FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik: Games Engineering

A Description Language for Structural Smells

Carolin E. Brandt

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik: Games Engineering

Eine Beschreibungssprache für strukturelle Smells

A Description Language for Structural Smells

Bearbeiter:	Carolin E. Brandt
Themensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Henning Femmer, Dr. Maximilian Junker
Abgabedatum:	15. Juli 2017

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Juli 2017 Carolin E. Brandt

Acknowledgements

I would like to sincerely thank my supervisor Professor Dr. Dr. h.c. Manfred Broy for giving me the opportunity to write this thesis at his chair.
I would also like to thank my advisor Henning Femmer for his great support on the design of RSL and the thorough reviews of my thesis, as well as my advisor Maximilian Junker for the helpful support in technical aspects and thesis structure.
Furthermore I would like to thank Sebastian for the feedback on my language design and Martin for his rigorous review.

Abstract

High quality requirements artifacts are an important factor in the engineering process of any product, since quality defects within these artifacts are costly to resolve. Therefore many companies try to standardize their requirements artifacts by providing templates and guidelines on how requirements should be documented. These aim at standardizing the structure of requirements and making them easier to understand for all readers, so they can find the information they are looking for faster.

Conformance to these guidelines is still checked with manual reviews. These reviews are however very costly in terms of time the stakeholders and reviewers have to invest. The reviews' quality also greatly depends on how skilled the respective stakeholder is to assess requirements quality. Combining manual reviews with automatic ones can speed up this process. At the moment, there is no structured language to define structural guidelines for requirements artifacts so that they can be automatically quality assured.

To address this shortcoming, we collect different kinds of structural rules that are defined in guidelines for writing requirements used in industrial practice. From the collected rules we infer structural requirements smells. These can be used to detect structural defects within requirements artifacts. We develop a domain specific language to configure an automated analysis of requirements artifacts that detects structural defects. With this language, the user can define the structure, to which an artifact has to conform to. In addition, the user can specify how the author has to be warned about potential quality defects in his requirements documentation. To evaluate our domain specific language, we take rules from existing requirements authoring guides and assess to which extent they can be described by our language. Our analysis found that we can already describe 69% of the structural rules describe in selected guidelines used in industrial practice. Further possible extensions to our language are described at the end of the thesis.

Contents

1. Introduction **3**

2. Background **5**
2.1. Definition of Structural Requirements Smells 5
2.2. Existing Environment 5
2.2.1. Qualicen Scout 5
2.2.2. Structural Analysis Definition – Current Approach 6
2.3. Domain Specific Languages 7
2.3.1. Abstract and Concrete Syntax 7
2.3.2. Motivation for Using a Domain Specific Language 7

3. Development of RSL **9**
3.1. Language Goal 9
3.2. Pre-Study 9
3.2.1. Types of Structural Smells 9
3.2.2. Accessible Information 11
3.2.3. Result 11
3.3. Abstract Syntax 11
3.4. Concrete Syntax 14
3.5. Development Environment 16
3.6. Example Smell Description 17

4. Case Study/Evaluation **19**
4.1. Study Goal 19
4.2. Study Design 19
4.3. Study Objects 19
4.4. Study Execution 19
4.5. Study Results 26
4.6. Interpretation 26
4.7. Threats to Validity 26

5. Related Work **27**
5.1. Automatic Detection of Quality Defects 27
5.2. Using Requirements Guidelines For Quality Assurance 27
5.3. Structural Defects in Requirements Artifacts 28
5.4. Domain Specific Languages for Smells 29

6. Conclusion **31**
6.1. Summary 31
6.2. Future Work 31
6.2.1. Technical Work on RSL 31
6.2.2. Studies 32

A. Complete Smell Description **35**

1. Introduction

At the start of a development process, customers and developers together have to define, which functionalities a new product should have. This is performed in the so called requirements engineering phase, where the requirements are elicited and documented by requirements engineers. Based on these requirements artifacts, the developers realize the product and testers design test cases for the product. Therefore the requirements artifacts are the basis for the further development activities.

As "errors become exponentially more costly with each phase in which they are unresolved" [Wes02], it is important to eradicate possible quality defects in the requirements artifacts as early as possible. As an additional difficulty, requirements are sometimes formulated in natural language. Natural language is used because very diverse stakeholders are involved in the creation and usage of requirements artifacts and they all can understand natural language without additional training. However, natural language is prone to misunderstandings and inconsistency, two critical quality defects in requirements artifacts [Ott12], because natural language lacks precise syntactic and semantic rules [PSS14]. In summary, natural language requirements artifact are liable to critical quality defects.

It was shown, that assuring the quality of natural language artifacts by manual reviews is effective in terms of detecting defects more cheaply than through testing the developed product [ABL89]. A manual review process involves a serious time commitment from several people and is therefore expensive. For a manual review, several stakeholders have to thoroughly read the artifact and understand it. They have to search for signs of quality defects, as well as document their results and have to communicate them back to the requirements author. Also, the quality of manual reviews greatly depends on how skilled the reviewer is to asses the quality of requirements artifacts. Therefore, assuring the quality of natural language requirements artifacts through manual reviews is cheaper than through testing, but still costly in terms of time the stakeholders have to invest.

The manual quality assurance process can can be supplemented with automatic analyses. An automatic detection of defect signs can contribute to the process in several ways: First, if an automated analysis is performed directly while a requirement is documented or shortly after the documentation is finished, the requirements author has a chance to cull some defects and inconsistencies immediately, making the artifact more readable for the stakeholders reviewing it afterwards. Second, a reviewer can use the results from an automated analysis to identify areas with probable quality defects. This simplifies the review for him, as he knows where to focus his effort towards improving the quality of the requirements artifact. Third, the review of some aspects can be automated completely, so there is no need for the stakeholder to review these aspects. Hence, a review process can benefit from a combination of manual reviews and automatic analyses.

To simplify the review process and the usage of requirements artifacts in general, companies standardize their requirements by defining templates and guidelines on how requirements have to be documented. For example, some guidelines request a section that describes the standard flow of actions in a use case. In addition, the guidelines could define the title for the section describing this flow of actions. The aim of such guidelines is to make the artifacts easier to understand for all readers, so they can find the information they are looking for faster [PSS14]. In turn, a requirements artifact not conforming to those templates and guidelines could take more time for a reader to understand and use. This could have a negative impact on the time and cost budget of the software project. In summary, documentation guidelines are used to speed up the review process.

Authoring guidelines can consist of several different types of rules: lexical, grammatical, structural and semantic rules [FUG17]. As lexical, grammatical and semantic rules were already covered by Simon Lang in his bachelor's thesis [Lan15], this thesis focuses only on structural rules. The structural rules defined in such templates and guidelines could be checked by an automated analysis tool, simplifying the feedback to the author on whether his artifact conforms to these rules or not.

Problem Statement Currently there is no structured language to define structural guidelines for requirements artifacts so that the artifacts can be automatically quality assured.

Thesis Goal In this thesis we design a domain specific language to configure an automated analysis of requirements artifacts that detects structural defects.

Thesis Structure At first, we describe how a structural analysis is currently defined using code. We motivate why we develop a domain specific language for such definitions. As a base for our research, we collect different kinds of structural rules that are defined in guidelines for writing requirements and infer structural requirements smells from them. We use them to create the domain specific language RSL (Requirements Structure Language). With this language, the user can define the structure, to which an artifact has to conform to. In addition, the user can specify how the author has to be warned about potential quality defects in his requirements documentation. To validate our language, we take rules from existing requirements authoring guidelines and evaluate to which extent they can be described by RSL. At the end of the thesis, we give an overview over the state of the art on automatic analysis of requirements artifacts and how this thesis differs from existing ones. As we focused on developing an extendable core with our language, we propose extensions to RSL and present aspects for further research.

2. Background

A very diverse group of stakeholders reads requirements documents. They consist of technical experts as well as management and include customers from different domains, such as the automotive or the insurance domain. Natural language is used to document requirements in the form of use cases, because it can be understood by all stakeholders involved and gives a lot of elaboration capabilities to the author. However, natural language descriptions are prone to ambiguities and inconsistencies as they do not have precise syntactic and semantic rules [PSS14].

To minimize these common defects, companies often introduce authoring guidelines, consisting of various rules about the desired content, wording and structure of requirements artifacts. These guidelines are inferred from known types of quality defects and try to help the writer to avoid these by prohibiting various possible signs of quality defects that might be showing in the texts.

These signs can be seen as requirements smells, a concept introduced by Femmer et al. [Fem+17], that is similar to code smells, which were first mentioned by Fowler and Beck [FB99]. In the following, we present the definition of requirements smells and extend it to distinct structural smells. Afterwards we present a list of structural smells, explaining which quality defects they indicate and which impact these might have on the software development process.

2.1. Definition of Structural Requirements Smells

Femmer et al. give these central characteristics for a Requirements Smell [Fem+17]:

- A Smell *indicates a quality violation* in a requirements document. Bad quality is in this case defined by its negative impact on the software engineering process based on the requirements document.
- The existence of a Requirements Smell *does not automatically imply the existence of a defect* in the requirements document and has to be judged by the context.
- Every Smell has a *concrete location in the requirements document*, which differs from general or abstract quality criteria like completeness or consistency in requirements.
- Each Smell can be detected by a *concrete detection mechanism*, which can be more or less accurate.

In the following we understand a structural Smell as:

- A structural Smell has a *detection mechanism* that looks at how the text is *structured, formatted* or looks at the *plain, unprocessed text* itself.

2.2. Existing Environment

2.2.1. Qualicen Scout

This thesis builds on the software tool Qualicen Scout, which is a quality analysis tool for natural language requirements and tests. It builds upon ConQAT[1], a software quality analysis framework (developed by CQSE GmbH and the Competence Center Software Management at Technische Universität München) and upon Teamscale, a code quality tool (developed by CQSE GmbH[2]). Qualicen GmbH equipped Teamscale with natural language processing techniques and defined smell detectors for a variety of requirements and tests smells. This thesis aims at improving the configuration of the structural analysis performed by Qualicen Scout.

1 www.conqat.org
2 https://www.cqse.eu/en/products/teamscale/overview/

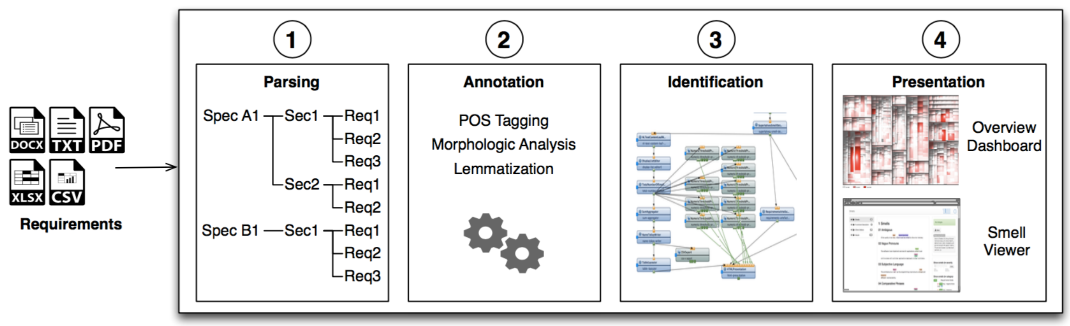

Figure 2.1.: The smell detection process in Qualicen Scout, Image taken from [Fem+17]

Detection Process in Qualicen Scout The smell detection process of Qualicen Scout consists of four phases [Fem+17]:

- **Parsing:** Scout parses requirements from various formats (mainly MSWord and plain text) into a tree of single items. It represents the structure of the document and contains plain text as well as formatting information for each item.
- **Annotation:** Afterwards, Scout applies various natural language processing techniques (e.g., Part-of-Speech Tagging, Morphological Analysis, Lemmatization) to add meta information about the language and words used.
- **Identification:** Based on the plain text, the artifacts structure parsed in the first step and the annotated meta information from the second step, occurrences of requirements smells are detected. This is done by *smell detectors*, who tag certain words or phrases as findings for a particular smell. This process is further refined by *findings filters*, who filter out false positives.
- **Presentation:** The remaining findings are then shown to the user. To accomplish this, the tool displays the whole requirements artifact and highlights the concrete locations of the findings. If the user hovers over the highlighted text, she gets detailed information about each finding. The presentation of the findings directly within the document is important, as a smell *does not necessarily imply the existence of a defect* [Fem+17]. Therefore each smell instance has to be assessed in its context, in order to identify actual quality defects.
  Another important feature are dashboards, where the individual findings are combined into global metrics. These indicate hotspots: documents or sections that have a high number of critical findings relative to other parts of the requirements specification. The user can also blacklist false positives and disable the detection of smells that are irrelevant to him.

The structural analysis that should be defined by the language which we develop in this thesis, builds upon the parsed tree structure with plain text and formatting information. The analysis itself is part of the third step **Identification**. As we look only at structural smells, which in our definition do not rely on meta data obtained by natural language processing techniques, we do not access any of the data annotated in the second step **Annotation**.

2.2.2. Structural Analysis Definition – Current Approach

At the moment, the structural analysis of requirements in Scout is defined through Java code, which is specifically written for one set of authoring guidelines. However, guidelines differ throughout companies and with the current approach the whole structural analysis code has to be rewritten and Scout has to be rebuilt for every new guideline. In our opinion, adapting the structural analysis to new guidelines should be a configuration task. Instead, at the moment, this is a development task, which involves the developers of a tool. In addition, rewriting the code every time is not efficient, as many checks are similar for several writing guidelines. One example for that could be the correct number of steps in a flow description of a use case. Writing guidelines would give an upper or lower boundary, although the exact values are different. Therefore, if Scout had a method to configure its structural analysis outside of the code, it would reduce development efforts and improve the Qualicen Scout.

2.3. Domain Specific Languages

In his book *Domain Specific Languages* [Fow10] Martin Fowler defines a domain specific language as following:

> Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain.

This definition can be broken up into several parts:

- **"a computer programming language":** A domain specific language has to be executable by a computer, so it can be used by a human to direct the actions of a computer.
- **"of limited expressiveness":** Other than a general-purpose programming language, a domain specific language only supports the smallest possible set of features that are needed to support its domain.
- **"focused on a particular domain.":** A language with a limited expressiveness is only useful if it focuses on a small domain.

In this thesis, we develop an *external* domain specific language [Fow10]. This means that it is separate from the application code and usually has a different syntax, which is parsed by the main application's code. In contrast to that, an *internal* domain specific is a particular way of using a general purpose language.

2.3.1. Abstract and Concrete Syntax

Each domain specific language has two types of syntax. First, the *abstract syntax* describes the logical data structure of the information that is retrieved from the language [Fow10]. The *abstract syntax* is a semantic meta-model that is independent of any domain specific language. Second, the *concrete syntax* is a grammar, which defines the legal syntax in a language. It is used to create a parser that turns the input text into a parse tree, that is similar to the structure of the grammar rules. With this parse tree, the semantic model is populated, so it represents the information that was encoded in the input text.

2.3.2. Motivation for Using a Domain Specific Language

We decided to define the Scout's structural analysis outside of its Java code. There are two possible methods to do that: Let the user define the structural analysis in separate Java classes, which are dynamically loaded at runtime. Or define the analysis in an external domain specific language. We chose develop a separate domain specific language for such definitions for the following reasons:

Target Group: Non-IT background One main reason for configuring a structural analysis outside of the code of a tool is that no developers have to be involved. Instead the automated structural analysis should be defined by the same persons who also wrote the artifact authoring guidelines. One the one hand, they are already familiar with the rules the authors have to adhere to, so they know, which aspects have to be checked by an automated structural analysis. On the other hand, they can give more detailed explanations to the author, why there is a structural error at this location in the document, as they are more familiar with the rationale for a certain rule in their guideline, as an external developer. Since the guideline authors might not be programmers, they do not necessarily know one of the popular general purpose programming languages.

Abstraction from the analysis Defining requirements smells in a separate artifact introduces an additional layer of abstraction. This layer separates the analysis definition from the analysis implementation, which has several benefits:

- Reusing the smell definition in another tool is simple, because it does not rely on any implementation details.
- Modular, reusable code can be written for the analysis. This means, that very general checks are implemented once and can be executed several times with different parameters for different smells.
- Ensuring backwards compatibility is easier for the developers, as there is only a limited, well defined set of items to be used in the smell definition. If the analysis definition would be written directly against the analysis API, any changes in there could potentially break the analysis, resulting in higher maintenance effort for the customer.

Focus on the important concepts We also chose to define a new language for structural smells, so it consists only of features that are sensible in this context. Defining an analysis in a general purpose language would force us to handle all already existing constructs of the language. In our case, describing smells is a rather descriptive task, that should be done by a declarative programming language. Using an popular iterative programming language would require us to give a semantic meaning to for example loops, which are not present in a declarative language.

Safety Naively executing external code in your system can lead to problems during runtime. The external code could break the system, if the code is not tested or checked for correctness beforehand. Also you would have to verify, that the external code is not malicious and does not influence the system outside the structural analysis. If the external configuration code is written and deployed by the customer, the system would need failure tracking and recovery, in case of faults in the external code. To enable the customer's developers to trace the origin of such failures, the system would need to provide detailed feedback on occurring errors and the surrounding state of the system. This would reveal otherwise invisible implementation details to the customer.

3. Development of RSL

In this section, we present the language we develop in this thesis, Requirements Structure Language (RSL). We state our design goal and justify the decisions on what our language should be able to represent. Afterwards we introduce the abstract syntax of the language and demonstrate how the needed information is encoded in the concrete syntax.

3.1. Language Goal

For this thesis, we design a language that empowers the user to define an automated analysis for structural issues within a requirements artifact.

3.2. Pre-Study

When deciding which smells the user should be able to describe with RSL, there are two possible approaches: The first approach is, to consider which smells the user wants to detect. The second one is, to consider which smells can be detected in the existing environment.

3.2.1. Types of Structural Smells

To identify which smells the users of RSL might want to detect, we examine a real-world requirements writing guide, the *Use Case Authoring & Review Guide* employed by an insurance company, and select all rules, which relate to the structure of the desired artifact. We group similar rules together, considering the location to apply the rule and the detection mechanism. They differ only in the data the detection mechanism has to check against. For each group, we define a more general smell definition. One example for that can be seen in Figure 3.1.

Additionally, we collected a range of possible structural smells from other papers, mainly from Parachuri et al. [PSS14] and Femmer et. al [FUG17].
In this section we present possible structural smells we encountered. We show a summary of them in Table 3.1.

Violation of naming conventions Some companies have conventions on how to name different requirements artifacts or certain sections in these artifacts. These can consist of abbreviations that refer to the project the individual requirement belongs to and are often numbered to make them uniquely identifiable. Violating such naming conventions could be confusing to further readers and lead to inconsistent references to requirements.

Rule 1: *The title of a use case consists of an identifier and a name. <ProductPrefix>-UC<nn> <name>*
Rule 2: *Always start the name of an alternative flow with an identifier. The identifier consists of the acronym "AF" followed by a double-digit number. AF<nn> <name>*
Generalized Smell: *Headline not conforming to naming convention.*

Figure 3.1.: Example for deriving a general smell description

Smell Type	Information needed for detection	Indicated defect	References
Violation of naming conventions	unprocessed text, formatting	inconsistent references	
Leaving out mandatory sections	structure	incompleteness	
Incorrect references	unprocessed text, formatting, glossary	inconsistent references	[FUG17] [Fem+17]
Too few/too many steps	structure	high complexity	[Coc00]
Clones	unprocessed text	greater maintenance effort	[Jue+10]

Table 3.1.: Types of Structural Smells

Leaving out mandatory sections Most requirement templates have several mandatory sections that have to exist in the final requirement artifact. For example use cases following a certain template might have to contain a section stating the motivation of the user and one describing the flow of events during the interaction with the system. A mandatory section that is not present in a requirement artifact indicates that the requirement is incomplete and misses important information for a stakeholder.

Incorrect references References in requirements are used to avoid clones by reusing texts or whole requirements that are already defined in the requirements artifact or associated artifacts. These references are mostly indicated by a predefined syntax and use a name (e.g. the other document's title [FUG17]) to indicate what is referenced. If a reference is not correct either in syntax or format, it cannot be recognized as such or the reader could have difficulties retracing what is supposed to be referenced [Fem+17].

Too few/too many steps Requirements are often documented as use cases, which consist of a sequence of actions or events to describe what the user does and how the system reacts. Having too little steps in a use case could indicate that its subject matter is too small to justify a separate requirements artifact or that too many interactions between the user and the system were combined into one step [Coc00].

Having too many steps on the other hand can also be a problem. If the interaction between user and system is cut into too many small steps, it becomes more difficult for the reader to pinpoint the important aspects of the sequence of events described. A programmer for example could fail to determine which features exactly are described by a given requirements artifact, if the user interacts with too many parts of the system within one use case.

Clones Clones are a problem in many software engineering domains. They occur if an engineer reuses already existing text by copying and pasting it to another document. This leads to a greater maintenance effort [Jue+10], because changes that have to be made in one clone instance also have to be made in all siblings of this instance. If an engineer misses to adjust one of the siblings, the clones can lead to inconsistency in the requirements.

The general smell definitions which we identified from the authoring guideline overlap with the smell descriptions we collected from the literature, so we combined them into an "ideal set" of smells that RSL should be able to describe. Apart from cloning, all the smells described there can be mapped to at least one rule in the *Use Case Authoring & Review Guide*.

3.2.2. Accessible Information

Part of our goal is, to define an automatic analysis with RSL, which means that the language can only use such rules, that can be tested by a computer. For example, the smell *"Unnatural itemizations"* [Fem+17] might be hard for a computer to decide nowadays, as computers can not yet build a narrative understanding of natural language texts [CW14].

In addition to that, our checks have to rely on structural information that can be parsed from a requirements document. A possible result of such a parsing process of a MSWord file would be a tree like structure that represents the parts of the document and their relation to each other. Each leaf in the tree consists of the plain text from a particular document part, inner nodes cluster their children together into one section, while the headline of this section is saved in the plain text attribute of the parent node. In addition to encoding the structural information within the structure of the tree, you can annotate each node with the style the corresponding document part has (this is referring to MSWord styles, e.g., "`Heading 3`", "`List Paragraph`") and how it is formatted (is it `numbered` or `itemized`).

The tree structure created in the **Parsing** step of the smell detection process in the Qualicen Scout we presented in Section 2.2 is similar to the one we just proposed.

3.2.3. Result

Taking our earlier described "ideal set" of smells that we should be able to describe with our language, we sorted out those smells, which relied on a semantic or narrative understanding of the text for detection. Afterwards we selected a set of rules to make available in the language. These rules are simple checks, that a computer can execute at a certain location in the parsed document tree. One example would be, if the text in the certain document part matches a given regular expression. The selected rules should enable the computer to detect the remaining structural smells. The complete set of rules we selected is described in Section 3.3.

3.3. Abstract Syntax

As described in Section 2.3.1, the *abstract syntax* describes the information that the user can encode in a domain specific language. Before explaining the abstract syntax of RSL in detail, we want to introduce the main concept with the help of an example:

With RSL, the user should be able to encode an automated analysis for structural defects within a requirements document. As an example, we take a guideline from the *Use Case Authoring & Review Guide*:

"The Basic Flow should consist of 3 to 9 steps."

For an analysis, this rule can be broken down into the following aspects:

Location: The first aspect is the location, which a guideline refers to. In this case, the guideline's location is described by "The Basic Flow". When looking at the the template, which corresponds to the *Use Case Authoring & Review Guide*, the document part called *Basic Flow* is the fourth section in the document.

Therefore, when defining an automated analysis for the selected guideline, the user has to first define, where the rule should be applied. In our syntax, this is symbolized by `Sections`. One `Section` corresponds to one document part. Several `Sections` following each other are combined into a `SectionBlock`. Each part in the document can also have subparts (e.g., part 2 has the subparts 2.1 and 2.2). To represent this in our abstract syntax, each `Section` can have `subsections`. As these are again multiple `Sections`, which should be next to each other in the document, they are combined into one `SectionBlock`. The location of a `Section` in the analysis is determined by its location in the definition of the analysis using RSL. For each `Section` the user can define checks for the structural smells that can occur in this section.

Rule: The second aspect of each guideline taken from the authoring guideline is the actual rule that has to be checked. In this case, the constraint given by the guideline is: there should be "3 to 9 steps" at the beforehand defined location.

Consequently, when defining an automated analysis for the selected guideline, the user has to define checks, that are performed in the analysis, to determine whether the requirements artifact conforms to a given guideline or not. In RSL, this is done by `Conditions`. Think of `Conditions` as a sequence of simple checks that determine whether a smell is present at a certain location or not. Each `Condition` consists of a simple `Rule`, that is checked. If a `Rule` is met, there are two possible ways the analysis can continue: First, if meeting this `Rule` implies the presence of a smell at this location, the user can specify that a finding with a certain `name` and `message` should be added. Second, if there are still checks necessary to determine, whether there is a smell or not, the user can define another `Condition` that should be evaluated next.

In summary, the location, where a guideline has to be applied, is described by the use of `Sections` in RSL. The actual checks, that have to be done in order to determine, whether a smell is present at that location or not, are described by `Conditions`. These are the basic aspects of our abstract syntax, which is however more detailed in order to more precisely define the analysis. This includes more narrow characterizations of the location where a `Condition` should be applied (with `selectionConditions` for `Sections`, as well as `header`, `text` and `sectionConditions`). It also includes several additional `Rule`, which can be used in `Conditions`.

In the following, we describe the abstract syntax of RSL in detail. For an overview, you can see the whole abstract syntax diagram in Figure 3.5. To simplify explanation, we break up the diagram into several parts:

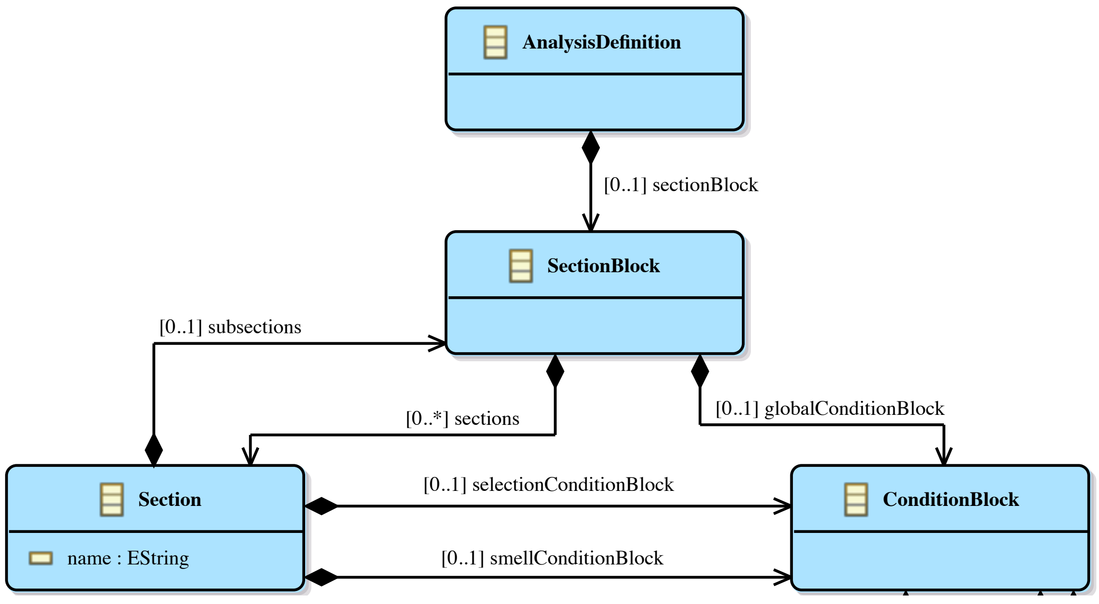

Figure 3.2.: Abstract Syntax of RSL, Sections

Sections

As you can see in Figure 3.2, the whole smell description for one document is encapsulated in the `AnalysisDefinition`. It consists of one `SectionBlock`, which as convention, contains one section for the whole document. Each `SectionBlock` is composed of several `Sections`, which are next to each other in a document. This list of Sections is boxed in the type SectionBlock, because each SectionBlock also has a block of global Conditions, which are checked in each Section of this SectionBlock, as well as recursively in every subsection of those.

A Section has two ConditionBlocks:

- The selectionConditionBlock describes, which characteristics a part in the document has to have, so that it corresponds to the defined Section.
- The smellConditionBlock contains all the smell descriptions (in form of Conditions) that should be checked in a document part, which corresponds to the selectionCondition-Block.

If the selectionConditionBlock is left out, the location of the part corresponding to the Section is inferred from the location of the Section inside the AnalysisDefinition. If the selectionCon-ditionBlock is specified, the selection characteristics are checked for every part relating to the SectionBlock, as well as their subparts.

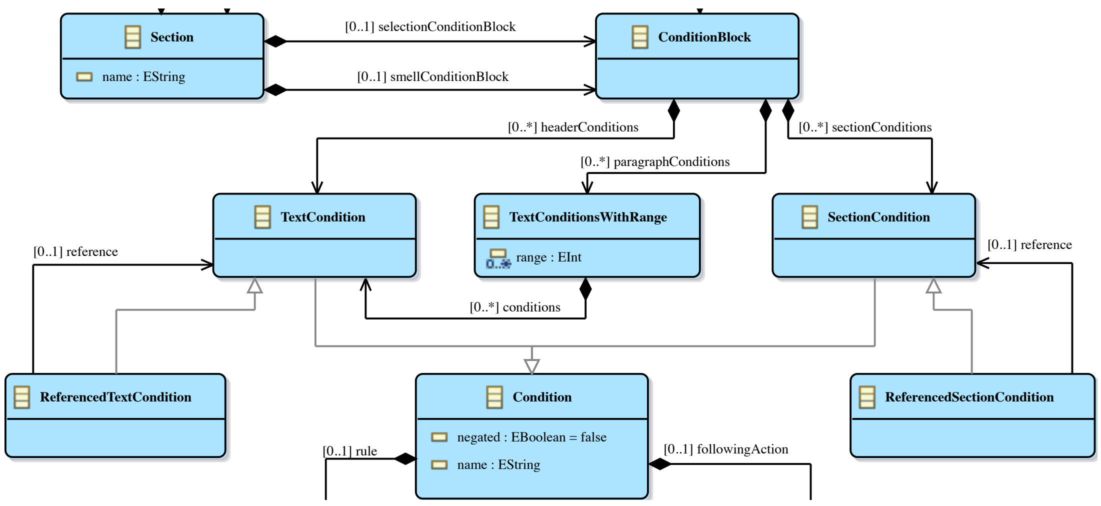

Figure 3.3.: Abstract Syntax of RSL, Conditions

Conditions and Rules

A ConditionBlock can always be applied to a node from the documents tree structure, representing one part of the document. As you can see in Figure 3.3, a ConditionBlock has three categories of Conditions in which the user can each define arbitrarily many Conditions:

- headerConditions, which are of the type TextCondition and will be checked by looking at the headline of the document part.
- paragraphConditions, which are also TextConditions, but are annotated with the range of text objects they should be checked for. For example, a document part consist of three paragraphs, but only the second and the third should be checked for a certain smell. The Condition encoding this smell would then be annotated with the range 2,3.
- sectionConditions, which are of the type SectionCondition. These describe smells which relate to the document structure.

As the user might not want to redefine similar Conditions over and over again, we defined ReferencedConditions: Any Condition can be annotated with a name and that name can later be reused instead of redefining the identical Condition again.

Every smell or selection characteristic is defined through a chain of Conditions. You can see the class structure corresponding to Conditions in Figure 3.4.

Each Condition has one Rule that will be applied to the document part the Condition is checked for. You can think of a Rule as a check that returns a boolean value. There are two types of Rules:

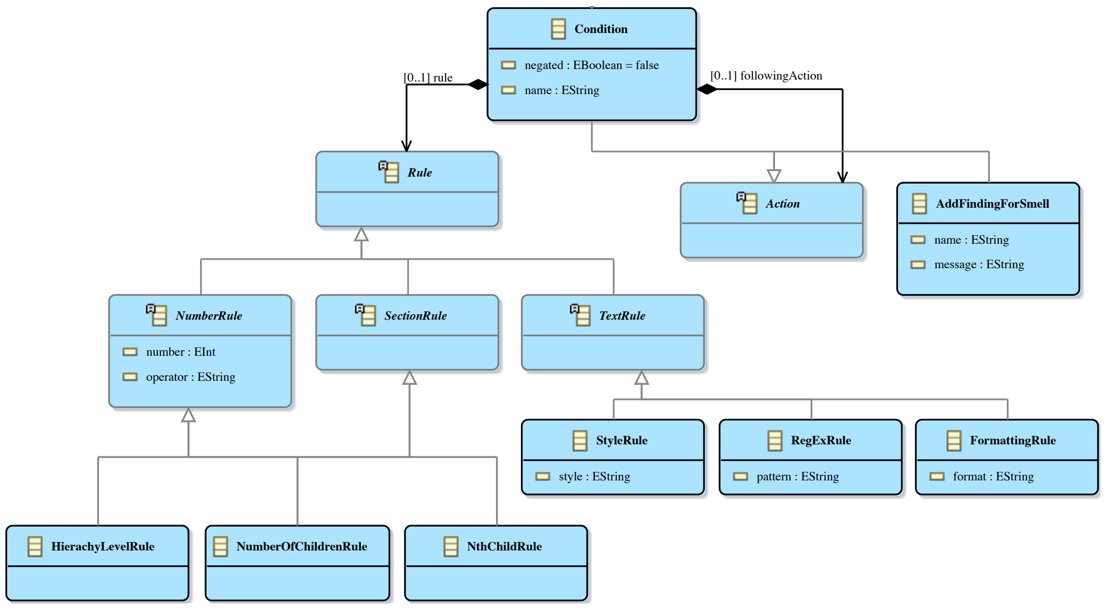

Figure 3.4.: Abstract Syntax of RSL, Rules

- `TextRules` can be applied in `TextConditions`:
  - The `StyleRule` checks, whether the MSWord Style of a document part is the same as a given `String`.
  - The `RegExRule` checks, whether the plain text of a document part matches a given (Java) regular expression.
  - The `FormattingRule` checks, whether the document part is either "`Itemized`" or "`Numbered`".
- `SectionRules` can be applied in `SectionConditions`:
  - The `HierachyLevelRule` checks, whether the document part is on a specified hierarchy level.
  - The `NumberOfChildrenRule` checks, whether the document part has a given number of suparts.
  - The `NthChildRule` checks, whether the document part has a specified position in the subparts of his superpart.

  We were able to generalize all the `SectionRules` into `NumberRules`, which consist of a number and a corresponding operator, such as "=", "<" or ">=". With the operator the user can also specify a range of values for the checks done by `NumberRules`.

Each `Condition` can additionally be `negated`, to invert the result of a `Rule`. This should enable the user to define smells using positive and negative examples (e.g., the text should match "x", but should not match "y").

If a `Condition` is met by the document part it is applied to, the analysis should continue with the `followingAction` for that `Condition`. This can be another `Condition` of the same type as before (as `TextConditions` and `SectionConditions` are applied to different document parts they can not be mixed). When using `Conditions` to define smells, the `followingAction` can also be of the type `AddFindingForSmell`. This implies, that if the chain of `Conditions` matched to a certain document part, a smell finding with a given `name` and `message` should be created.

3.4. Concrete Syntax

The concrete syntax of a domain specific language defines the keywords and characters used to describe the information from the abstract syntax. It is used to define the parser, which extracts the encoded information.

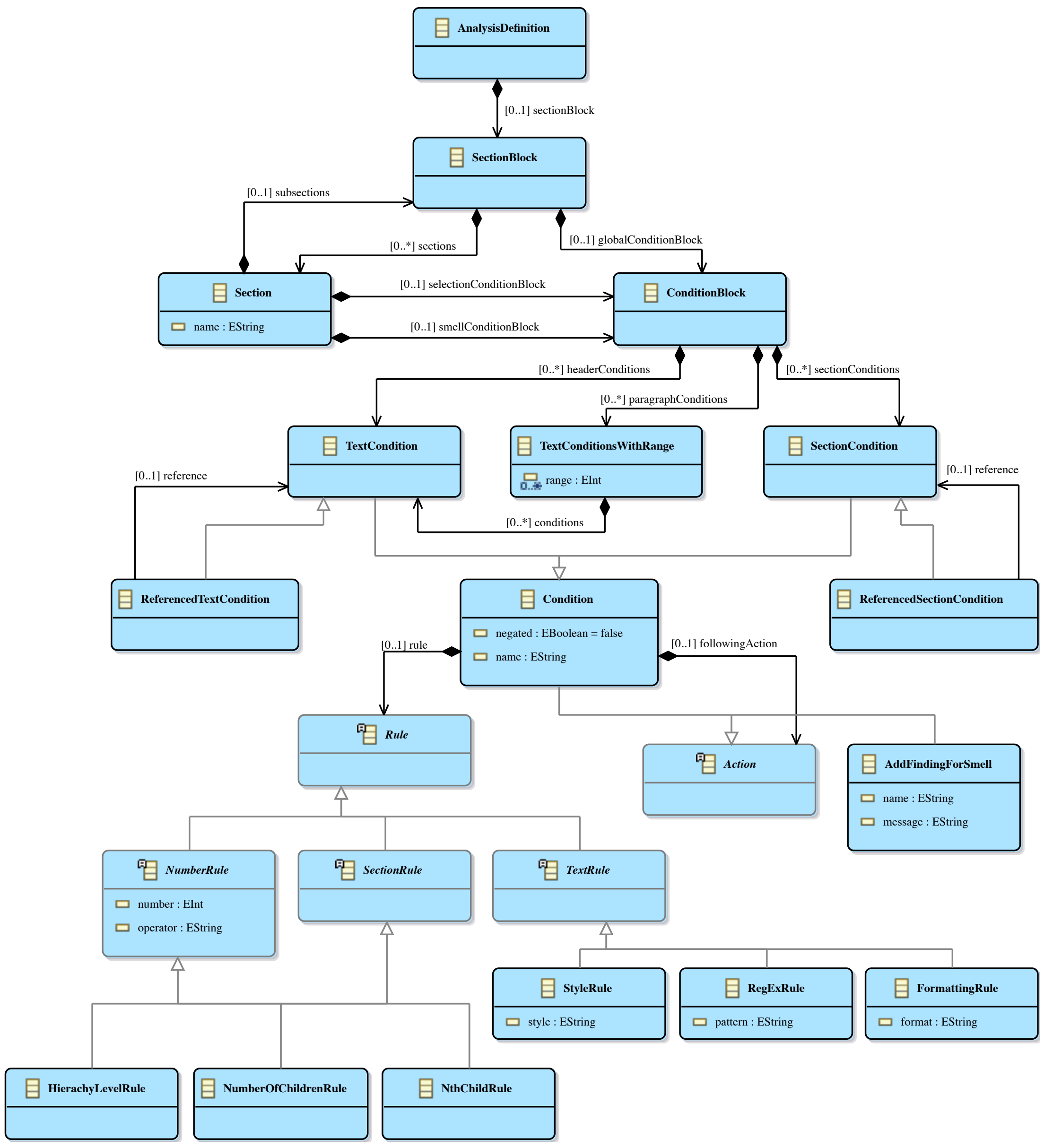

Figure 3.5.: Abstract Syntax of RSL

Our goal for the concrete syntax was to keep it short and precise, to not force the user to type unnecessary keywords over and over again. Another goal was, that an uneducated user should roughly understand, what is going on in the analysis definition on the first read, to simplify introducing new users to our language. Therefore we chose to define hierarchical information with brackets and indention, while formulating `Conditions` in a slimmed sentence style.

We will now present the concrete syntax for the entities of our abstract syntax. For further explanation on the semantics of the used types and variables, please refer to Section 3.3. All elements framed by "?" are optional for the user.

SectionBlock

```
?all sections: <globalConditionBlock>?
<sections>
```

ConditionBlock

```
?headline: <headerConditions>?
?text ?<range>? : <paragraphConditions>?
?section: <sectionConditions>?
```

If no range is defined for the `textConditions`, they are applied to every text of a corresponding document part.

Section

```
section ?<name>? ?found by <selectionConditionBlock>?
{
  <smellConditionBlock>
  ?<SectionBlock with subsections>?
}
```

Condition

```
?[<name>]? if ?not? <rule> and <following condition>
?[<name>]? if ?not? <rule> <smell definition>
```

Referencing a Condition

```
-> [<name of referenced condition>]
```

Smell Definition

```
add finding <name> with message <message>
```

Rule

```
TextMatches("<Java regular expression>")
Format("<'Itemzed' or 'Numbered'>")
Style("<MSWord style id>")
HierachyLevel(?<operator>?<number>)
NumberOfSubsections(?<operator>?<number>)
HasChildPosition(?<operator>?<number>)
```

"=" is the default value for the operator of `NumberRules` and can also be left out.

3.5. Development Environment

We build our abstract syntax model with the *Eclipse Modeling Framework*[1], which also generated the corresponding code for each class in the model. To define the concrete syntax we used *Xtext*[2]. We used the parser provided by Xtext and implemented the inferred structural analysis in Java, building on the exisiting ConQAT analysis in Qualicen Scout.

1 http://www.eclipse.org/modeling/emf/
2 https://eclipse.org/Xtext/

3.6. Example Smell Description

To conclude this section we give an example smell description in RSL and explain the abstract and concrete Syntax for it.

As an example we define a check for the following guideline in RSL:

> *"The headline of the second section of the requirements document should be 'Background' or have the MSWord style 'heading1'."*

This guideline can be separated into a location and a rule.

Location The location in this guideline is specified by: *"The headline of the second section of the requirements document"*. Therefore the smell description in RSL has to identify the second section of the whole document. This can for example be done by using the `selectionConditionBlock` of a `Section`:

```
section background found by
section:
  if HierachyLevel(1)
  and if HasChildPosition(2);
{
  headline:
    ...
}
```

In this case, we use a `SectionCondition` with two steps to select the correct document part. The first step of the `SectionCondition` (`if HierachyLevel(1)`) checks that the document part is on the first hierarchy level. The guideline does not apply to any deeper nested document parts and these are excluded by the rule check in this first step. The `followingAction` of the first step is the second step: `and if HasChildPosition(2);` checks that the document part is the second part on its hierachy level, as the guideline refers to the second document section. The semicolon indicates that the `Condition` is finished after the second step, and the correct part of the document was identified. The guideline defines rules for the headline of the identified document section. Therefore the `Condition` describing the smell has to be applied to the headline. This is specified by placing it inside the `headerConditions` of the section with `headline: ...`.

Rule The second aspect is the actual rule, which has to be checked. In the example guideline it is defined by: *"should be 'Background' or have the MSWord style 'heading1'."*. This is defined again by a `Condition`:

```
if not Style("heading1")
  and if not TextMatches("Background")
  add finding "Wrong Title"
  with message "Please name the second section 'Background'.";
```

This `Condition` first checks a `StyleRule`: Is the MSWord style of the text "heading1"? The result is negated using the keyword `not`, as the automatic analysis should identify headlines that do not have the specified style. The `followingAction` is also a `Condition` that checks a `RegExRule`: Does the text match the regular expression "Background"? This `Condition` is also negated. If this `Condition` is also met, the example guideline is violated. Therefore the `followingAction` for this `Condition` adds a finding titled "Wrong Title" with the description "Please name the second section 'Background'."

This is the complete smell description for a structural analysis only checking this guideline:

```
section document
{
  section background found by
  section:
    if HierachyLevel(1)
    and if HasChildPosition(2);
  {
    headline:
      if not Style("heading1")
        and if not TextMatches("Background")
        add finding "Wrong Title"
        with message "Please name the second section 'Background'.";
  }
}
```

You can see further examples of smell descriptions in RSL in our study in Section 4.4 and in Appendix A.

4. Case Study/Evaluation

In this chapter we will present the case study we conducted in order to validate the functionality of RSL, the language we developed in this thesis.

4.1. Study Goal

The goal of this case study is to validate the functionality of RSL. We refine this task into two research questions:

RQ1: To which extent can RSL describe checks for conformance to structural authoring guidelines?

RQ2: Which rules can not be described in RSL and why?

4.2. Study Design

For the study we take a set of structural rules from an authoring guideline and write a description in RSL for each rule. Then, we evaluate how well the rule could be described by assigning one of the following categories to it:

Category 1: Completely describable with RSL
Category 2: Partly describable with RSL
Category 3: Not describable with RSL, but suited for automatic detection
Category 4: Not suited for automatic detection

In the end, we calculate the percentage of how many rules fell into each category.

4.3. Study Objects

It is important to have a concrete set of rules, preferably with a corresponding template, as vague guidelines can not be quality assured automatically, without interpreting them in a way, that an algorithm can understand. Our study objects are rules from the *Use Case Authoring & Review Guide*. We inspected all 53 rules from the authoring guide and selected those that can be expressed with a structural smell, following our definition from Section 2.1. This means, that whether the document conforms to the selected rule can be determined by either checking how the document is structured, how certain sections are formatted, analyzing the unprocessed text of certain sections or a combination of the above. Our inspection yielded 16 rules that conform to our definition of structural smells.

4.4. Study Execution

In the following we will present the selected 16 rules by showing an excerpt of them from the authoring guide and the corresponding smell descriptions in RSL.

Rule 1 - Use Case Title

Excerpt from the *Use Case Authoring & Review Guide*:
Use Cases: <ProductPrefix>-UC<nn> <name>
The title of a use case consists of an identifier and a name. [...]
The identifier consists of a product prefix, the acronym "- UC" and a double-digit number .
Example: DIY-UC01 Browse and Shop

Smell description in RSL:

```
section document
{
  text 1:
    // rule 1
    if not TextMatches("[A-Z]+-UC\\d\\d .*")
      add finding "Incorrect Use Case Title"
      with message "The title of a use case consists of an identifier and a name. The
     name always starts with an active verb, in most cases followed by an object. The
     name describes the goal of the primary actor. The identifier consists of a
    product prefix, the acronym '- UC' and a double-digit number. Examples: DIY-UC01
    Browse and Shop, DIY-UC02 Checkout, DIY-UC03 Manage Account";
  ...
```

When looking at the template provided with the *Use Case Authoring & Review Guide*, the use case title is the first text object in the whole document. Therefore we check if text item number 1 in the first Section of our analysis definition (which as a convention represents the whole document) conforms to the specified name template.

This rule can be described in RSL, therefore we assign it to *Category 1*.

Rule 2 - Title of Basic Flow

Basic Flow: Basic Flow
Always label it "Basic Flow"

```
section document
{
  ...
  section intro { }

  section brief_descripition { }

  section preconditions { }

  section basic_flow
  {
    headline:
      // rule 2
      if not TextMatches("Basic Flow")
        add finding "Basic Flow Title"
        with message "This section should contain the basic flow of events and should
     always be titled 'Basic Flow'.";
...
```

Identifying the document part containing the basic flow of events in the use case is rather tricky. For other purposes it could be identified by

```
section basic_flow found by
headline:
  if TextMatches("Basic Flow");
{ ... }
```

but this would defeat the whole purpose of checking the rule, as the document part corresponding to the Section basic_flow would not be found if the headline is misspelled.

Therefore we chose to identify the Section basic_flow as the fourth part of the document, because it is at this same position in the template provided with the authoring guidelines. Identifying a Section as the fourth Section of a SectionBlock is done by placing three Sections, in this case empty ones, above it.

As this rule can also be completely described with RSL, it falls into *Category 1*.

Rule 3 - Title of Alternative Flow

Alternative Flows: AF<nn> <name>
Always start the name of an alternative flow with an identifier. The identifier consists of the acronym "AF" followed by a double-digit number.
Example:
AF01 Handle the Withdrawal of a Non-Standard Amount

```
...
section alternative_flows found by
headline:
  if TextMatches("Alternative Flows");
section:
  if HierachyLevel(1)
    and HasChildPosition(>5);
{
  ...
  // for fully described use case narrative
  section alternative_flow found by
  headline:
    if TextMatches(".*");
  {
    headline:
      // rule 3
      [alternative_flow_title]
      if not TextMatches("AF\\d\\d .*")
        add finding "Alternative Flow Title"
        with message "Always start the name of an alternative flow with an identifier
    . The identifier consists of the acronym 'AF' followed by a double-digit number:
    Example: AF01 Handle the Withdrawal of a Non-Standard Amount";
...
```

When looking at the template provided with the authoring guide, the section containing the *alternative flows* are on the sixth position in the first hierachy level and the headline is *"Alternative Flows"*. This is represented in the `selectionConditionBlock` of the described section `alternative_flows`. The `selectionConditionBlock` of the `Section` `alternative_flow` checks for a headline matching the regular expression `".*"`, which describes a sequence of arbitrary characters. This check always yields true, which is intended, as we want the `Condition [alternative_flow_title]` to be applied to each subpart of the document part containing the *alternative flows*. Without the `selectionConditionBlock` the corresponding document part would be determined by the structure in which the `Section`s are described, so only the first subpart would correspond to the `Section` `alternative_flow`

This rule can be describe in RSL, therefore it falls into *Category 1*.

Rules 4 to 7 - Other Identifiers

Rule 4:
Subflows: SF<nn> <name> P-SF<nn> <name>
Each Subflow has an ID and a Name. [. . .]
The identifier for private subflows consists of the acronym "SF" followed by a double-digit number.
The identifier for public subflows consists of the acronym "P-SF" followed by a double-digit number.

Rule 5 to 7 are similar to rule 4, they define how a special requirement, system-wide requirements and business rules should be named and referenced. We were not able to formulate all these rules in RSL, because neither the authoring guide nor the corresponding template defined, where in the requirements artifact these objects are documented. References to them can only be identified by using the specified syntax, but as for rule 2 (Title of the Basic Flow) this would defeat the purpose of checking for conformance to that syntax. We assign these rules to *Category 4*, not suited for automation.

Rule 25 - Step Numbering

Number the steps of the flows.
Extension points are not numbered.
[Extension points are written in curly brackets.]

```
section basic_flow
{
  ...
  text:
    // rule 25
    if not Format(Numbered)
      and not TextMatches("{.*}")
      add finding "Step Numbering"
      with message "Each step of the basic flow should be numbered.";

    if TextMatches("{.*}")
      and Format(Numbered)
      add finding "Numbered Extension Points"
      with message "Extension Points should not be numbered";
...
```

To verify that rule 25 is met, we take all texts which are not numbered and check if they look like an extension point. Extension points should always be in curly brackets and boldface. As we do not access the font style, we can only check, whether the text of the paragraph is written in curly brackets. We also verify, that texts that look like an extension point are not numbered, as this is also part of the rule description. This rule falls into *Category 1*.

Rule 28 - Basic Flow in Bulleted Outline

"Bulleted Outline" is the most condensed form of a Use Case Narrative.
Typically the steps of the basic flow only consist of a verb and an object. The actor is in most cases not identified explicitly. Only the names of the alternative flows are listed.

```
section basic_flow
{
  ...
  text:
    ...
    // rule 28, for bulleted outline use case narrative
    if TextMatches(".*\n.*")
      add finding "Too detailed Step"
      with message "A step in the basic flow of a use case in 'Bulleted Outline'
    should be short and only consist of a verb and an object.";
...
```

We described this rule in RSL, by creating a smell finding, whenever a text consisted of more than one line. This is a simplification, as we do not have access to natural language metadata with RSL. We can not check, whether the single line only consist of a verb and an object. This rule falls into *Category 2*. The last part of the description ("Only the names of the alternative flows are listed.") is covered by the implementation of rule 35 further down.

Rule 29 - Basic Flow in Fully Described

"Fully Described" is the most elaborated form of a Use Case Narrative.
In "Fully Described" it often improves readability when each step has a short descriptive title.

```
section basic_flow
{
  ...
  text:
    ...
    // rule 29, for fully described use case narrative
    if not TextMatches(".*\n.*")
      add finding "Step missing descriptive title"
      with message "In 'Fully Described' use cases it often improves readability when
     each step has a short descriptive title.";
...
```

This rules description is opposite to the one for rule 28. The authoring guide we use for our evaluation, distinguishes two different types of use case documentations in reference to how detailed the steps are described. As RSL can not distinguish in which of the types the document is written, one would have to use two different analysis definitions and select the appropriate one for each project, according to which use case template they use. We assign *Category 2* to this rule.

Rule 30 - Extension Points for Alternative Flows

If you want to write an Extension Point for jumping to or back from an alternative, always put the Extension Point above the step for which you want to specify an alternative flow.
Extensions points are written in curly brackets and bold face like this: **{Extension Point}**

Similar to rules 4-7, as we have no way of determining when an extension point is created other than the specified syntax, we can not check, whether the syntax was followed every time the author intended to define an extension point. This rule therefore belongs to *Category 4*.

Rule 35 - Alternative Flows in Bulleted Outline

In the level of detail "bulleted outline" alternative flows only have an ID and a name.

```
section alternative_flows found by
headline:
  if TextMatches("Alternative Flows");
section:
  if HierachyLevel(1)
    and HasChildPosition(>5);
{
  text:
    // rule 35, for bulleted outline
    -> [alternative_flow_title];

  // rule 35, for bulleted outline
  all sections:
  section:
    if HasChildPosition(>=0);
      add finding "Described Alternative Flow in 'Bulleted Outline'"
      with message "In the level of detail 'bulleted outline' alternative flows only
    have an ID and a name.";
...
// rule 3
[alternative_flow_title]
if not TextMatches("AF\\d\\d .*")
  add finding "Alternative Flow Title"
  with message "Always start the name of an alternative flow with an identifier. The
    identifier consists of the acronym 'AF' followed by a double-digit number:
    Example: AF01 Handle the Withdrawal of a Non-Standard Amount";
```

For this rule, we first test if the texts of the `alternative_flows Section` conform to the title syntax from rule 3. As we already defined this smell for another rule, we reused it by giving it a name and referencing it instead of rewriting the `Condition`.

Further we checked if the `alternative_flows Section` had any subparts (parts that are headlines, because they have subparts themselves), and if yes, annotated them with a respective finding. This rule falls into *Category 1*.

Rule 36 - Structure of Alternative Flows

Common structure for describing alternative flows
At a level of detail greater than "bulleted outline", alternative flows have a common structure [. . .] :
AF<nn> <name of the alternative flow>
At Extension Point if <condition>
1. <step 1 of the alternative flow>

. . .
5. Resume the use case from Extension Point
There are several options for describing the start of an alternative flow:

- *If the alternative flow can start at multiple places, describe the triggering condition like this: At extension point 1, extension point 2 or extension point 3 if <condition>*
- *If the alternative flow can start at an arbitrary placebetween two extension points, describe the triggering condition like this: Between extension point 1 and extension point 2 if <condition>*
- *If the alternative flow can start at an arbitrary place all over the use case, describe the triggering condition like this: At any time if <condition>*
- *There are several options for describing resuming: Resume the basic flow from extension point Resume AF05 at extension point Resume the SF02 at extension point*

```
// for fully described use case narrative
section alternative_flow found by
headline:
  if TextMatches(".*");
{
  ...
  text:
    // rule 36
    if not Format(Numbered)
      and if not TextMatches("At {.*} if .*")
      and if not TextMatches("Between {.*} and {.*} if .*")
      and if not TextMatches("At any time if .*")
      and if not TextMatches("Resume (the basic flow|AF\\d\\d|the SF\\d\\d) from {.*}
    ")
      add finding "Malformed Alternative Flow description"
      with message "Please refer to rule 36 of the Use Case Authoring & Review Guide
    for explanation.";
...
```

We assign this rule to *Category 1*,

Rule 41 - Step Pattern

Describe the flow of steps, not only the functionality.
To enforce this, make yourself start every step description with
"The <actor> . . . "
or
"The <system> "

```
// rule 41, for fully described use case narrative
section flow found by
headline:
  if TextMatches("Basic Flow|AF .*");
{
  text:
    if not TextMatches("The .*")
    and if not TextMatches("{.*}")
      add finding "Please describe flow of steps, not only the functionality."
      with message "To enforce this, make yourself start every step description with
    'The <actor> ...' or 'The <system> ...'.";
}
```

As we can not determine, whether a phrase really describes the actor or the system, we could only describe the structural part of this smell. Therefore this rule falls into *Category 2*.

Rule 49 - Use Case for CRUD Operations

If a Use Case's main goal is the handling of Create-Read-Update-Delete (CRUD) operations of data, start the name of the use case with "Maintain" followed by the name of the data.
Do not tear the CRUD operations apart in different use cases.

```
section document
{
  text 1:
    ...
    // rule 49
    if TextMatches("(Create|Read|Update|Delete) .*")
      add finding "Individual Use Case for CRUD Operation"
      with message "Do not tear Create-Read-Update-Delete (CRUD) operations apart in
    different use cases. Combine them and start the name of the use case with '
    Maintain' followed by the name of the data.";
...
```

This rule can be completely described in RSL and therefore belongs to *Category 1*.

Rule 52 - Number of Steps in Basic Flow

The Basic Flow should consist of 3 to 9 steps.

```
section basic_flow
{
  ...
  // rule 52
  section:
    if NumberOfSubsections(<3)
      add finding "Not enough steps"
      with message "Please make sure your scenario is big enough for a separate use
    case.";

    if NumberOfSubsections(>9)
      add finding "Too many steps"
      with message "Please make sure that each step is important enough to be a
    proper step in the flow.";
}
```

We assgin this rule to *Category 1*.

Non-structural Rules Described by RSL

While executing our study, we found two rules that we could express with RSL, although they do not fit our definition of a structural smell.

Avoid compound sentences
Avoid compound sentences like " .and ..and ..if .then ..and ..where "

```
all sections:
  text:
    ...
    // rule 23, not structural
    if TextMatches(" and | if | then | where ")
      add finding "Compound sentence"
      with message "Avoid compound sentences like ' ..and ..if ..then ..where '.";
```

Use "Validate", not "Check"
"Check" is not a good action verb and leads to "If check passes " and "If check fails " steps (the latter is an alternative flow). Instead use "validates", "ensures" or "establishes" and specify the alternatives in the section of alternative flows of the use case.

```
all sections:
  text:
    // rule 24, not structural
    if TextMatches(" Check | check ")
      add finding "Use 'Validate', not 'Check'"
      with message "'Check' is not a good action verb and leads to 'If check passes '
     and 'If check fails ' steps (the latter is an alternative flow). Instead use '
    validates', 'ensures' or 'establishes' and specify the alternatives in the
    section of alternative flows of the use case.";
```

We could describe these rules, because certain words were prohibited and these could be identified by using a regular expression. However, words can be inflected and therefore change the exact letters they are made of, which would not be taken into account by the regular expression. This is the reason we did not classify these rules as structural.

Lemmatization, a natural language processing techniques that maps inflected words to their base word, could be used to combat the issue of inflected words. One possibility would be to extend the `Rules` provided by RSL with some that leverage the annotated meta data from natural language processing techniques. We talk about further extensions to our language in Section 6.2.1.

4.5. Study Results

These are the results of our study:

Category 1: Completely describale with RSL: 8 out of the 16 selected, structural rules, 50%
Category 2: Partly describable with RSL: 3 out of 16, 19%
Category 3: Not describable with RSL, but suited for automatic detection: 0 out of 16
Category 4: Not suited for automatic detection: 5 out of 16, 31%

In addition to that, we found that we were able to describe 2 of the remaining 37 non-structural rules.

4.6. Interpretation

Our results show, that RSL is capable of describing over two thirds of the structural rules from the selected authoring guide.

Answer to RQ1: To which extent can RSL describe checks for conformance to structural authoring guidelines?
In total, we were able to describe 11 rules (69%) out of the 16 structural rules we selected from the authoring guide. Some of these rules (those falling into *Category 2*) had a structural and a non-structural component, where we only described the former one.

Answer to RQ2: Which rules can not be described in RSL and why?
The five rules we were not able to describe mostly lacked a clear structural location where they have to be applied or required semantic knowledge of the text to determine this location. One could argue that these rules therefore are not suited for automation. In addition, we could only partly describe three rules, as they are actually a combination of several rules, where some fall into our definition of structural and some require additional natural language metadata.

4.7. Threats to Validity

There are a number of threats to the validity of our study that evaluates RSL.

Internal validity A threat to the internal validity of our study is, that the *Use Case Authoring & Review Guide*, from which we selected our study objects, heavily influenced the development of RSL as one of the primary sources of possible structural smells. A second threat is, that the language designers are the same persons, who also conducted the study. This leads to a less objective study, as results immediately influence corrections and improvements to the language. A third threat is, that we had to interpret some of the rules to express them in RSL. For example, rule 29 describes, that each step of the flow should have a short descriptive title. Considering the template corresponding to the guideline we employed in our study, this short descriptive title is separated by a line break from the rest of the step description. Therefore we check, whether a line break occurs in every step. This is an interpretation of the rule, which is needed in order to automatically detect it However this introduces the threat, that the actual rule is not described and then checked adequately.

External validity The results of our study can not necessarily be generalized for all requirements artifact authoring guidelines used in the industry, as we only employed a single authoring guideline from the insurance domain for our study.

5. Related Work

In the following we discuss the state of the art on automatic analysis of requirements artifacts.

5.1. Automatic Detection of Quality Defects

Femmer et al. [FUG17] conducted an extensive case study on which proportion of writing guidelines can be checked automatically. They collected around 130 rules from the requirements guidelines of the Swedish Transport Agency. These were introduced to quality assure their specifications. The rules were classified into four different types: lexical, grammatical, structural and semantic rules, as well as into five levels of detectability with automated methods, ranging from perfectly detectable to not detectable at all.

They found, that about 50% of the considered rules could be analyzed deterministically or at least with a good heuristic, and that there is an especially high detection accuracy for structural rules. This is because the necessary information for checking them concerns mostly formatting options or pure text. Semantic information, which is often hard for an algorithm to obtain, is rarely needed. They also observed that the main reason for rules not to be automatically detectable is that the respective rules are vague and formulated imprecisely or unclearly.

Further research on automatically detecting faults in requirements was done by Lucassen et al. [Luc+15]. They presented an improved notion of user story quality, as well as a conceptual model of user stories, which captures the compact and strict template user stories follow these days. Within this conceptual model they collected a set of properties that indicate low-quality user stories and can be detected by an automated analysis prototype they presented. Arora et al. [Aro+15] demonstrate a tool-supported approach for automatically verifying conformance to two popular requirement templates, namely Rupp's and EARS. For the analysis they employed a natural language processing technique called text chunking. They performed several case studies that show that text chunking provides an accurate approach for verifying requirements template conformance.

Femmer et al. focused on automatic detection of requirements smells in general, while in this thesis we aimed at detecting structural smells specifically. Lucassen et al. and Arora et al. considered specific kinds of requirements documentation techniques (user stories, Rupp's and EARS templates), whereas RSL is designed to be independent from any predefined requirements template.

5.2. Using Requirements Guidelines For Quality Assurance

Femmer et al. [Fem+17] applied the principle of code smells to requirements and investigated how the resulting requirements smells can be used to immediately detect defects when the requirements are documented.

They developed a prototype for the automated detection of smells within requirements artifacts, called Smella. It parses the document into single text items and annotates the resulting sentences with meta-information from various natural language processing techniques. They then identify findings based on the added meta-information or the text itself and display them to the author in a spellchecker style directly in the requirements artifact. They propose to incorporate the smell detection process at two stages into quality assurance, before sending the artifact out for review and while preparing a manual review. In both cases, the detected smells give the author or reviewer a hint which areas of the document might be problematic.

Femmer et al. also emphasize that a major advantage of their smell detection is to increase the awareness of smells and support continuous feedback between the requirements author and the QA engineer.

Femmer et al. conducted a case study with three industrial and one university project, in which their tools showed an average precision of 59% and an average recall of 82% over all smells and projects. After providing empirical evidence from manual inspection, independent manual reviews and interviews with practitioners, they concluded that requirements smells can point to relevant defects across different types of requirements artifacts from different domains.

They again looked on requirements defects and smells in general, though we focused especially on structural ones.

In the area of requirements guidelines, research was also done by Cox et al. [CPS01], who developed the CREWS Use Case Guidelines and Achour et al. [Ach+99], who evaluated them. Alistair Cockburn laid the foundation for many use case writing guidelines with his book "Writing effective Use Cases" [Coc00]. These guidelines are created for general use cases in any domain. Most of the rules they contain formulate weak structural rules which heavily depend on the content of certain sections, like this rule from the CREWS Use Case Guidelines:

> *"Iterations and concurrent actions can be expressed in the same section of the UC, whereas alternative actions should be written in a different section."*

To check whether this rule is followed, an algorithm would have to understand whether a document part describes a concurrent or an alternative action, which requires a semantic understanding of natural language algorithms can not yet aquire [CW14]. Therefore we could not employ these guidelines in our study.

5.3. Structural Defects in Requirements Artifacts

Parachuri et al. [PSS14] performed a study to determine the kinds of structural defects that occur in use cases. They collected 380 use cases from an industrial partner, which were partly created for internal company projects and partly for external client projects. From existing papers about quality defects, they collected a set of common defects in requirements, leaving out any non-structural ones. However they did not specify how they defined "structural" defects further than that they "can be detected using direct measurement of structural features" [PSS14]. They performed several statistical tests on the presence of various structural defects, which they identified by manual review. As a control sample, they also analyzed a group of "ideal" model use-cases, that follow Cockburn's use case guidelines [Coc00], to see whether their results were statistically significant.

In their study, defects like passive voice, negative sentences and too many alternative scenarios (which hint at analysis paralysis) are not present in the industrial use cases they analyzed. However, steps with a missing actor, a lack of focus on the customer and inputs or interactions that were not followed by a response of the system were widely observed. Furthermore a high number of inconsistencies and use cases with a lot of structural complexity were found in samples from projects developed for external clients. The authors did not collect any qualitative data on the structural defects they identified and could therefore not present any reasons for the defects.

Parachuri et al. set the focus for their study on structural defects, but only apply manual reviews for detecting them. In contrast to that, we centered our thesis around the automatic detection of structural smells.

5.4. Domain Specific Languages for Smells

In his bachelors thesis [Lan15], Simon Lang developed a domain specific language that is able to describe many natural language smells within requirements. He started by exploring the existing smell detection framework *Smella*, developed by Femmer et al., but also took a less technical view into account by looking at how non-programmers would model a smell and its description. The resulting language closely resembles natural language when read, but is constrained enough to be correctly parsed to complete smell definitions. He validated his language by showing that it can describe all requirements smells defined by Femmer at al. [Fem+17] and proposed possible extensions of his language, including adding structural aspects to the analysis.

Lang's language is similarly to ours capable of describing smells, but targets smells detectable through natural language processing techniques. Our domain specific language will employ a different set of information for its smell detection, namely structural and formatting data and unprocessed text.

6. Conclusion

6.1. Summary

Requirements are the basis for later development activities in software engineering. Because of the lack of precise syntactic and structural rules for natural language [PSS14], requirements artifacts written in natural language are prone to critical quality defects. To combat these defects, companies use manual reviews. These can be supported by automatic reviews, which can identify particularly troublesome areas beforehand. To ease the review process, companies standardize the format of their requirements artifacts and an automatic analysis for adherence these structural rules simplifies the feedback to the requirements author, whether his artifact is correctly structured or not.

To our knowledge, there is no standardized way to define structural guidelines for requirements artifacts so that artifacts can be automatically quality assured. In this thesis, we presented a domain specific language that empowers the user to define an automated analysis for structural issues within a requirements artifact.

We described the current structural analysis definition in Qualicen Scout and motivated the use of a domain specific language, as it abstracts from the analysis language, enables to focus on the crucial concepts for defining a structural analysis and is more safe than just executing external code. Next we introduced the concept of requirements smells and defined our notion of structural smells. We collected types of structural smells from literature and presented the existing smell detection process of Qualicen Scout.

We explained the decision approach on which checks our language can define in a structural analysis. The selected checks are mostly based on the tree structure of a document, formatting information and the plain text. In the following we presented our abstract syntax to show which information is encoded in RSL and our concrete syntax to illustrate how the information is encoded.

To evaluate the language designed, we conducted a study in which we expressed structural rules from an industrial use case authoring guideline in RSL. We found hat we could express 69% of the selected rules, while in the remaining ones, structural templates were often mixed with semantic meaning to determine the location where the template should be applied. As we limited our language to describing smells that can be detected without semantic understanding of the artifact, such rules can not be described in RSL.

6.2. Future Work

In the following we propose extensions to the language developed in this thesis and present aspects to be covered in further research.

6.2.1. Technical Work on RSL

RSL was developed with modularity in mind, to make extensions with new features easier. Here we want to present some ideas on what could be additionally incorporated into RSL:

- **NLP Rules:** The provided rules (like `TextMatches("<RegEx>")`) can be extended with further ones that use natural language processing techniques, e.g., `WordHasLemma("<dict. form>")`.

- **Nonrecursive Section Search:** At the moment, if a `Section` defines a `selectionCondition-Block`, the analysis has to match the selection conditions with every document part on the same hierarchy level, as well as all subparts of these. This should simplify the declaration of sections, as the user does not need to find out the correct hierarchy level on which to define his section. However this can lead to unwanted matches further down in the hierarchy. A new keyword `nonrecursive` could be annotated to sections so they are only searched for in the hierarchy level they are defined in.
- **Analysis Configuration:** In the authoring guide we used for our study there were two types of use case documents that differed in how detailed each step has to be described. We proposed that a different structural analysis has to be written in RSL for each use case type and always be applied to the correct type. A useful extension would be to enable the user to turn certain smell describing conditions on/off during analysis, based on the content of the document. For the use cases written according to the *Use Case Authoring & Review Guide*, which we employed in our study, there could be a location that states whether the particular use case is a "bulleted outline" or "fully described". RSL could provide an `Action` that defines a boolean variable, which can then be used as a rule to decide in any `Condition` if this `Condition` should be applied further.
- **References and Glossaries:** Two inspections that companies ask for are correct references and a correct format of glossary terms. At the moment, RSL has no functionality to check these, as it has no notion of a glossary containing words that have to be treated specially.
  An extension could look as follows: The user can define multiple glossaries (identified by their name) and, with a new `Action`, add words to these at the end of a `Condition` chain. With an additional `Rule` they can check whether a word is contained in a certain glossary or not.
  Correct references inside a document and glossaries which are also defined in the same document could be expressed like that. As RSL as of now only defines analyses of one document at a time, references to other documents or external glossaries would need greater extensions.

6.2.2. Studies

In this thesis we presented a small study on the expressivity of RSL. We suggest to extend the theoretical work in several further studies:

- **Atomic Rules:** As we described in Section 4.6, several rules from our study were not atomic, but contained several subrules that have to be checked independently. Also, some of the subrules fell in our definition of structural rules, while others needed additional metadata from natural language processing techniques. For a future study, we propose to split each rule into its subrules and evaluate these independent from the rest of the rule. This would yield a more detailed result on which proportion of the rules requirements authoring guidelines are structural rules, as well as refine the evaluation on how many actual rules can be described with RSL.
- **Diverse Authoring Guidelines from Different Domains:** For the study performed in this thesis, we took authoring guidelines for requirements that are used in an insurance company. To show that RSL can describe analyses for structural smells throughout requirements artifacts from different domains, we propose to replicate our study with more diverse authoring guidelines from other domains.
- **Usability Study:** Another important aspect for research would be the usability of RSL for practitioners. As the designers of RSL were programmers, it still closely resembles the popular programming languages. Therefore, it is important to test how easy it is for specialists, which do not have a background in the IT domain, to learn and use RSL.

Bibliography

[ABL89] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. "Software Inspections: an Effective Verification Process". In: *IEEE Software* 6.3 (1989), pp. 31–36.

[Ach+99] Camille Ben Achour et al. "Guiding Use Case Authoring: Results of an Empirical Study". In: *Proceedings of the IEEE International Symposium on Requirements Engineering*. (1999), pp. 36–43.

[Aro+15] Chetan Arora et al. "Automated Checking of Conformance to Requirements Templates Using Natural Language Processing". In: *IEEE Transactions on Software Engineering* 41.10 (2015), pp. 944–968.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2000.

[CPS01] Karl Cox, Keith Phalp, and Martin Shepperd. "Comparing Use Case Writing Guidelines". In: *7th International Workshop on Requirements Engineering: Foundation for Software Quality*. (2001).

[CW14] Erik Cambria and Bebo White. "Jumping NLP Curves: a Review of Natural Language Processing Research". In: *IEEE Computational Intelligence Magazine* 9.2 (2014), pp. 48–57.

[FB99] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[Fem+17] Henning Femmer et al. "Rapid Quality Assurance with Requirements Smells". In: *Journal of Systems and Software* 123 (2017), pp. 190–213.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[FUG17] Henning Femmer, Michael Unterkalmsteiner, and Tony Gorschek. "Which Requirements Artifact Quality Defects are Automatically Detectable? A Case Study". In: *Fourth International Workshop on Artificial Intelligence for Requirements Engineering*. AIRE. IEEE, (2017), pp. 1–7.

[Jue+10] Elmar Juergens et al. "Can Clone Detection Support Quality Assessments of Requirements Specifications?" In: *ACM/IEEE 32nd International Conference on Software Engineering*. (2010), pp. 79–88.

[Lan15] Simon Lang. "A Description Language for Natural Language Smells". Master's thesis. Technische Universität München, 2015.

[Luc+15] Garm Lucassen et al. "Forging High-Quality User Stories: Towards a Discipline for Agile Requirements". In: *IEEE 23rd International Requirements Engineering Conference*. (2015), pp. 126–135.

[Ott12] Daniel Ott. "Defects in Natural Language Requirement Specifications at Mercedes-Benz: An Investigation Using a Combination of Legacy Data and Expert Opinion". In: *20th IEEE International Requirements Engineering Conference*. (2012), pp. 291–296.

[PSS14] Deepti Parachuri, A. S. M. Sajeev, and Rakesh Shukla. "An Empirical Study of Structural Defects in Industrial Use-Cases". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM. (2014), pp. 14–23.

[Wes02] James Christopher Westland. "The Cost of Errors in Software Development: Evidence from Industry". In: *Journal of Systems and Software* 62.1 (2002), pp. 1–9.

A. Complete Smell Description

This is the complete structural description in RSL we created for our study of the *Use Case Authoring & Review Guide*.

```
all sections:
  text:
    // rule 24, not structural
    if TextMatches(" Check | check ")
      add finding "Use 'Validate', not 'Check'"
      with message "'Check' is not a good action verb and leads to 'If check passes '
     and 'If check fails ' steps (the latter is an alternative flow). Instead use '
    validates', 'ensures' or 'establishes' and specify the alternatives in the
    section of alternative flows of the use case.";

    // rule 23, not structural
    if TextMatches(" and | if | then | where ")
      add finding "Compound sentence"
      with message "Avoid compound sentences like ' ..and ..if ..then ..where '.";

section document
{
  text 1:
    // rule 1
    if not TextMatches("[A-Z]+-UC\\d\\d .*")
      add finding "Incorrect Use Case Title"
      with message "The title of a use case consists of an identifier and a name. The
     name always starts with an active verb, in most cases followed by an object. The
     name describes the goal of the primary actor. The identifier consists of a
    product prefix, the acronym '- UC' and a double-digit number. Examples: DIY-UC01
    Browse and Shop, DIY-UC02 Checkout, DIY-UC03 Manage Account";

    // rule 49
    if TextMatches("(Create|Read|Update|Delete) .*")
      add finding "Individual Use Case for CRUD Operation"
      with message "Do not tear Create-Read-Update-Delete (CRUD) operations apart in
    different use cases. Combine them and start the name of the use case with '
    Maintain' followed by the name of the data.";

  section intro { }

  section brief_descripition { }

  section preconditions { }

  section basic_flow
  {
    headline:
      // rule 2
      if not TextMatches("Basic Flow")
        add finding "Basic Flow Title"
        with message "This section should contain the basic flow of events and should
     always be titled 'Basic Flow'.";

    text:
      // rule 25
      if not Format(Numbered)
        and not TextMatches("{.*}")
        add finding "Step Numbering"
        with message "Each step of the basic flow should be numbered.";

      if TextMatches("{.*}")
        and Format(Numbered)
        add finding "Numbered Extension Points"
        with message "Extension Points should not be numbered";

      // rule 28, for bulleted outline use case narrative
      if TextMatches(".*\n.*")
        add finding "Too detailed Step"
      with message "A step in the basic flow of a use case in 'Bulleted Outline'
  should be short and only consist of a verb and an object.";

    // rule 29, for fully described use case narrative
    if not TextMatches(".*\n.*")
      add finding "Step missing descriptive title"
      with message "In 'Fully Described' use cases it often improves readability
  when each step has a short descriptive title.";

  // rule 52
  section:
    if NumberOfSubsections(<3)
      add finding "Not enough steps"
      with message "Please make sure your scenario is big enough for a separate use
   case.";

    if NumberOfSubsections(>9)
      add finding "Too many steps"
      with message "Please make sure that each step is important enough to be a
  proper step in the flow.";
}

section alternative_flows found by
headline:
  if TextMatches("Alternative Flows");
section:
  if HierachyLevel(1)
    and HasChildPosition(>5);
{
  text:
    // rule 35, for bulleted outline
    -> [alternative_flow_title];

  // rule 35, for bulleted outline
  all sections:
  section:
    if HasChildPosition(>=0);
      add finding "Described Alternative Flow in 'Bulleted Outline'"
      with message "In the level of detail 'bulleted outline' alternative flows
  only have an ID and a name.";

  // for fully described use case narrative
  section alternative_flow found by
  headline:
    if TextMatches(".*");
  {
    headline:
      // rule 3
      [alternative_flow_title]
      if not TextMatches("AF\\d\\d .*")
        add finding "Alternative Flow Title"
        with message "Always start the name of an alternative flow with an
  identifier. The identifier consists of the acronym 'AF' followed by a
  double-digit number: Example: AF01 Handle the Withdrawal of a Non-Standard Amount
  ";

    text:
      // rule 36
      if not Format(Numbered)
        and if not TextMatches("At {.*} if .*")
        and if not TextMatches("Between {.*} and {.*} if .*")
        and if not TextMatches("At any time if .*")
        and if not TextMatches("Resume (the basic flow|AF\\d\\d|the SF\\d\\d) from
  {.*}")
        add finding "Malformed Alternative Flow description"
        with message "Please refer to rule 36 of the Use Case Authoring & Review
  Guide for explanation.";
  }
}

// rule 41, for fully described use case narrative
section flow found by
headline:
  if TextMatches("Basic Flow|AF .*");
{
    text:
      if not TextMatches("The .*")
      and if not TextMatches("{.*}")
        add finding "Please describe flow of steps, not only the functionality."
        with message "To enforce this, make yourself start every step description
    with 'The <actor> ...' or 'The <system> ...'.";
  }
}
```