* feature and the *Media* feature.

Alternative

Alternative relationships are cardinality constraints of a feature model. An alternative relationship is between one parent feature and a group of it's child features. It means that if the parent feature is included in the configuration, then exactly one of these child features is also to be included. We can see an alternate relationship in Figure 2.1 between the *Screen* feature and its children.

Or

Or relations are another type of cardinality constraint. They describe the constraint that if a parent feature is included in the configuration, then one or more of the child features in the specified group must also be included in the configuration. In Figure 2.1 an Or group can be seen for the feature *Media* and its children.

In addition to constraints between variation points and variants, we also add constraints between two variants.

The basic relationships that can be defined are require and restrict. The require relationship is used to define that variant A requires variant B to be selected.

$require(var1, var2) := ifselected(var1) => selected(var2)$
$var1, var2 \in Variants$
This can also be written as the logical implication:

$var1 \rightarrow var2$

Similarly the restrict relationship excludes variant B if variant A is selected. This means that if variant A was selected, variant B cannot be selected.

$restrict(var1, var2) := ifselected(var1) => notselected(var2)$
$var1, var2 \in Variants$

In logic terms this would be:

$var1 \rightarrow \neg var2$

Cardinality constraints can be defined on variation points. If a variation point has a cardinality constraint: $cardinality(vp, n, m)$, that means that at least n and at most m variants of this variation point have to be selected.

$cardinality(vp, n, m) := count(vp) >= n \wedge count(vp) <= m$
$count(vp) := numberofselectedvariantsbelongingtovp$

4.2.2 Extended Features

The section above defines a feature model with basic functionality. We want to extend the functionality of our feature model to allow for more complex rules. The extensions we will add are attributes, which were proposed multiple times as extensions for feature models([14, 15, 16]), logical combinations of selections for rules [4], aggregate functions that work with attributes and global parameters.

4.2.3 Combination of selections

We want to extend this simple rule set to enable the combination of selections as conditions for restriction and requirement relationships. We extend the rule set from before with logical and, or, and not operators. This will extend our require and restrict actions to use conditional expressions instead of just variants.

$require(condition, var) := ifisTrue(condition) => selected(var)$
$var \in Variants, condition := logic - expression$

$restrict(condition, var) := ifisTrue(condition) => notselected(var)$
$var \in Variants, condition \in logic - expression$

logic-expression:

$A \wedge B$
$A \vee B$
$\neg A$
$A, B, C \in Variants$

These operators can be arbitrarily combined allowing to specify complex conditions of variant selections. This operator nesting is also possible for the statements inside the if body.

4.2.4 Attributes

We extend the feature model by attributes. Attributes provide additional information about variants. They consist of a name and a value. The value can be either a string or an integer. Having attributes enables us to extend the functionality of rules. Instead of requiring or restricting a specific variant, it is possible to require or restrict an attribute.

Because attributes can have values other than simply selected/not selected, we can extend the require and restrict rules to include comparison operators. This allows us to define constraints on properties of variants instead of variants themselves which makes rules simpler to define for large feature models. When defining an attribute rule, an attribute name, an operator and a value need to be specified. The resulting expression needs to hold for every attribute with that name, that belongs to a selected variant.

```
For every attribute

require(cond, attr, operator, value) :=
    if cond => attr.Value operator value
operator in {==, !=, <, <=, >, >=}
```

The statement above needs to hold for each attribute with the correct attribute name of selected variants.

In the context of our example, each motherboard will only be compatible with a subset of possible CPUs. If we were to use only the basic feature set, we would have to define constraints to explicitly exclude any non-compatible CPU for a selected motherboard. In reality, the most common restriction for CPU choice is the type of CPU socket on the motherboard. We add an attribute to our motherboard variants to indicate which socket it has, and create a rule to only allow selections of motherboards with said attribute.

4.2.5 Functions

Having defined attributes for variants in the previous section, we now want to enhance the functionality of them by functions that aggregate a result from all instances of an attribute. With the attributes before, we would always evaluate all instances of an attribute in a rule separately. We add the functions sum, and count, that calculate the sum of all instances of the attribute and then compare the resulting value with the value given in the rule.

In the context of our example, power supplies have a maximum of power they can safely supply. The sum of the power consumption attributes of all parts cannot exceed this value. So we could create a rule saying that if a power supply with a maximum power of 500W is selected, then the sum of all values from attributes with the name 'power_consumption' must be smaller than 500.

4.2.6 Global Parameters

We extend the model by global parameters. Without parameters, when a rule contained a comparison operator, the values that were compared against had to be constant literals. We allow the definition of parameters that behave like variables with a name and value. Instead of a constant value, we can specify the name of a parameter, the value of which will then be used for the comparison. The value of these parameters can be adjusted during the configuration process.

A user might want to specify a maximum price for the entire system. This price might not be a hard limit, but just a soft cap that they might be willing to go over, so we might want to adjust it later. With the global parameter model extension, we can just specify the maximum price as a global parameter and have a rule that ensures that the sum of prices does not exceed that.

4.3 Language Design

We have defined how our feature model is structured. Now we need to define a way to specify extra constraints. For that purpose we will define a new textual domain specific language.

The language is split into two parts: First the requirement part, which allows specifying constraints that always have to hold. Secondly, the conditional rules, which are structured as if-then constraints.

There are several reasons to include a requirement construct. The main reason is to allow the specification of cardinality constraints. Secondly it allows the user to specify that certain features have to be included in any configuration.

Conditional rules are the more general constraint type. They consist of the condition and the constraint body, which can contain any number of constraints. These have a constraint type and then the actual constraint. The constraint type can be either require or restrict. Require means the constraint must be fulfilled, restrict means the constraint must not be fulfilled.

The simplest constraint we can define is requiring or restricting a specific feature, in which case the constraint is simply the identifier for the feature.

```
if featureA then
   require featureB
   restrict featureC
```

Since the features in our model have attributes we can also specify the identifier for them like we do for variants. Attributes also allow us to form constraints involving the comparison of values. Therefore, if the given identifier is an attribute, we can add a comparison operator and a value or parameter identifier.

```
if featureA then
   require attributeA == $parameter1
   require attributeA < 100
```

To specify a function we write the function name followed by the attribute the function should act on in parentheses.

```
if featureA then
   require sum(attributeA) < 100
```

These were the basic methods of constraint bodies. It is possible to combine these using logical operators to form more expressive constraints. Available operators are *and*, *or* and *not*. The expressions can be nested using parentheses.

```
if featureA then
   require attributeA < 100 || featureB
```

When a constraint specifies an attribute, more than one feature that is selected can have that attribute or none. In this context, there is an important distinction to be made between two cases. Looking at the above example, we could either mean that there has to be an attribute < 100 on a selected feature. Or we could see this as all *attributeA* need to be < 100, but it is not required that there are any. For this purpose we can specify either the *each* or *any* modifier before a constraint. Any meaning there has to be at least one attribute, each meaning that each attribute needs to fulfill the condition.

```
if featureA then
   require any attributeA < 100 || featureB
   comment "There has to be at least one attributeA < 100
            or featureB has to be selected."
```

To make it possible to describe why constraints are there, it is possible to create comments. One comment can be specified in each rule block. These comments will also be displayed in case a rule is violated during the validation process. An example for a comment can be seen in the example above.

5 Implementation and Realization

In this chapter we will explain the implementation details of our product configurator. We will illustrate the approach using the PC configuration example from Chapter 4.2.1.

5.1 System Architecture

First we will briefly talk about how the application is structured and give an overview of the general control.

5.1.1 Components

The application can be grouped into several parts:

- Model Reading
- Compiling
- Model Transformation
- Model Validation

Figure 5.1 shows an overview of the structure of the system, and how the parts interact. We will briefly introduce each of them in turn.

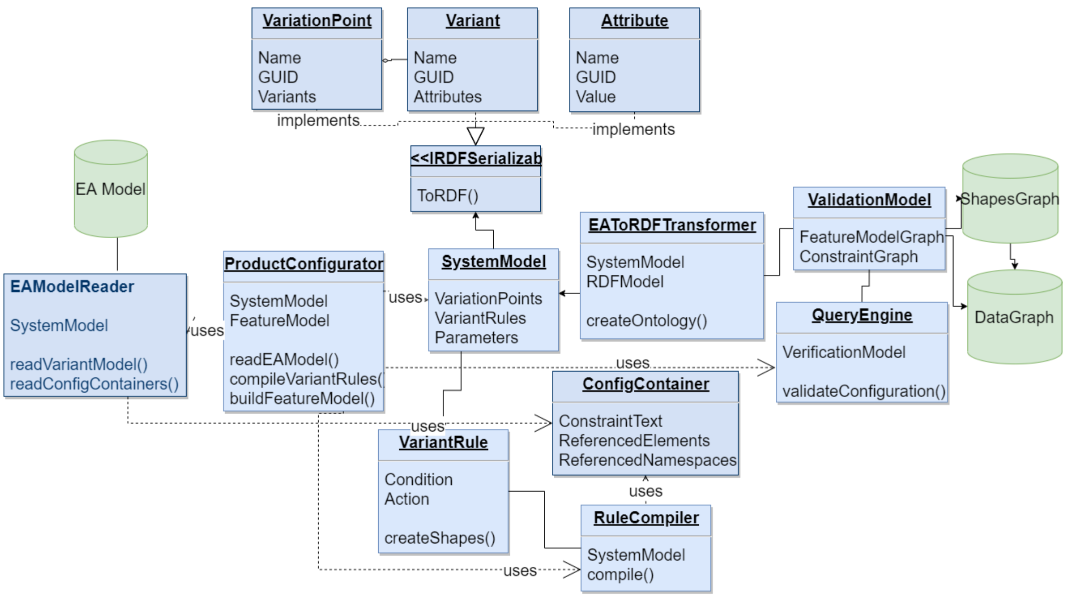


Figure 5.1: Class diagram of System components

Model Reading

The model reading part takes a model consisting of the variant model and the rules, and reads it into an internal data structure. We create structures here that let us look up variants by name or by identifier. This will be important for the rule compiler. The model we read is an Enterprise Architect model. We use Enterprise Architect, because it provides an easy way to build UML models, and also provides an interface for C# applications to work with them. The model is split into the feature part and the rule part. The elements are differentiated by their UML stereotype.

Compiling

The compiler is used to translate the rules from the user language into a structure that can be transformed into SHACL rules. It contains a parser for the grammar and a visitor for building the rule objects.

Model Transformation

The model transformation transforms the model into RDF and SHACL. It takes the read feature model from the model reader component and the compiled constraints from the compiler component and builds a valid RDF model and SHACL shapes corresponding to the constraints.

Model Validation

The model validation component is the component that does the actual validation part of the system. It takes a selection of features as input and updates the model with this selection. Then it runs the SHACL validator which validates the constraints in the shapes graph against the data graph. It will then generate a report with all violated constraints.

Figure 5.1 shows an overview of the structure of the system, and how the parts interact.

5.1.2 Application Flow

This section gives a rough overview of how the system works. After the application is started, the user sees the interface, from which they can select a path to the feature model that is supposed to be used. Having selected a feature model, the user can then instruct the system to read this model. Along with the reading process, the system also compiles the rules defined alongside the feature model. Once both the reading and compiling are completed the model transformer creates the actual RDF and SHACL graphs used for validation. When this is completed, the user interface is updated, showing all features that were defined in the model. Figure 5.2 shows a rough overview of this process.

The user can now select the components they want to be included in the product configuration. Once all components that should be part of the configuration have been selected, this selection can be validated. Clicking the validate button will pass the selection to the model validator, which validates all constraints that were defined. A report is created containing all

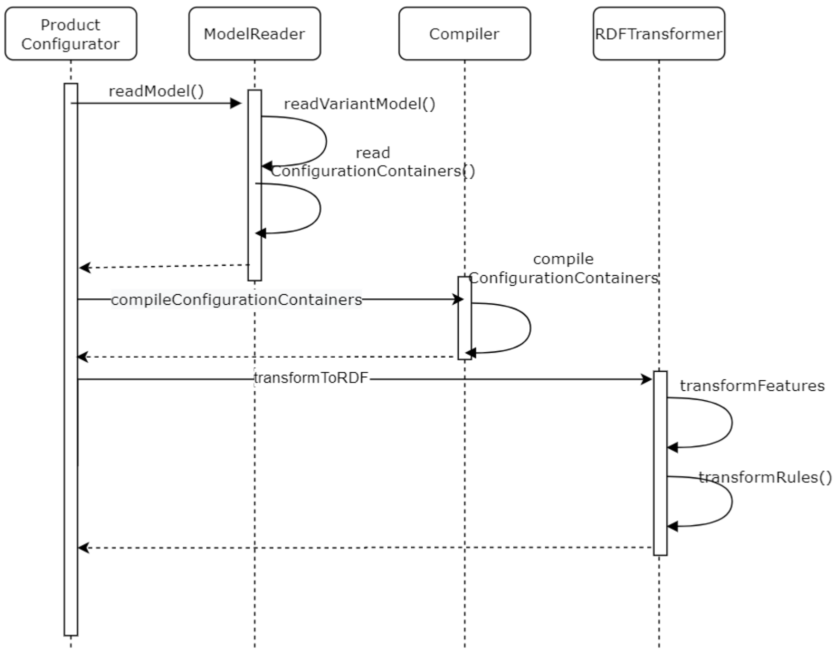


Figure 5.2: Flow diagram of model building.

violated rules together with the reason why they are violated. The contents of the report are shown to the user in the interface. We can see this illustrated in Figure 5.2

5.2 Creating the System Model

The model we read is an Enterprise Architect model. The model is split into the feature part and the rule part. Different types of elements are differentiated by their UML stereotype. The feature tree starts with the variation points. These are elements with the *VariationPoint* stereotype. Their child elements are the variants belonging to them. We would have *Motherboard* and *CPU* and the other component categories as variation points, and concrete instances of the categories as the variants. Variants can have multiple different types of child elements. First, variants can be nested, so they can have child variants themselves. Secondly, rules can be

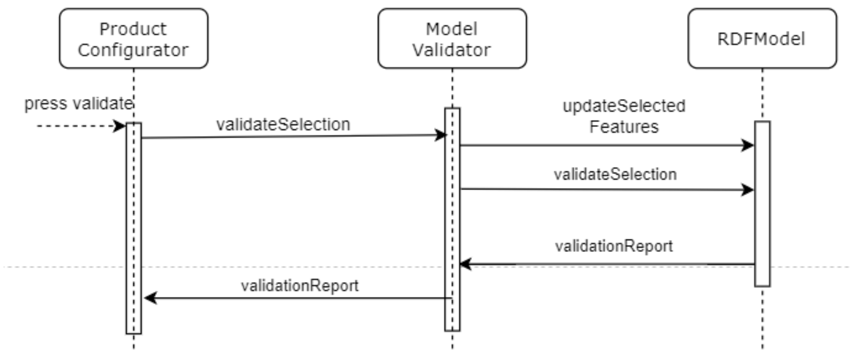


Figure 5.3: Flow diagram showing the process of validating a configuration

defined directly for a certain variant by defining a configuration container as a child element. Configuration containers are described in the next section. Attributes can be defined via the tagged value function of Enterprise Architect. Tagged values are name-value pairs defined on the element itself. Each feature of the *Motherboard* variation point has an attribute called *CPU_SOCKET*, denoting which type of CPU is compatible with the board. For example, the motherboard *Gigabyte B550* has the attribute *CPU_SOCKET: "AM4"*.

To read the elements of the model, we use the Enterprise Architect API for C#. This gives access to elements in the model and all their associated information. For us the properties `Name`, `GUID`, `TaggedValues`, as well as `ChildElements` are important. We start by creating a `SystemModel` object. The `SystemModel` contains all the information we need to create our RDF feature model. This includes all the variation points and variants with their attributes and configuration containers.

First, we create variation point objects with the `Name` and `GUID` of their respective Enterprise Architect element. For each variation point, we we create `Variant` objects from its child elements. Each variant also has a reference to it's variation point. The variant objects get added to the list of variants of the variation point. Next, we read the attributes, defined as tagged values on the variant elements. The `Attribute` object has a `Type` property. This property can be either integer or string. If the value of the attribute is enclosed in quotes, the type is string, otherwise integer. If the value cannot be parsed to an integer an error is thrown. We now recursively create variant objects for the child variants. Child variants get added to the child variant list in the variant object and the child variant gets a reference to it's parent. Lastly, we create configuration container objects for any configuration container child element for the variant and add it to the variant object.

Next, we read the configuration containers that were not already part of a variant. These are any elements with the *ConfigurationContainer* stereotype that are not a child element of a variant. Configuration containers are simply objects with a `Name`, `GUID`, and `ConfigurationText` property. Configuration containers defined on a variant also have a reference to that variant. Now that we have a system model and configuration containers, we can compile the latter.

5.3 Compiling Configuration Containers

The language for creating model constraints is implemented as an ANTLR grammar. It consists of the grammar rules and a visitor for interpreting the rules. We will first describe the implementation details for the used grammar, and then how the configuration containers are compiled using a visitor.

5.3.1 Grammar

We have given a short explanation of ANTLR grammars in Chapter 2. This chapter covers the specifics of the grammar we use.

We start with the main delimiter that wraps around the actual rules. This rule is the entry point for parsing a constraint text. The rule starts with an optional text that can be used to describe the configuration container. The actual rule construct starts with the `#rules` keyword. The actual rules start after this. The ANTLR grammar rule is the following:

```
rules : (TEXT '---')? '#rules '{' rule* '}' ;
```

Rules can be either a requirement or a conditional rule. For configuration containers that are defined on a feature, it can also be a variant rule. A variant rule is short for a conditional rule that has the selection of the attached variant as condition.

```
rule : requirementRule | conditionalRule | variantRule ;
```

The bodies for !verb!conditionalRule! and `variantRule` are the same, they only differ in their delimiter. `requirementRule` has mostly the same body, except it is also possible to specify cardinality constraints.

```
conditionalRule : '#if' '('condExpression')' '{'constraint*'}';
variantRule : '#rule' '{' constraint* '}' ;
requirementRule : '#requirement' '{'
                      (constraint | cardinalityConstraint)* '}' ;
```

The condition for conditional rules are logical expressions where each leaf expression identifies a feature.

```
condExpression : condExpression BOOLEAN_OPERATOR condExpression
               | NOT condExpression
               | '(' condExpression ')'
               | IDENTIFIER ('==' | '!=') IDENTIFIER;
```

An example for an expression like this would be the following, which evaluates to true if either the *Ryzen 5 3600X* or *Ryzen 9 5950X* CPUs are selected.

```
#if ('CPU' == 'Ryzen 5 3600X' || ('CPU' == 'Ryzen 9 5950X') { }
```

On the last line of the expressions rule above, the first `IDENTIFIER` rule is the identifier for a variation point, the second for a variant of this variation point. This is done to ensure that the right feature can be found if there are multiple features with the same name. Identifiers are delimited by single quotes and can contain spaces and other special characters.

The body for a rule always consists of a rule type and then the constraint, with a semicolon signaling the end of the rule. The rule type can be either *#require* or *#restrict*. This is followed by the actual constraint, which we call action in our system.

```
ruleBody : actionType actionExpression;
```

The `actionExpression` can be nested using logical operators the same way as conditional expressions. The difference is in the leaf expressions, which are the actual constraints of the rule. The most simple type of constraint is just an identifier for an element. This can be either a variant or an attribute. This is written as the type of element followed by the identifier.

```
constraint : '<' type '>' IDENTIFIER ;
type : 'Variant' | 'Attribute';
```

For example, if the motherboard *Gigabyte B550M* is selected, then the CPU *Intel i7-10700K* cannot be selected:

```
#if ('Motherboard' == 'Gigabyte B550M') {
  #restrict <Variant>'Intel i7-10700K';
}
```

If `type` is an attribute, we can extend with a comparison operator and an operand. Additionally, the *any* and *each* modifier can be added to specify whether we want to check that there exists at least one such attribute, or that all attributes fulfill the condition.

```
constraint :  ('#each' | '#any')? '<' type '>'
                  IDENTIFIER (operator operand)?;
```

Eg. the CPU *Ryzen 5 3600X* requires the motherboard to have the *AM4* socket:

```
#if ('CPU' == 'Ryzen 5 3600X') {
  #require <Attribute>'Motherboard::CPU_SOCKET' == "AM4";
}
```

This example illustrates the point we made above about the usefulness of attributes. Instead of having to specify all CPUs that are incompatible with a Motherboard, we can specify that the motherboard needs to have the correct socket for the CPU. The rule also only needs to be updated when another constraint is introduced via new component traits.

In addition to simply using an attribute in a constraint, we can also use the sum function to constrain the value of the sum of all instances of an attribute:

```
constraint : functionIdentifier '(' IDENTIFIER ')'
                 (operator operand);
```

As an example, a power supply has a maximum load it can handle. The sum of the power required by each component has to be less than the corresponding value.

```
#if ('PSU' == 'beQuiet Power 550') {
  #require sum('POWER_CONSUMPTION') < 550;
}
```

For attribute identifiers, it is possible to specify where in the feature tree the search for the attribute should start. To do so, starting from the root of the tree, each intermediate feature of the path has to be written in order, separated by '::'. We did this in the CPU socket example by specifying that the attribute has to be in a variant of *Motherboard*.

Lastly we will define the additional rules we can specify in a requirement body. These rules are for setting the minimum and maximum of selected variants of a variation point.

```
    cardinalityConstraint : ('#minCount' | '#maxCount')
                                IDENTIFIER NUMBER;
```

Naturally, every PC has one motherboard and at least one Memory module:

```
#requirement {
  #minCount 'Motherboard' 1;
  #maxCount 'Motherboard' 1;
  #minCount 'Memory' 1;
}
```

Every rule body can have one comment to describe the rule. The grammar rule for a comment is as follows:

```
    comment : '#comment' QUOTE TEXT QUOTE;
    TEXT : ~["]* ;
```

5.3.2 Rule Parser

The rule parser is a wrapper around the visitor. It iterates over all configuration container, adding the rules for each of them to the system model. A new visitor is created for each configuration container. Before the start rule of the visitor is called, an `ElementResolver` object is created and given the system model. This object is used to find variant objects given their name. This resolver is assigned to a member in the visitor.

For each configuration container, the first step is to create an instance of the lexer class generated from the grammar by ANTLR. This lexer will create a token stream from the rule text of the configuration container. This token stream is then handed to the parser, which was also generated by ANTLR, which generates an abstract syntax tree, and return the root context of it. We then call the visitor with this context as parameter to compile the configuration container into rule objects.

5.3.3 Visitor

The visitor creates concrete `VariantRule` objects from a configuration container. These will later be transformed to RDF. A base visitor class is generated by ANTLR. We derive from this class and overwrite the methods for each grammar rule.

The entry point for creating rules is the `VisitRules` method. Here, we iterate over all child contexts, which are the individual rule statements, so either an if statement, a requirement or a variant rule. In the body of each rule are the individual actions.

Condition

A condition has a selection expression at its core. There are 3 types of expression contexts, each with their own visitor method. One type is the leaf of the expression tree, which identifies a feature that should or should not be selected. The other two are expressions to combine the first type or another expression using logical operators and parentheses. Parsing starts at the top of the tree. Each of the methods visits its child nodes and creates an `Expression` object with the expressions returned from the child nodes. The types of expressions are `AndExpression`, `OrExpression`, `NotExpression`, and `SelectExpression`.

If the rule is a `RequirementRule` then the condition is simply an `AlwaysTrueCondition`. In case of a `VariantRule` the condition is simply a `SelectExpression` of the variant that the configuration container is attached to.

Actions

Actions have an action type and an action expression. The type denotes whether the expression has to evaluate to true or to false for the rule to be valid. The type and parsed action expression is used to create an `Action` object. Action expressions are parsed the same way conditions are. The only difference are the leaf nodes of the expression tree. Here, we differentiate between variant, attribute and sum actions.

Variant actions are an action denoted by having a variant identifier prefixed with <*Variant*>. First, the identifier for the variant is passed to the `ElementResolver` to retrieve the variant object from the system model. This is then passed to the `VariantAction` constructor and returned. For example, in the rule: `#require <Variant>'Motherboard::Gigabyte B550'` the element resolver would only search for the variant in the *Motherboard* variation point.

The second type are attribute actions. They are denoted by having an identifier prefixed with `<Attribute>`. Optionally, there can also be the `#each` or `#any` modifier before the identifier. Additionally, we can have an operator and an operand. The operand can be either a string or integer literal or a parameter. Strings are enclosed in quotes, and parameter names start with a `$`. In case the parameter has not been used before in another rule, it is added to the system model. The attribute identifier can contain a sequence of variant names separated by `::`. This denotes a path starting from the root of the feature model. Only attributes that are found in children of the feature at the end of this path are considered when evaluating the expression. The identifier, along with the modifier and operand are added to an `AttributeAction` object which is then returned. Lastly, we have the sum action expression. It consists of the sum function call with an attribute identifier as a parameter. Otherwise it's the same as the attribute expression. A SumAction object is created and returned. To illustrate this take the following example:

```
#if('Motherboard' == 'Gigabyte B550') {
  #require #any <Attribute>'MEMORY_TYPE' == "DDR4";
  #require #each <Attribute>'DRIVE_CONNECTOR' == "SATA";
}
#requirement {
  #require sum('HDD::CAPACITY') > $minimumStorage;
}
```

If the motherboard *Gigabyte B550* is selected, then every selected RAM module needs to be of type *DDR4*. The *#any* keyword denotes, that there has to be at least one feature included in the configuration that has that Attribute. Similarly, each hard drive that is included needs to use the *SATA* connector. The *#each* keyword here means, that each selected feature with that attribute needs to have the *SATA* value, but it is also valid that there are no such features. Lastly the requirement rule, says that the sum of all hard drive capacities in the configuration has to be at least as high as the value of the user defined parameter *minimumStorage*. Note the use of the property path `HDD::CAPACITY`, meaning that only the features in the sub tree with the HDD variation point as root are looked at.

A `VariantRule` object is created with the condition and the action. If one of the statements in the rule block was a comment context, then the provided comment text will be used for the rule. Otherwise a comment is automatically generated. This object is added to the list of created rules of the visitor, which can be accessed by the `RuleParser`.

5.4 Building the Feature Model

We have built the system model previously and compiled the configuration containers. Now we need to transform them into RDF. This is the purpose of the `SystemModelToRDFTransformer`. Variation point, variants, attributes and parameters form the feature tree and are transformed into an RDF datagraph. The variant rules form the extra constraints for the feature tree and are transformed into a SHACL shapes graph. Building the graphs is realised using the interfaces provided by DotnetRDF. DotnetRDF works by creating Triple objects consisting of 3 nodes objects and adding them to a graph instance.

Each element of the `SystemModel` implements the `IRDFSerializable` interface, which has the `ToRDF` function. This function is used to create all Triples necessary for the element and return them in as a list.

5.4.1 Feature tree

Generating the feature tree is fairly straight forward. We start with adding the variation points. We start by iterating over the Variation point list of the `SystemModel`. The variation point nodes themselves are added as triples `ex:root ex:VP ex:vpGUID` where the subject identifies a variation point connection, and the object is an URI with the unique identifier of the the variation point. We add a second triple `ex:root ex:vpGUID ex:vpGUID`. This is needed when a rule requires a specific path in the tree to be followed. Now we create the triples for the variants belonging to the variation point.

Variants are also added as `UriNodes` containing their GUID. We also add two triples that connect the variation point and the variant with the `ex:Variant` and `ex:variantGUID` predicates, where `variantGUID` is the actual GUID of the variant. For each variant, we first add their attributes. Each attribute has a value associated with it. This value is reached from the attribute node via the `ex:Value` predicate (`fiof:attribute1 ex:Value "value"`). Attributes are connected to their variant via their attribute name as predicate (`fiof:variant1 ex:attribute1Name fiof:attribute1`). We use the parameter name here, because we usually need to consider all attributes with a specific name in rules. Now we repeat this same process for all child variants of this variant. After adding the variants we also need to add any user defined parameters to the feature model graph. These are connected to the root node via the ex:parameter predicate (`ex:root ex:parameter fiof:param1`). Like attributes, parameters also have a value that is reached via the `ex:Value` predicate (`fiof:param1 ex:Value "value"`). Here is a complete example in turtle format:

```
ex:root ex:VP ft:CPU;
        vp:CPU ft:CPU.
ft:CPU ex:Variant ft:Ryzen_5.
ft:Ryzen_5 attr:CLOCK_SPEED ft:CLOCK_SPEED_Ry5;
           attr:SOCKET ft:SOCKET_Ry5.
ft:SOCKET_Ry5 ex:value "AM4".
ft:CLOCK_SPEEDRy5 ex:Value 3400.
ex:root ex:Parameter ft:maxPrice.
ft:maxPrice ex:Value 1200.
```

We have one variation point, *CPU*, with one variant, *Ryzen 5*. The variant has two attributes: clock speed, and socket with the values *"AM4"* and *3400* respectively. The predicate *attr:SOCKET* identifies the name of the attribute, whereas *ft:CLOCK* is the specific instance of the attribute. We also have one parameter, *maxPrice* in the model. The parameter is directly reachable from the root node.

5.4.2 Shapes Graph

Having established how the feature tree is built, now we will explain how to create the extra constraints on the tree. In this text, whenever the prefix `sh:` is used, this means a node that is included in the SHACL namespace. These nodes are instructions for the SHACL validator for how to interpret the shapes.

Constraints are split into two graphs: The condition graph which determines which actions should be evaluated, and the action graph containing these action shapes.

The condition of a rule object becomes a shape in the condition graph, and the action in the action graph. We start by describing the condition shapes.

Condition Shapes

We start by declaring the root node of the shape to be a SHACL shape node: `ex:shape a sh:NodeShape`. This instructs SHACL that this node has rules to be validated. Next, we set the target node for the shape to be the root of the feature tree by inserting the following triple:
`ex:shape sh:targetNode ex:root`. The logical expressions of the condition can be directly remapped to SHACL, as it has built in support for and, or and not operators. We use the `sh:or` predicate and a list of shapes as the object. This list contains the terms of the or clause. A term can be either another logical operator, or a shape node. An example for how an *or* triple looks like is: `ex:condshape sh:or (sh:and (...) ex:shapeNode).`

Now, we translate the `SelectionExpression`. This is realised using a property shape. The sh:path property of the shape is a list of connections to the variant we want to select. We use the `sh:hasValue` property to check that the node from the path is found. The path can be either a generic path of ex:Variant connections or a specific path to a node. To require the variant *Ryzen 5 3600*, we would use the following property shape:

```
[ a sh:PropertyShape;
  sh:path ( ex:VP [sh:zeroOrMorePath ex:Variant]);
  sh:hasValue ft:Ryzen_5_3600;
]
```

Condition shapes always start the actual expression with `sh:not`, because SHACL only reports failures in the report. So in order to get all fulfilled conditions, the expression has to be negated.

Lastly, we add the identifier for the action shape that should be evaluated if this condition is true.

A short but complete example for a condition shape with the condition `('CPU' == 'Ryzen 5 3600' && !('CPU' == 'Intel i7'))`:

```
ex:nodeshape a sh:NodeShape;
    sh:targetNode ex:root;
    sh:not [sh:and (co:Ryzen5Sel [sh:not co:Intel_i7Sel])];
    sh:message "message";
    ex:action ac:action1.

co:Ryzen5Sel a sh:PropertyShape;
    sh:path ([sh:zeroOrMorePath ex:Variant]);
    sh:hasValue ft:Ryzen_5_3600;
co:Intel_i7Sel a sh:PropertyShape;
    sh:path ([sh:zeroOrMorePath ex:Variant]);
    sh:hasValue ft:Intel_i7;
```

Action Shapes

For action shapes the general structure of the shape with the target node and the like stays the same as for the condition shapes. The comment that was either defined for the rule or automatically generated is added to the action shape using the `sh:message` predicate. The logical expression part of the actual rule is also the same. The difference comes with the actual actions, the leaves of the expression tree.

The first type of action we define is the simple requirement for a specific variant. This is generated in the same way as the selection property shape from the condition graph.

Next, we have attribute actions. These can have several different forms. Attribute actions without any operators or modifiers attached are treated the same as variant actions. We simply add the attribute name to the end of the path of the selection shape.

For attributes with an operator and a constant as an operand we need to differentiate between which of the `each` and `any` modifiers has been applied. For the `each` modifier, we simply create a property shape with a path to where in the tree we want to consider the attribute. Then, we simply check this condition against any of the attribute with the right name we find. This is done via the `sh:maxInclusive`, `sh:minInclusive` and `sh:hasValue` predicates. The shape for the `any` modifier looks the same with the exception that we also add the triple: `[sh:minCount 1]` triple to produce a rule violation if no variant with this attribute is selected.

```
ac:actionShape a sh:NodeShape;
    sh:targetNode ex:root.
     sh:or ([sh:path (vp:Memory ex:Variant* attr:MEM_TYPE) ;
              sh:hasValue "DDR4" ;
              sh:minCount 1 ]);
    ex:targetIdentifier "Memory::MEM_TYPE";
    sh:message "Memory modules are required to be DDR4.";
```

If the operand is a parameter, then we need to use a SPARQL query to obtain the value for comparison, as there is no way to specifically find the parameter value in base SHACL. We can use a SPARQL query in SHACL by using the `sh:sparql` predicate with the query string as the object.

```
SELECT $this ?value WHERE {
    $this a ex:configuration.
    param:operandName ex:Value ?paramvalue.
    $this ((ex:Variant*/attr:attrName>/ex:Value)) ?value.
    FILTER(?paramvalue < ?value)
}
```

For a *sum* action we also need to use SPARQL, as the base SHACL functions do not support aggregation functions like this. The query we use is the following:

```
SELECT $this WHERE {
$this a ex:configuration.
$this ((ex:Variant*/attr:atName}/ex:Value) | ex:zero ) ?attr.
}
group by $this
having(SUM(?attr) < $value$ )
```

In case the operand is a parameter, we need to modify the query like this:

```
SELECT $this ?value WHERE {
    $this a ex:configuration.
    param:paramName ex:Value ?param.
    $this ((ex:Variant*/attr:attributeName/ex:Value)
                | ex:zero ) ?attr.
}
group by $this ?value
having(SUM(?attr) < ?param)
```

5.5 Validating the Model

The `FeatureModelValidator` is the class that handles the actual validation of the feature model. First of all, the user defined parameters get updated with their newest values. For each parameter, we simply need to remove the value triple from the feature graph, and insert the same triple with the new value.

Now the `validateFeatureModel` method is called with the list of currently selected variants as parameter. We need to update the model to reflect this selection. Selections are reflected in the feature model by their connections with their parent node. This is a triple in the form of:

```
ex:parentvariant ex:Variant ex:childvariant.
```

First we remove these triples from the graph. Then we create triples for each selected variant and add them.

Now, we can evaluate the conditions for our constraints. This is simply done by validating the feature model graph with the condition shapes graph. The `ShapesGraph` class of DotnetRDF has a `Validate` method for this purpose. This returns a report containing all shapes that violated their rules. For us, these are all conditions that are fulfilled, because we invert them in the shapes graph. The shapes have a triple that points to the action that has to be evaluated when the condition is true.

We now need to evaluate all previously obtained action shapes. By default all action shapes are deactivated using the `sh:deactivated` predicate. This causes the SHACL validator to ignore them during evaluation. We remove this deactivated triple for all action nodes that should be evaluated.

Evaluation is, as before, simply calling the `Validate` method of the action shape graph. If there are any violated constraints they are listed in the report returned from `Validate`. We now create a List of validation results that contain information important to inform the user of why the constraint is violated. This result included the configuration container the constraint is from, the original constraint text, and the message. This list is returned and it's content displayed in a form.

6 Building a Model

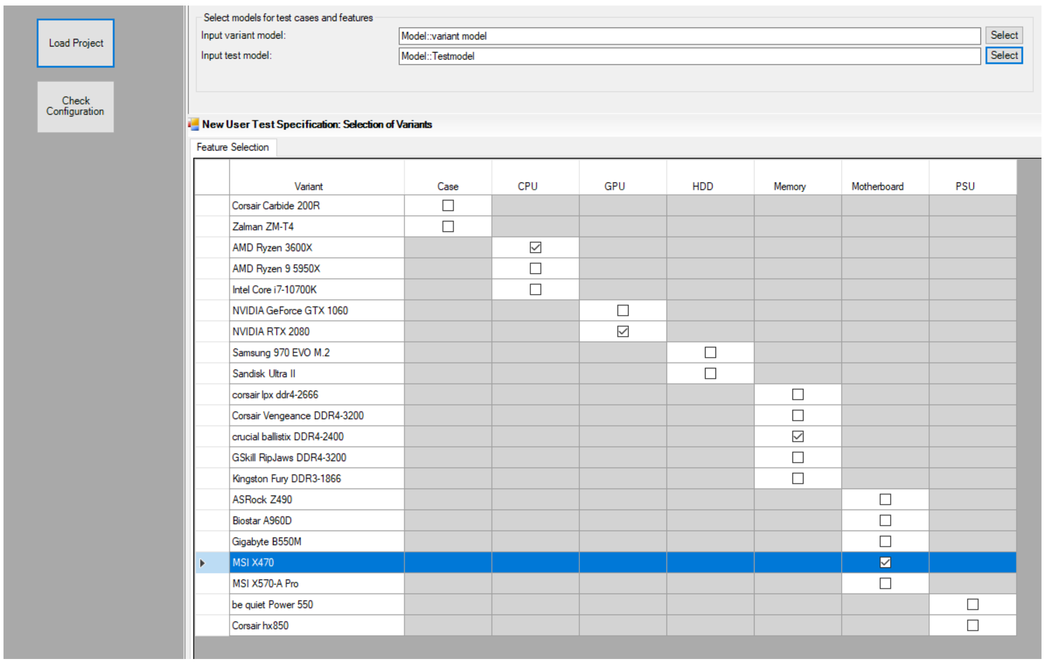
In this chapter we will take a look at how to build a feature model from the ground up from a user's perspective. After having defining the system formally and discussing the implementation, this chapter should give an overview over the actual features the system has. We will go over how to create a project, create variation points, variants and attributes. Then take a look at how to write constraints using configuration containers. Figure 6.1 shows the interface for selecting features.

6.1 Model Creation

Before creating the actual model we need to setup an enterprise architect project. Simply create an empty project from the file menu. Now create a root package with any name. This will be the package the feature model is in. Inside it create two more packages: one for the variant information, and one for the rules.

Variation Points

Now that we have an empty model we can start defining the variation points of our feature model. To create a variation point, create a new element in the variant model package and give it the stereotype *VariationPoint*.



Select models for test cases and features

Input variant model: Model::variant model

Input test model: Model::Testmodel

New User Test Specification: Selection of Variants

Variant	Case	CPU	GPU	HDD	Memory	Motherboard	PSU
Corsair Carbide 200R	☐						
Zalman ZM-T4	☐						
AMD Ryzen 3600X		☑					
AMD Ryzen 9 5950X		☐					
Intel Core i7-10700K		☐					
NVIDIA GeForce GTX 1060			☐				
NVIDIA RTX 2080			☑				
Samsung 970 EVO M.2				☐			
Sandisk Ultra II				☐			
corsair lpx ddr4-2666					☐		
Corsair Vengeance DDR4-3200					☐		
crucial ballistix DDR4-2400					☑		
GSkill RipJaws DDR4-3200					☐		
Kingston Fury DDR3-1866					☐		
ASRock Z490						☐	
Biostar A960D						☐	
Gigabyte B550M						☐	
MSI X470						☑	
MSI X570-A Pro						☐	
be quiet Power 550							☐
Corsair hx850							☐

Figure 6.1: A screenshot of the user interface for selecting features

Variants

Now that the variation points are created we can create the variants. Variants always belong to one variation point. To define the in Enterprise architect, we add them as child elements of their variation point. Select the variation point you want to add a variant to and create a new element. Give the element the *Variant* stereotype.

Attributes

Are key value pairs that are used to give additional information to variants. To define an attribute, go to the property page of the variant you want to add it to, and then go to tagged values. Add a new tagged value and give it the name and value you want it to have.

6.2 Creating Rules

Now that we have a base feature model, we can add restrictions to variants. Restrictions are defined in a textual grammar inside so called configuration containers. Configuration containers are defined as elements with the *ConfigurationContainer* stereotype. They can be either added in their own package, or as a child element of a variant. They act the same except that containers of variants will have the condition that the variant is selected on any rule. Rules are written in the description text of the configuration container. There can be any number of rules on a container. Rules must be written inside a rule body:

```
#rules {
  ...
}
```

A rule can have different forms. First we have simple if-then rules, where there is a condition, and if the condition fulfilled the statements in the body are evaluated. Then we have requirements, for which the statements inside the body are always executed. This is used for constraints that must always hold. For example a minimum count for a variation point, or a feature that has to be included in any configuration. Finally, we have variant rules, which are rules that can only be defined in configuration containers that are created as child elements of a feature. The statements in the body of a variant rule only get evaluated when the parent feature of the configuration container is included in the configuration. The statements inside the body have the same form for all rule types.

```
#rules {
  #if ('CPU'=='Ryzen 5') {
    ...
  }
  #requirement {
    ...
  }
  #variantrule {
    ...
  }
}
```

Before we talk about the rule bodies, we will briefly discuss conditions for if-then rules. The condition in the above example means the feature *Ryzen 5* that is a child of *CPU* is included in the configuration. Conditions can have combinations of logical operators to make them more expressive:

```
    (!A && (B || C))
```

Here, A, B, and C are short for an expression like in the previous example.

Next we will talk about the rule bodies. Each of body can have an arbitrary number of statements or actions inside it. Each action is delimited by a semicolon. Actions begin with an action type and then the actual constraint to be evaluated. Types are either *#require* or *#restrict*, require meaning the action has to evaluate to true, restrict the opposite. For the constraint part we first have simple feature constraints, either requiring or excluding a specific feature from the configuration.

```
#if ('CPU' == 'Ryzen 5') {
  #require <Variant>'Gigabyte B550';
  #restrict <Variant>'MSI Z18';
}
```

Next we have attribute constraints. Here we can specify an operator and a value to check against. The value can be either a string, an integer, or an identifier for a parameter. The actual value for a parameter can be set in the GUI during configuration. The *#each* modifier means that all attributes with the given name in the configuration adhere to the constraint. The*#any* modifier means that additionally, at least one such attribute has to be included in the configuration.

```
#if ('CPU' == 'Ryzen 5') {
  #require #any <Attribute>'Motherboard::CPU_SOCKET' == "AM4";
  #require <Attribute>'Motherboard::POWER_CONSUMPTION' < 100;
  #require #each <Attribute>'Motherboard::PRICE' == $MotherboardPrice;
}
```

Lastly, instead of just an attribute we can also use the same over all instance of an attribute in the configuration:

```
#requirement {
  #require sum('POWER_CONSUMPTION') < 1000;
  #require sum('PRICE') < $maximumPrice;
}
```

Any of these constraints can be combined using logical operators, the same way conditions for if-then rules can.

7 Evaluation and Analysis

In this chapter we will evaluate our system. We will first give an evaluation of the performance of the model. We give an evaluation on a complete feature model spanning the entire feature set of our model validation system, and then evaluate the features by themselves. For an overview over what features the system has, see Chapter 6.

7.1 Performance Evaluation

The performance evaluation will be split into two parts:

- Feature model of PC parts configuration introduced previously
- Evaluation of a big feature model with Conjunctive Normal Form (CNF) clauses as rules provided by SPLOT
- Evaluation of attributes, parameters and sum function

7.1.1 Setup

We start with the more realistic but small feature model of a PC configuration. An explanation for the model can be found in Chapter 4.2.1. For this model, we will evaluate the time the system takes to build the model, which includes reading the input model from Enterprise Architect, compiling the rules, and transforming the model to RDF. We then evaluate the time needed to validate feature selections. The model has 21 features and 31 rules. The rules cover all capabilities of the system. An excerpt of the model can be found in the appendix.

Software Product Lines Online Tools (SPLOT) [32] provides multiple test models designed for empirical evaluation of feature models. SPLOT provides multiple models with similar structure that have a varying number of features and constraints. We will use multiple of these models to see how model variations impact the performance of our system. The constraints of the SPLOT models are 3-CNF formulas. It has been shown that 3-CNF formulas are often harder to solve for these systems than realistic feature models, which makes them a good candidate for benchmarking [36]. Below is a small excerpt of one of the models, showing a part of the feature tree, and a rule consisting of a condition shape and an action shape.

```
ex:root ex:VP fm:o_3,
              fm:m_5,
         ex:true schema:true;
         ex:zero 0 ;
         vp:m_5 fm:m_5;
         vp:o_3 fm:o_3;
         a ex:configuration.
fm:o_3 a fm:VP;
       ex:Variant fm:g_2_3;
       ex:Variant fm:g_2_4;
fm:m_5 a fm:VP;
       ex:Variant fm:m_3_1;
fm:g_2_3 a fm:variant.
fm:g_2_4 a fm:variant.
fm:m_3_1 a fm:variant.


schema:g_2_4Selected a sh:PropertyShape;
      sh:hasValue fm:g_2_4;
      sh:path (ex:VP ex:Variant).

schema:conditionShape-1 a sh:NodeShape;
    sh:not [sh:or (schema:g_2_4Selected schema:g_2_3Selected)];
    sh:targetNode ex:root.
    schema:targetAction schema:actionShape-1.
```

```
schema:m_3_1Selected a sh:PropertyShape>;
      sh:hasValue fm:m_3_1;
      sh:path (ex:VP ex:Variant).

schema:actionShape-1 a sh:NodeShape>;
    sh:targetNode ex:root;
    sh:message "cnf action";
    sh:or (schema:m_3_1Selected).
```

The SPLOT system uses the SXFM format to save feature models, therefore we have to convert this format to one we can use. The SPLOT team has provided a Java library for parsing the format, which we use to transform the model and the constraints to XML in a form we can use to create our feature model. The feature model part is transformed so that the first level of features are variation points, and all children are variants of these variation points. The constraint part stays mostly the same, except that we remove the prefix of each line.

The built-in XML functionality of C# is used for parsing the model and creating the RDF graph and shapes graph. We model the constraints such that the first two terms of a 3-CNF clause form the condition for a rule, and the third term is the action.

After creating the feature model, we can now test the performance. We test the performance of our model by randomly selecting features and measuring the time it takes to validate the model. We average this time for each model over 50 iterations.

All tests are conducted on a Lenovo ThinkPad P51, Intel i7-7820HQ(2.9 GHz).

7.1.2 Extended Constraints

To evaluate the extended features of our model we will automatically generate multiple models with a varying number of features and constraints.

Extended features are attributes of variants, user defined parameters, and the sum constraint.

To start with we just build models with a single feature and gradually increase the number of rules. Each feature of the model has the attribute we test on, and every condition is true. This means every action has to be evaluated. This is not a realistic model, but it should give us a first impression for how these constraints scale.

After that we analyse multiple models where we fix the number of features and increase the number of constraints. We build multiple of these models for each of the constraint type. Each of the types will have one model with 10, 100, 500 and 1000 features, respectively. The number of constraints will be varied from 10 up to 10000 for each model. For evaluation we select a varying number of random features and average over the validation times. We performed 50 iterations per model except for the largest ones which for which 10 were performed.

```
ex:root ex:VP fm:vp1;
      ex:true schema:true;
      ex:zero 0 ;
      vp:vp1 fm:vp1;
fm:vp1 ex:Variant fm:variant1;
       a fm:VariationPoint.
fm:variant1 attr:attribute1 fm:attribute1;
                        a fm:variant.
fm:attribute1 ex:Value 1;


schema:actionShape-1 sh:targetNode ex:root;
     a sh:NodeShape>;
     sh:message "attribute constraint";
     sh:or ([sh:path ex:VP ex:Variant attr:attribute1 ;
             sh:maxExclusive 50  ;
             sh:minCount 1 ]).


schema:actionShape-1 sh:targetNode ex:root.
     a sh:NodeShape;
     sh:message "sum constraint";
     sh:or ([sh:sparql [sh:select """
    SELECT $this WHERE {
        $this a ex:configuration.
        $this ((ex:VP/ex:Variant/attr:attribute1/ex:Value)
                    | ex:zero ) ?attr.
    }
    group by $this
    having(SUM(?attr) >= 100) """ ]]).


schema:actionShape-1 sh:targetNode ex:root;
     a sh:NodeShape;
     sh:message "parameter constraint";
     sh:or ([sh:path (ex:VP ex:Variant* attr:attribute1 ex:value) ;
             sh:sparql [sh:select> """
    SELECT $this ?value WHERE {
       $this a ex:configuration.
       fm:param1 ex:Value ?paramvalue.
       $this
         ((ex:VP/ex:Variant/attr:attribute1/ex:Value)) ?value.
          FILTER(?paramvalue < ?value)
    } """ ]]).
```

7.1.3 Results

PC building model

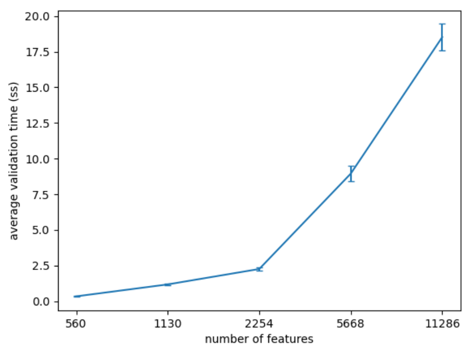
We start with the evaluation of the model building. Reading the model elements from Enterprise Architect took up the majority of the time with an average of 1581 ms and a standard deviation of 172 ms. Compiling the configuration containers using ANTLR took an average of 403 ms with an 86 ms standard deviation, and transforming the model to RDF required an average 124 ms with a 7 ms standard deviation.

The average for validation over all validation runs with different features selections is 102 ms with a standard deviation of 21 ms. The time for evaluating conditions and actions is somewhat equal with a mean of 51 ms and a standard deviation of 12 ms for conditions and a mean of 52 ms with a standard deviation of 15 ms for actions. The minimum was 84 ms and the maximum 190 ms. The minimum had 15 conditions evaluating to true, producing 6 faults. The maximum had 28 conditions evaluating to true, producing 13 violations. The time for updating the selected features in the graph and the active actions is negligible being at maximum 1ms.

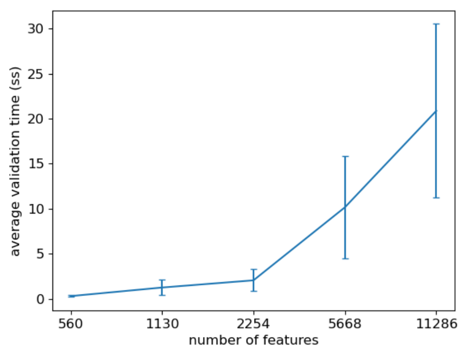
CNF Model Results

For the models from SPLOT, we evaluated 5 models which consisted of 500, 1000, 2000, 5000 and 10000 features respectively. The number of constraints varied between 50 and 150. For the evaluation, the time the system required to validate a configuration was measured and averaged over multiple iterations. Both the number of features that were in the configuration and which features are in it were randomized in each iteration. Figure 7.1a shows the results of this evaluation. Every point in the graph is the average time it took the SHACL validator to validate the configuration with random assignments of features. The time it takes to validate a model stays within a reasonable frame up until 2000 features. At 5000 features the time is already above 10 seconds, and passes 20 second at 10000 features.

The time for validation is a combination of the time it takes for validation of condition shapes and, the amount of active action shapes. Here we have conditions that consist of an or clause of two feature selection shapes, and actions consisting of one feature selection shape. The way the shapes work is that at least one node in the feature tree, found using the specified path has to be the specified node. This means that if a feature is not selected, the entire the entire feature tree needs to be checked for this node, whereas if the node is found, the validator can return immediately. We can observe that the majority of the validation time is spent on validating the condition shapes. This is natural, as condition as have twice as many feature selection shapes per constraint as action shapes. Additionally, every condition shape



(a) Fixed number of selected features



(b) Variable number of selected features

Figure 7.1: Results of running validation on SPLOT models

needs to be validated, whereas only the action shapes of fulfilled conditions are considered.

A second test was conducted with the same models, this time the number of features in the configuration was kept constant at half the features of the model for all iterations. The actual selected features were still randomized for in every iteration. The results for this test run can be seen in Figure 7.1b. While the mean time between the two test runs remains about equal, the standard deviation is very high when varying the amount of selected features. The validation time increases linearly with the amount of selected features, as the number of paths in the feature graph that the validator has to consider, also increases. Naturally, this causes the standard deviation to increase. We can see this increase for a different model in Figure 7.2

Extended Feature Results

We start with the model that has only one feature and we increase the number of constraints in each step. Unsurprisingly, this model shows a linear increase in the validation time with the number of rules for all three model types. This is because there is only one path in the feature tree that will reach an attribute value node. For the attribute constraint model, the validator follows the path specified in the property shape, and evaluates the found nodes against the SHACL properties of the property shape. Since there is only one path, the validator can simply iterate over all constraint shapes, get this one attribute value and evaluate, leading to a linear increase in validation time, with the increase in constraints. For the sum and parameter models, the process is similar, but here the attribute values are found in a SPARQL query, which also handles the evaluation by filtering out the valid nodes. Validation of SPARQL queries is slower, but the increase in validation time still stays linear. The average time per rule validation is roughly 7 ms for a sum constraint, 2 ms for an attribute constraint, and 4ms for a parameter constraint.

Next, the results from varying the number of constraints with a fixed amount of features. This was done separately for all 3 constraint types. We can see from Figure 7.3 that above 1000 constraints is the point where performance starts to decrease. Attribute constraints perform better than sum and parameter constraints. This is because sum and parameter constraints use a SPARQL query for validation, which has to be interpreted in addition. Note that the models used for evaluation represent extreme cases in which each of the features in the model, has the attribute we check for. With more realistic models where not every feature has every attribute, the number of paths in the feature tree SHACL has to check remains the same, while the number of actual nodes to check against properties is lower.

We can see from Figure 7.3 that for models with up to around 500 features, the system performs at an acceptable level. As the number of features and rules increases, the time it takes to validate a configuration quickly becomes too high to be reasonably used. We can clearly see the increase in validation time caused by interpretation of SPARQL queries when comparing the attribute and parameter plots. The constraints in both models are similar, but parameter constraints compare attributes to a parameter stored in the feature model instead of a constant. This requires the validation to be done using a SPARQL query. The actual comparisons are the same between the two constraint types otherwise.

The validation time correlates with the number of selected features in the configuration. The higher the number of features, the more paths exist in the feature tree which the validator has to follow to find the right value nodes to check. This means that, in general, the more features are selected,

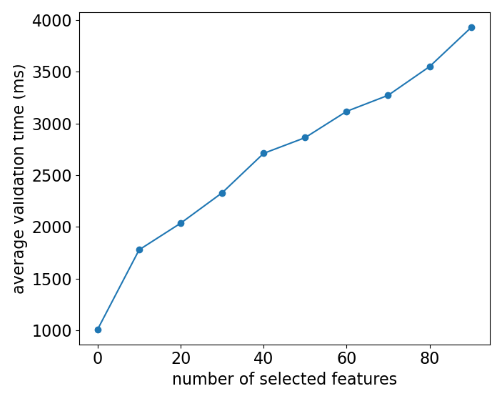
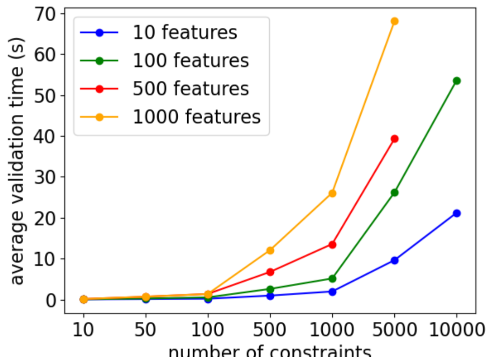
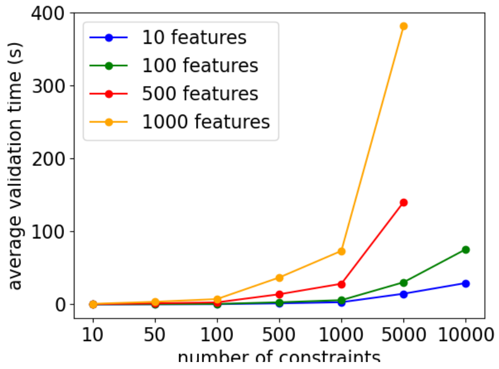


Figure 7.2: Validation times for against number of selected features

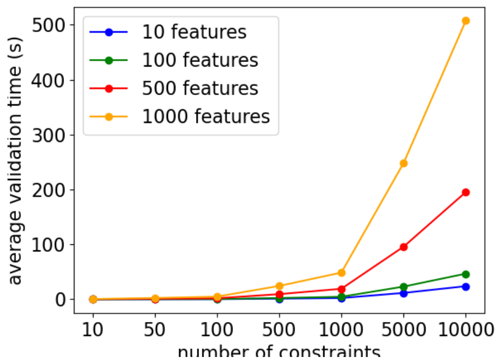
the longer the evaluation takes on average. Take for example the model consisting of only attribute constraints with 500 constraints and 100 features from this evaluation. We can observe a linear increase in validation time with the number of selected features (see Figure 7.2).



(a) Attributes



(b) Parameter



(c) Sum

Figure 7.3: Average time for model validation using models with advanced constraints

8 Conclusion and Outlook

8.1 Outlook

We showed that the system performs well for smaller feature models however there do exist larger models in practice. The feature model for the eCos operating system has roughly 1000 features [37]. The model uses if-then rules, with feature selection constraints. Considering our benchmark of a pure CNF feature model, this model should still perform well.

The feature model for the Linux kernel is comprised of around 6000 features. With our current implementation, validating a selection in this model would most likely take more than 10 seconds, which would make the validation process cumbersome in practice. We can look for ways to increase performance of our system. Currently all validation is done using the DotnetRDF implementation of SHACL. There are multiple other implementations, which differ in performance. We currently also use the standard in-memory graph storage. Performance could possibly be improved by switching to a specialized graph database like GraphDB [38].

Many product configuration systems support automatically inferring configuration decisions where there is only one valid option. OWL based feature models often use the inherent power of reasoning to make these decisions. In our case, we could possibly make use of SHACL rules to infer configurations. SHACL rules allow the construction of new triples in a graph, according to a SPARQL query.

8.1.1 Limitations of SHACL

Semantic Web technologies are powerful tools to build models and also for validation, but they do have some limitations when it comes to more complex validation tasks, which can make it difficult to build constraints in some cases.

SHACL-SPARQL

For cases where the built in SHACL properties are not sufficient to validate the model, SHACL allows the creation of SPARQL queries for creating custom validation rules. This is often useful when needing to select nodes for comparisons that aren't direct properties of the focus node. It also enables the use of aggregate functions like sum and count, and use filter expressions.

SPARQL queries for SHACL shapes need to adhere to certain conditions [6]. For example, the first returned variable of the result set always has to be the focus node of the node shape the query is defined on. All subqueries also need to return this node. Queries cannot have MINUS or VALUES clauses. This means we cannot constraint variable bindings to certain values, and cannot exclude the results from one query from the results of another.

Recursive Constraints

SHACL allows a shape to define property shapes via the sh:property predicate. These shapes are used to validate all nodes that are found along a specified path from the focus node, according to the properties defined in the shape. A node shape can have multiple property shapes, but property shapes cannot be defined recursively. This is important, as SHACL favors the definition of constraints that reference other constraints [39].

8.2 Conclusion

In this thesis we approached the problem of product configuration by using Semantic Web technologies. We do so by building a feature model using SHACL. Feature models have been built using OWL before, but there has not been much research into using SHACL, in part because SHACL is a relatively new standard [6].

We developed a feature model using RDF to model the feature tree, and SHACL to model extra constraints on the tree. Additionally, a domain specific language was developed to enable the practical construction of these constraints. The SHACL specification provides functionality that is needed to validate these extra constraints. An added benefit of using the Semantic Web is that the model is platform independent, as Semantic Web technologies are implemented in many different languages and operating systems.

The system handles feature models of small to medium size relatively well. We can have several hundred features and rules, with validation of the configuration still being fast. For larger models performance starts to decrease considerably. We showed that our approach is still performant on models with several hundred features and rules. Models that are purely made of feature selection constraints in logical formulas perform better than models with sum and attribute constraints. With larger feature models of up to around 2000 features the system still performs well.

Bibliography

[1] Li Da Xu, Eric L Xu, and Ling Li. “Industry 4.0: state of the art and future trends.” In: *International Journal of Production Research* 56.8 (2018), pp. 2941–2962 (cit. on p. 1).

[2] M Reza Abdi and AW Labib. “Grouping and selecting products: the design key of reconfigurable manufacturing systems (RMSs).” In: *International journal of production research* 42.3 (2004), pp. 521–546 (cit. on p. 1).

[3] Abdelrahman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Ho. “Knowledge Based Method to Validate Feature Models.” In: vol. 2. Jan. 2008, pp. 217–225 (cit. on pp. 1, 5).

[4] Don Batory. “Feature models, grammars, and propositional formulas.” In: *International Conference on Software Product Lines*. Springer. 2005, pp. 7–20 (cit. on pp. 1, 37).

[5] Schreiber Guus and Raimond Yves. *Resource Description Framework (RDF)*. 2014. URL: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/ (cit. on pp. 2, 8, 9).

[6] Knublauch Holger and Dimitris Kontokostas. *Shapes Constraint Language (SHACL)*. 2017. URL: https://www.w3.org/TR/shacl/ (cit. on pp. 2, 13, 80, 81).

[7] *Understand how structured data works*. 2021. URL: https://developers.google.com/search/docs/guides/intro-structured-data (cit. on p. 2).

[8] Shakeel Ahmad Khan and Rubina Bhatti. “Semantic Web and ontology-based applications for digital libraries.” In: *The Electronic Library* (2018) (cit. on pp. 2, 8).

[9] Pieter Pauwels, Sijie Zhang, and Yong-Cheol Lee. "Semantic web technologies in AEC industry: A literature overview." In: *Automation in Construction* 73 (2017), pp. 145–165 (cit. on p. 2).

[10] Núria Queralt-Rosinach et al. "DisGeNET-RDF: harnessing the innovative power of the Semantic Web to explore the genetic basis of diseases." In: *Bioinformatics* 32.14 (2016), pp. 2236–2238 (cit. on pp. 3, 8).

[11] Maxwell Lewis Neal et al. "Harmonizing semantic annotations for computational models in biology." In: *Briefings in bioinformatics* 20.2 (2019), pp. 540–550 (cit. on pp. 3, 8).

[12] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. "Automated analysis of feature models 20 years later: A literature review." In: *Information systems* 35.6 (2010), pp. 615–636 (cit. on pp. 6, 7).

[13] Uzma Afzal et al. "Feature Selection Optimization in Software Product Lines." In: *IEEE Access* PP (Sept. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.3020795 (cit. on pp. 6, 35).

[14] Guillaume Bécan et al. "Synthesis of attributed feature models from product descriptions." In: *Proceedings of the 19th International Conference on Software Product Line*. 2015, pp. 1–10 (cit. on pp. 7, 37).

[15] Andreas Classen, Quentin Boucher, and Patrick Heymans. "A text-based approach to feature modelling: Syntax and semantics of TVL." In: *Science of Computer Programming* 76.12 (2011), pp. 1130–1143 (cit. on pp. 7, 37).

[16] Maxime Cordy et al. "Beyond boolean product-line model checking: dealing with feature attributes and multi-features." In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 472–481 (cit. on pp. 7, 37).

[17] Tim Berners-Lee, James Hendler, and Ora Lassila. "The semantic web." In: *Scientific american* 284.5 (2001), pp. 34–43 (cit. on p. 8).

[18] Hitzler Pascal and Krötzsch Markus. *OWL 2 Web Ontology Language Primer*. 2012. URL: https://www.w3.org/TR/2012/REC-owl2-primer-20121211 (cit. on pp. 8, 10).

[19] Ramanathan V Guha. "Light at the end of the tunnel." In: *International semantic web conference*. Vol. 12. 2013 (cit. on p. 8).

[20] K Selçuk Candan, Huan Liu, and Reshma Suvarna. "Resource description framework: metadata and its applications." In: *ACM SIGKDD Explorations Newsletter* 3.1 (2001), pp. 6–19 (cit. on p. 8).

[21] Dan Brickley and R.V. Guha. *RDF Schema 1.1*. 2014. URL: `https://www.w3.org/TR/rdf-schema/` (cit. on p. 10).

[22] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. 2008. URL: `https://www.w3.org/TR/rdf-sparql-query/` (cit. on p. 11).

[23] Knublauch Holger, Hendler James A., and Idehen Kingsley. *SPIN - Overview and Motivation*. 2011. URL: `https://www.w3.org/Submission/spin-overview` (cit. on p. 13).

[24] Tomaž Kosar et al. "A preliminary study on various implementation approaches of domain-specific language." In: *Information and software technology* 50.5 (2008), pp. 390–405 (cit. on p. 15).

[25] Parr Terence. *ANTLR*. 2014. URL: `https://www.antlr.org/` (cit. on p. 16).

[26] Shamim Ripon et al. "Semantic web based analysis of product line variant model." In: *International Journal of Computer and Electrical Engineering* 6.1 (2014), pp. 1–6 (cit. on pp. 18–22).

[27] Krzysztof Czarnecki et al. "Feature models are views on ontologies." In: *10th International Software Product Line Conference (SPLC'06)*. IEEE. 2006, pp. 41–51 (cit. on p. 18).

[28] Shusheng Zhang, Weiming Shen, and Hamada Ghenniwa. "A review of Internet-based product information sharing and visualization." In: *Computers in Industry* 54.1 (2004), pp. 1–15 (cit. on p. 19).

[29] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. "Applying semantic web technology to feature modeling." In: *Proceedings of the 2009 ACM symposium on Applied Computing*. 2009, pp. 1252–1256 (cit. on pp. 21, 22).

[30] Horrocks Ian and Boley Harold. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. 2004. URL: `https://www.w3.org/Submission/SWRL/` (cit. on p. 22).

[31] Marcilio Mendonca and Donald Cowan. “Decision-making coordination and efficient reasoning techniques for feature-based configuration.” In: *Science of Computer Programming* 75.5 (2010), pp. 311–332 (cit. on pp. 23, 24).

[32] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. “Automated reasoning on feature models.” In: *International Conference on Advanced Information Systems Engineering*. Springer. 2005, pp. 491–503 (cit. on pp. 26, 70).

[33] Stefan Bischof et al. “Integrating Semantic Web Technologies and ASP for Product Configuration.” In: (Oct. 2018) (cit. on pp. 27, 28, 31).

[34] Marcilio Mendonca, Moises Branco, and Donald Cowan. “SPLOT: software product lines online tools.” In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009, pp. 761–762 (cit. on p. 28).

[35] *SXFM Format*. URL: `http://ec2-54-213-92-199.us-west-2.compute.amazonaws.com:8080/SPLOT/sxfm.html` (cit. on p. 29).

[36] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. “SAT-based analysis of feature models is easy.” In: *Proceedings of the 13th International Software Product Line Conference*. 2009, pp. 231–240 (cit. on p. 70).

[37] Jianmei Guo. *Feature models in the wild*. 2015. URL: `https://gsd.uwaterloo.ca/feature-models-in-the-wild.html` (cit. on p. 79).

[38] *GraphDB*. 2020. URL: `https://graphdb.ontotext.com/` (cit. on p. 79).

[39] Julien Corman, Juan L. Reutter, and Ognjen Savković. “Semantics and Validation of Recursive SHACL.” In: *The Semantic Web – ISWC 2018*. Ed. by Denny Vrandečić et al. Cham: Springer International Publishing, 2018, pp. 318–336. ISBN: 978-3-030-00671-6 (cit. on p. 80).

Appendix

PC model

Feature Tree

```
ex:root ex:VP fm:Motherboard,
                fm:cpu,
                fm:gpu,
                fm:memory;
          ex:param fiof:price,
                   fiof:clock_speed;
          ex:true schema:true;
          ex:zero 0 ;
          vp:motherboard fm:motherboard;
          vp:cpu fm:cpu;
          vp:gpu fm:gpu;
          vp:memory fm:memory.

fm:motherboard fm:name "Case";
        ex:Variant fm:gigabyte-b550m.
fm:cpu ex:Variant fm:ryzen-5-3600.
fm:gpu ex:Variant fm:rtx-2080.
fm:memory ex:Variant fm:kingston-ddr3-1600.

fm:kingston-ddr3-1600 attr:CLOCK_RATE fm:kingston-ddr3-1600-CLOCK_RATE;
                      attr:POWER_CNSMPT fm:kingston-ddr3-1600-POWER_CNSMPT;
                      attr:PRICE fm:kingston-ddr3-1600-PRICE;
                      attr:SLOT fm:kingston-ddr3-1600-SLOT;
                      fm:name "GSkill RipJaws DDR4-3200";
                      a fiof:variant.
fm:kingston-ddr3-1600-CLOCK_RATE fm:name "CLOCK_RATE";
```

```
                                ex:Value 3200 .
fm:kingston-ddr3-1600-CONNECTOR fm:name "CONNECTOR";
                               ex:Value "DDR3" .
fm:kingston-ddr3-1600-POWER_CNSMPT fm:name "POWER_CNSMPT";
                                       ex:Value 6 .
fm:kingston-ddr3-1600-PRICE fm:name "PRICE";
                           ex:Value 118 .
fm:kingston-ddr3-1600-SLOT fm:name "SLOT";
                          ex:Value "DDR4".

fm:hdd  fm:name "HDD";

fm:rtx-2080 attr:POWER_CNSMPT fm:rtx-2080-POWER_CNSMPT;
            attr:PRICE fm:rtx-2080-PRICE;
            fm:name "NVIDIA RTX 2080";
fm:rtx-2080-POWER_CNSMPT fm:name "POWER_CNSMPT";
                             ex:Value 400 .
fm:rtx-2080-PRICE fm:name "PRICE";
                  ex:Value 1025 .
fm:rtx-2080-SLOT fm:name "SLOT";
                 ex:Value "PCIE".

fm:kingston-ddr3 attr:POWER_CNSMPT fm:kingston-ddr3-POWER_CNSMPT;
                 attr:PRICE fm:kingston-ddr3-PRICE;
                 attr:SLOT fm:kingston-ddr3-SLOT;
                 fm:name "Kingston Fury DDR3-1866";
fm:kingston-ddr3-POWER_CNSMPT fm:name "POWER_CNSMPT";
                                   ex:Value 4 .
fm:kingston-ddr3-PRICE  fm:name "PRICE";
                        ex:Value 70 .
fm:kingston-ddr3-SLOT fm:name "SLOT";
                      ex:Value "DDR3".

fm:gigabyte-b550m attr:CPU_SOCKET fm:gigabyte-b550m-CPU_SOCKET;
                  attr:FORM_FACTOR fm:gigabyte-b550m-FORM_FACTOR;
                  attr:POWER_CNSMPT fm:gigabyte-b550m-POWER_CNSMPT;
                  attr:PRICE fm:gigabyte-b550m-PRICE;
                  fm:name "Gigabyte B550M";
fm:gigabyte-b550m-CPU_SOCKET fm:name "CPU_SOCKET";
                             ex:Value "AM4".
fm:gigabyte-b550m-FORM_FACTOR fm:name "FORM_FACTOR";
                              ex:Value "mATX".
fm:gigabyte-b550m-POWER_CNSMPT fm:name "POWER_CNSMPT";
                                    ex:Value 40 .
fm:gigabyte-b550m-PRICE  fm:name "PRICE";
                         ex:Value 90 .

fm:ryzen-5-3600 attr:CLOCK_SPEED fm:ryzen-5-3600-CLOCK_SPEED;
                attr:POWER_CNSMPT fm:ryzen-5-3600-POWER_CNSMPT;
                attr:PRICE fm:ryzen-5-3600-PRICE;
                attr:SOCKET fm:ryzen-5-3600-SOCKET;
                fm:name "AMD Ryzen 5 3600X";
fm:ryzen-5-3600-CLOCK_SPEED fm:name "CLOCK_SPEED";
                            ex:Value 3800 .
fm:ryzen-5-3600-POWER_CNSMPT fm:name "POWER_CNSMPT";
                                  ex:Value 95 .
fm:ryzen-5-3600-PRICE fm:name "PRICE";
                      ex:Value 217 .
fm:ryzen-5-3600-SOCKET fm:name "SOCKET";
                       ex:Value "AM4".

fm:corsair-hx850 attr:PRICE fm:corsair-hx850-PRICE;
                 fm:name "Corsair hx850";
fm:corsair-hx850-POWER fm:name "POWER";
                       ex:Value 850 .
fm:corsair-hx850-PRICE fm:name "PRICE";
                       ex:Value 216 .
```

Condition Shapes

```
schema:corsair-hx850Selected a sh:PropertyShape;
      sh:hasValue fm:corsair-hx850;
      sh:path (ex:VP ex:Variant).

schema:alwaysTrueShape schema:targetAction schema:actionShape-1;
                        schema:targetAction schema:actionShape-4;
                        a sh:NodeShape;
                        sh:not [
                            sh:path (ex:true);
                            sh:hasValue "true"
                        ];
                        sh:targetNode ex:root.
schema:conditionShape-1 schema:targetAction schema:actionShape-2;
                        a sh:NodeShape;
                        sh:not schema:corsair-hx850Selected;
                        sh:targetNode ex:root.
schema:conditionShape-2 schema:targetAction schema:actionShape-3;
                         a sh:NodeShape;
                         sh:not schema:ryzen-5-3600;
                         sh:targetNode ex:root.
schema:conditionShape-4 schema:targetAction schema:actionShape-5;
                         a sh:NodeShape;
                         sh:not schema:kingston-ddr3Selected;
                         sh:targetNode ex:root.
```

Action Shapes

```
schema:actionShape-1 a sh:NodeShape;
  sh:targetNode ex:root.
  sh:message "Must select a minimum of 1 variants
                of variation point: CPU";
  sh:or (
    [ sh:path (vp:cpu ex:Variant*) ;
      sh:minCount 1; ]);
```

```
schema:actionShape-2 ex:actionType "RequireAction";
  a sh:NodeShape;
  sh:message "The sum of the all selected
                attributes: POWER_CNSMPT has to be < 850.";
  sh:or ([sh:sparql [sh:select """
    SELECT $this WHERE {
        $this a ex:configuration.
        $this ((ex:VP/ex:Variant/<http://attribute.org/POWER_CNSMPT>/ex:Value)
                   | ex:zero ) ?attr.
    }
    group by $this
    having(SUM(?attr) >= 850)
   """ ]]);

schema:actionShape-3 sh:targetNode ex:root.
  ex:targetIdentifier "Motherboard::CPU_SOCKET";
  a sh:NodeShape;
  sh:message "A selected variant of variation point: Motherboard is required
               to have the attribute CPU_SOCKET::AM4.";
  sh:or ([sh:path (vp:motherboard ex:Variant* attr:CPU_SOCKET ex:Value) ;
           sh:hasValue "AM4"];
           sh:minCount 1
           ).

schema:actionShape-4 sh:targetNode ex:root.
  a sh:NodeShape;
  sh:message> "A selected variant of variation point: CPU is required
                  to have the attribute CLOCK_SPEED > $clock_speed.";
  sh:or ([sh:path (fm:cpu ex:Variant* attr:CLOCK_SPEED ex:Value) ;
   sh:sparql [sh:select """
    SELECT $this ?value WHERE {
        $this a ex:configuration.
        fm:clock_speed ex:Value ?paramvalue.
          $this (ex:VP/ex:Variant/attr:CLOCK_SPEED/ex:Value)
              ?value.
          FILTER(?paramvalue > ?value)
    }
  """ ] ]);
      ex:targetIdentifier "CPU::CLOCK_SPEED".

schema:actionShape-5 a sh:NodeShape;
     sh:message "All selected variants of variation point: Memory are
                    required to have the attribute CLOCK_RATE::1600.";
     sh:or ([sh:path (vp:memory ex:Variant* attr:CLOCK_RATE ex:Value);
             sh:hasValue 2400 ]);
     sh:targetNode ex:root.
```