



Universiteit  
Leiden  
The Netherlands

# Computer Science & Economics

Transforming natural language rules  
into executable code at run-time

Max van Doorn  
s2353350

Supervisors: Dr. G.J. Ramackers & Prof.dr.ir. J.M.W. Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)  
[www.liacs.leidenuniv.nl](http://www.liacs.leidenuniv.nl)

03/08/2022

## **Abstract**

Since the dawn of the Information Age, all facets of business are rapidly digitizing. While having accomplished impressive feats, the craft of software engineering still faces many challenges. In this thesis, business stakeholder involvement is wielded to tackle one significant pitfall: the misalignment of software functionality with the end-user's needs. To enable this, a software component is built that transforms business rules to Python source code. Firstly, a transformation framework applies Natural Language Processing techniques to map business rules, defined in natural language, to a formal rule language. Then, formal rule strings are converted to source code and inserted into a Django run-time prototype environment for instant feedback provision. In support of this endeavour, a taxonomy of business rules, together with a mapping to implementation types, is presented. Finally, the usability and effectiveness of the developed software solution are evaluated with a small-scale user experiment. This research was conducted as part of the Next Generation UML research group at the Leiden Institute of Advanced Computer Science. Their aim is to facilitate effective business stakeholder contribution to the software development process by integrating Natural Language Processing techniques and UML specification models within an environment for rapid prototype generation.

## **Acknowledgments**

This thesis project was conducted as a contribution to the Next Generation UML research group at the Leiden Institute of Advanced Computer Science. Many thanks to dr. Ramackers and prof.dr.ir. J.M.W. Visser for supervising this thesis. Also, my gratitude for the support of the ngUML team; with a special thank you to Pepijn Griffioen, Bram van Aggelen, and Willem-Pieter van Vlokhoven.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contribution . . . . .	1
1.2	Thesis overview . . . . .	1
<b>2</b>	<b>Background &amp; Related Work</b>	<b>3</b>
2.1	Model Driven Engineering . . . . .	3
2.2	Natural Language Processing . . . . .	4
2.2.1	Recent trends . . . . .	4
2.3	The ngUML project . . . . .	6
2.4	Formal languages . . . . .	7
2.4.1	The structure of a formal language . . . . .	8
2.4.2	Backus-Naur form . . . . .	8
<b>3</b>	<b>Business rule taxonomy</b>	<b>10</b>
3.1	Taxonomy of rules: rational view . . . . .	10
3.2	Taxonomy of rules: specified view . . . . .	12
<b>4</b>	<b>System design</b>	<b>16</b>
4.1	Logical design . . . . .	16
4.1.1	System requirements . . . . .	16
4.1.2	System overview . . . . .	16
4.2	Technical design . . . . .	19
4.2.1	Validator implementation methods . . . . .	19
4.2.2	Mapping rules to implementation . . . . .	22
4.2.3	EBNF definition of the inclusive demarcation rule . . . . .	24
4.2.4	From natural language to structured rules . . . . .	27
4.2.5	From structured rules to code . . . . .	28
<b>5</b>	<b>Worked example</b>	<b>32</b>
5.1	Error messaging . . . . .	35
<b>6</b>	<b>Validation</b>	<b>37</b>
6.1	Results . . . . .	37
<b>7</b>	<b>Further research</b>	<b>39</b>
<b>8</b>	<b>Conclusions</b>	<b>40</b>
	<b>References</b>	<b>41</b>
<b>A</b>	<b>User-experiment document</b>	<b>43</b>

# 1 Introduction

Since the creation of FORTRAN as the first general-purpose programming language in the 1950s [5], the world has been set on a course of increased digitization. As information processing devices like the laptop and smartphone are reaching a state of ubiquity, the general quality of software solutions is increasing in lock step. However, challenges remain, with 19% of software projects failing in 2006 [9]. Much research has been conducted on the pitfalls of software engineering, and a list of signs that prelude project failure is compiled in [28]. These include: “Project managers don’t understand user’s needs”, “The project’s scope is ill-defined”, “Project changes are handled poorly” and “Users are resistant” (p. 19). Their common denominator is an inadequate alignment of the provided solution with the user’s needs. Many contributions have been articulated to address this challenge, ranging from requirement elicitation methods to entire software development frameworks (e.g., SCRUM). However, these solutions did not challenge the foundation of software development, namely how source code is written.

Next Generation UML (ngUML) is a research group at the Leiden Institute of Advanced Computer Science. It was founded by dr. Ramackers with the aim to ease the participation of non-IT domain experts in the software development process through a re-imagination of this process. By extending a Model Driven Engineering approach with natural language processing capabilities, the software engineering process is structured with accessibility and clarity at its core. The ngUML group is working on a software solution that converts software requirements that are defined in natural language to a prototype application at run-time. Furthermore, prototypes are mapped to specification models expressed in the Unified Modeling Language (UML) standard, such that these models are usable to directly alter the prototype’s source code.

## 1.1 Contribution

This bachelor thesis contributes to the ngUML project by extending the software with rule handling capabilities. Natural language processing functionality enables the integration of business rules, stated in natural language, to application prototypes at run-time. To realize this, a formal language is defined to unambiguously express these rules. To demonstrate the viability of this solution, the aforementioned functionality is implemented for a specific rule type. Furthermore, extensibility is prioritized in the architectural design of the rule component to ensure a future-proof result. Lastly, in an endeavor to guide future discourse, a taxonomy of the rules-domain is outlined with a mapping to implementation types in Django, the development framework of the ngUML engine.

## 1.2 Thesis overview

Section 2 outlines related work and relevant background information. In 3, a taxonomy of business rules is presented. Section 4 outlines the system design of the software component over multiple subsections. Firstly, the logical design is discussed in 4.1. Then, in section 4.2, the technical design is outlined. After categorizing various validator implementation techniques in 4.2.1, a mapping between these implementation types and the rules taxonomy is discussed in 4.2.2. In section 4.2.3, the language to formulate rules of a certain type is formally defined using Extended Backus-Naur Form. Then, in 4.2.4, the conversion of natural language rules to formal rule strings is discussed, which is followed by an outline of the transformation of these formal rule strings into source code

in 4.2.5. Finally, section 5 gives a worked example, 6 outlines the validation results of this project's deliverables, future research opportunities are presented in section 7, and the conclusion is discussed in section 8.

## 2 Background & Related Work

### 2.1 Model Driven Engineering

Model Driven Architecture (MDA), as defined by the Object Management Group in [19], “provides an approach for deriving value from models and architecture in support of the full life cycle of physical, organizational and I.T. systems” (p. 1). It is a set of standards that enables the unambiguous definition of models that represent systems at any level of abstraction. At the core of this paradigm are the Platform Independent Model (PIM) and Platform Specific Model (PSM). Through this partition, separation of concerns is supported regarding a system’s end goals and implementation. Since a PIM abstracts away implementation-dependent details, it enables the gathering of a more durable body of knowledge regarding the specification of the intended result. Modeling adds value to the software engineering process by aiding mutual understanding of systems in their current and desired state. In OMG’s vision, these models should also offer functionality to be queried, analyzed, validated, simulated, and transformed into other models [19]. Applying this approach would result in the predictable, reproducible, and efficient production of a software solution.

In the widely cited article by S. Kent, a framework is outlined for a Model Driven Engineering (MDE) method that integrates the tools and philosophy provided by MDA [23]. It’s firstly argued that, within a software development project, there may be many more modeling dimensions of interest besides abstraction. For example, ‘subject area’ specifies a purpose like inventory logistics or customer support. ‘Aspect’ is also mentioned, this dimension would specify system characteristics like distribution or security. Within the envisioned MDE method, applicable dimensions for the project at hand are selected where they serve as guiding parameters for model definition. It’s also proposed that models and their mutual mappings should be viewed as infrastructure to be implemented in a broader mode of operation. Both components affect each other, as the modus operandi dictate the priorities by which models should be created and used, while models enable the exploration and definition of systems that result in the effective formulation of operational methods. The value of an MDA-driven application is determined by the benefit of producing and maintaining model artifacts in relation to its costs. Kent argues that effective tools that enable, among other things, enforcement of well-formedness constraints, mapping between models, dashboard representation, and distributed working are essential for lowering the break-even point. Finally, the value of meta-level language definition is discussed. As MDA should be applicable in a wide variety of domains, different modeling languages and dialects should be available. A comprehensive meta-language for defining such dialects could enable the configuration of tools to specific application domains, enabling specialized functionality.

G. Mussbacher et al. argue in [2] that, while having been successfully adopted in a wide variety of industries, model-driven engineering still is a “niche technology” (p. 184). Considerable progress has been made in the domains of modeling languages, model analysis, and model transformation. Successful adoption has however been hampered because modeling methods do not hold up to the complex requirements of software engineering applications. Limited interoperability between models and tools, the complicated usage of these modeling tools, and the absence of a Body of Knowledge exacerbate this. A 2020 analysis of the state of the research expresses a similar attitude: much progress has been made on several fronts but big challenges remain [1]. Amongst other things,

light is shed on the increasing MDE agility, modeling of heterogeneous views, and “expressing . . . requirements in human-readable notation that can be understood by a computer program” (p. 9).

## 2.2 Natural Language Processing

The first research on natural language processing (NLP) dates back to the late 1940s with the first application being machine translation [24]. The method was rule-based: a dictionary was used for word translation whereafter a permutation scheme reordered the words according to the target language’s structure. Performance was very limited as semantic ambiguity and context-dependency were not addressed. In the 1990s, the statistical or corpus-based approach gained popularity as relatively simple implementations proved effective when trained on large data sets [20]. These methods used a ‘bag of words’ approach: only the collection of words was used to derive meaning, disregarding the overarching context. Today, such approaches still form a baseline that is not easily surpassed.

D. Liddy outlines the ‘levels of language’ approach, which delineates several levels of language understanding [24].

- Phonology: “the interpretation of speech sounds within and across words” (p. 6).
- Morphology: the interpretation of word components that represent the smallest units of meaning. An example approach is the dissection of a word in prefix, suffix, and stem.
- Lexical: the semantic interpretation of individual words. Various processing techniques can be applied to achieve this, like part-of-speech (POS) tagging and the replacement of words by a semantic equivalent.
- Syntactic: the interpretation of the grammatical structure within sentences to uncover the dependency relationships between words.
- Semantic: the disambiguation of word meanings by interpreting word-level interactions in a sentence.
- Discourse: the holistic interpretation of multiple sentences.
- Pragmatic: the inference of meaning derived from domain knowledge, as opposed to exclusively using the knowledge that is endemic to the source text.

### 2.2.1 Recent trends

In an overview study by Young et al. [33], the recent rise of deep learning based methods in natural language processing is detailed. Following the statistical approach to NLP, distributional vectors (word embeddings) rest on the distributional hypothesis, which states that “words with similar meanings tend to occur in similar context” (p. 2). This focus on word-context enables the capture of lexical meaning within a corpus of text. Word embedding based models like the popular CBOW and skip-gram by Mikolov et al. have attained state-of-the-art results in a variety of NLP tasks in the past. However, as the incorporation of context into word representations proved fruitful, individual word embeddings using a small interval of neighboring words became a limitation. Early word



embedding methods considered every sentence containing a certain word to create a generalized representation. This limits semantic understanding of words as many are polysemic (“She went *right*” vs. “She was *right*”). Furthermore, the small window-size (nr. of neighboring words analyzed) resulted in similar embeddings for opposing words that generally occur within a similar context (e.g., “good” and “bad”), impeding downstream tasks like sentiment analysis. Contextualized word embeddings tackle this problem by creating a different representation for every occurrence of the word, thus enabling a variety of interpretations. Renowned models using this type of embedding are ELMo and BERT, the latter of which outperformed state-of-the-art results “by a large margin” (p. 6). The study continues with an outline of deep learning methods that have been the focus of research over recent years.

Convolutional neural networks (CNN) enable contextualized word embeddings. Consecutive words (n-grams) are extracted from the corpus, generating higher-level features. An input sequence  $\{s_1, s_2, \dots, s_n\}$  of  $n$  words is encoded into a series of word embeddings  $\{v_1, v_2, \dots, v_n\}$ . As opposed to previous methods that handled sequences of single words, the extended window-size enabled the construction of abstract features with richer context, proving effective for several NLP tasks. However, maintaining long-distance contextual information was still difficult (e.g., “Chris is easy to consider it impossible for anyone but a genius to try to talk to *him*.” [7]).

A recurrent neural network (RNN) also encodes an input sequence, the differential characteristic being that the word embeddings depend on previous word embeddings within the same sequence. This architecture creates continuity in encoding and is therefore referred to as a form of memory. Word order can now be utilized to extract semantic meaning, improving on the aforementioned challenges of polysemy and the embedding of opposing words with similar contexts. Another upside to this method is the ability to process input of arbitrary length. RNNs are not necessarily superior to CNNs however, this depends on the application context.

Later, an attention mechanism was introduced for application within RNNs and likewise methods. This mechanism enabled the prioritization of input-sequence components, resulting in more selective encoding capabilities and closer representation of the input sequence in the output sequence. Dense or lengthy text could thus be handled more effectively, improving applications like summarization and translation. Vaswani et al. progressed on this method by substituting recurrence and convolution in the encoding step altogether in favor of attention mechanisms [4]. This new architecture was called a transformer, well-known for its effective Sesame Street-themed models (BERT, ALBERT, et cetera).

The structure of recursive neural networks, like RNNs, lends itself to the representation of language. The former can be presented as a constituency parsing tree where the representation of each internal node depends on the representation of its descendant nodes, enabling bottom-up syntactical representations of sentences (see figure 1). Thus, the characteristics of a recurrent neural network fit the hierarchical characteristics of language, while a RNN corresponds to its sequential aspects. Applications of recursive neural networks include parsing and sentiment classification.

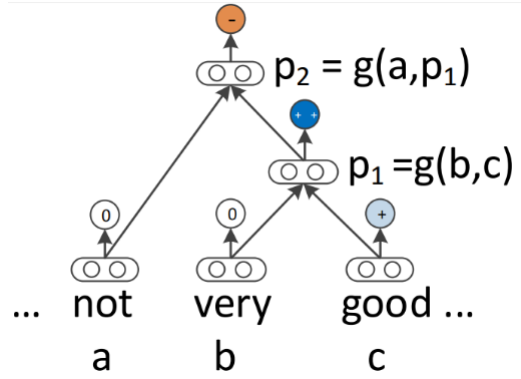


Figure 1: Sentiment classification using a recursive neural network. Source: [33]

## 2.3 The ngUML project

The ngUML project was established to provide an accessible approach to model and prototype generation from natural language requirements. Extending on the MDE framework, the project proposed ([8], p. 3) the usage of UML models “not just as design-time artifacts, but ... also deploying them as run-time artifacts alongside the application.” Thus, these models do not exclusively represent the software product’s design to the user, they are also directly mapped to the source code of a prototype application. Through the model editor, alterations to the UML diagram are reflected in the corresponding source code at run-time.

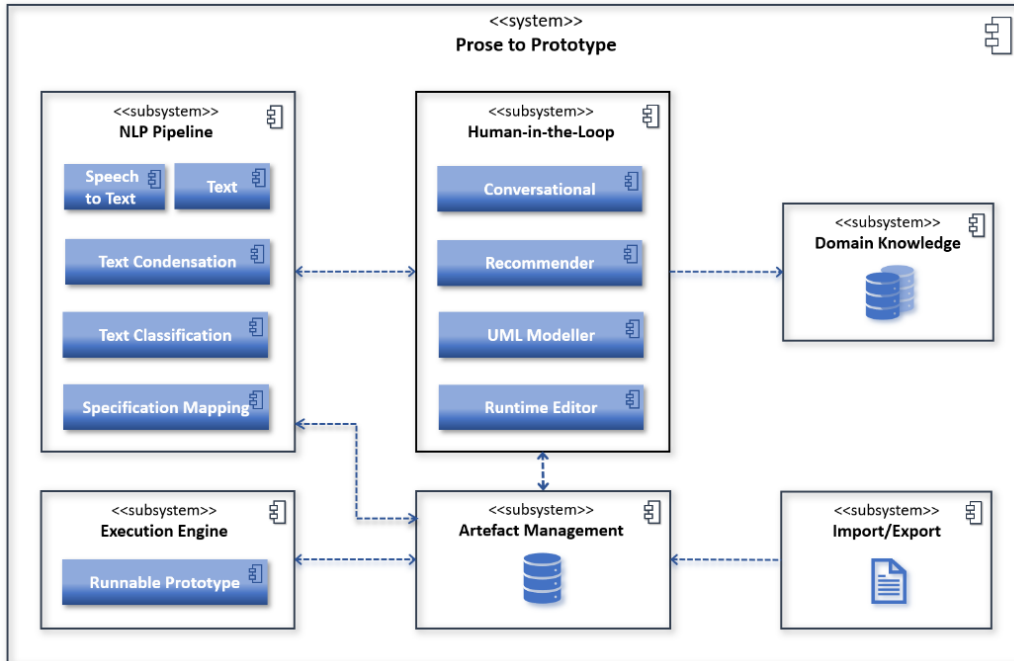


Figure 2: The envisioned ngUML architecture. Source: [27]

Natural language input is processed to compile ‘metadata’, data about models and their representation. This metadata is saved in the ‘metamodel’, a data store. Because natural language is inherently ambiguous, a human-in-the-loop approach is proposed in [27]. The aforementioned UML capabilities with its mapping to the source code already contribute to this as the model representation provides feedback to the user. Further progress will be realized by implementing, among others, conversational components (e.g., a chatbot) and recommender functionality. See figure 2 for an overview of the envisioned system architecture.

The workflow starts with a user submitting a natural language input in text or audio format. Natural language processing techniques are applied to the input to determine the UML diagram type that best represents it and to compile the entities that populate it. For example, “A customer has a name and address” would translate to a class diagram with a ‘Customer’ class that owns the properties ‘name’ and ‘address’. The extracted entities, together with their representation, form the metadata that is held in the metamodel. The UML diagram can be rendered in the front-end (see figure 3) and altered if needed. Any changes will be directly reflected on the metamodel and the prototype, enabling rapid iteration.

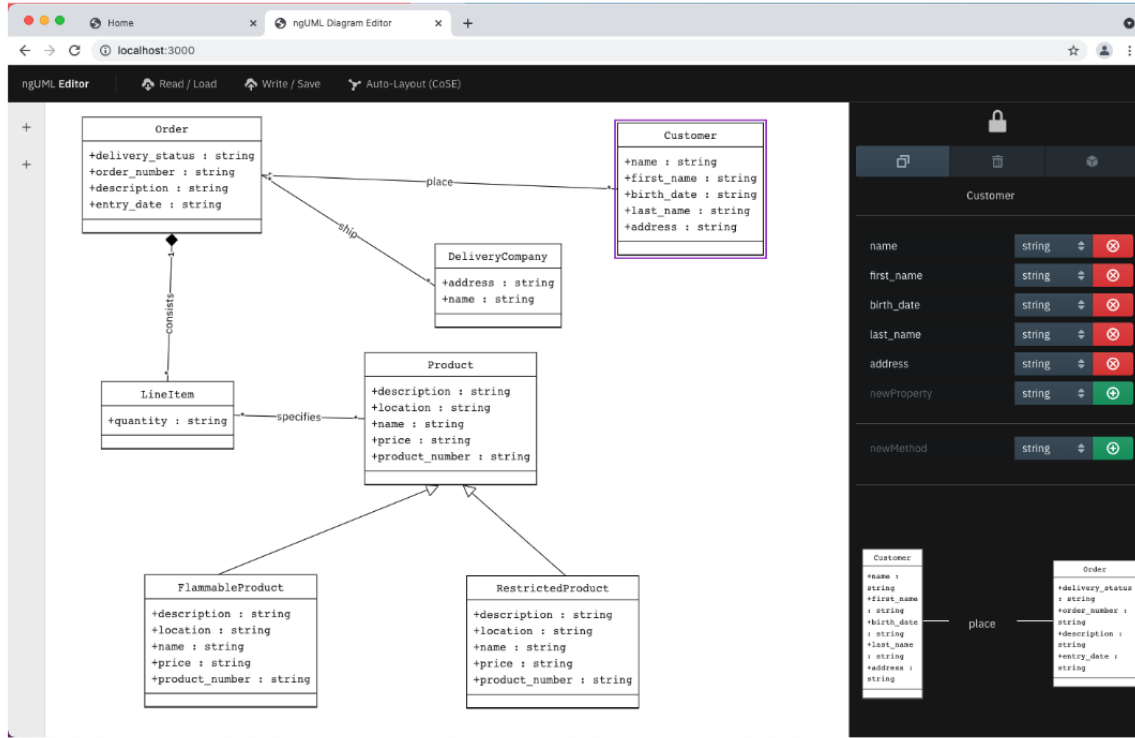


Figure 3: Example of a UML diagram in the front-end. Source: [27]

## 2.4 Formal languages

A formal language is an unambiguously defined structure whereby its products (strings), derived from an alphabet through a grammar, can be systematically checked for well-formedness (correctness).

Chomsky’s work on the precise characterization of the structure of natural languages has heavily influenced the contemporary field of formal language theory. Another important endeavor was the definition of ALGOL 60. This programming language improved on cornerstone concepts like recursion and nested functions, consequently paving the way for many others like Pascal and C. Interlinked with the rise of ALGOL 60 was the metasyntax Backus-Naur form (BNF), as it was used in the original report of the programming language [6].

### 2.4.1 The structure of a formal language

The following terminology is described in [29] by Moll et al. A formal language consists of terminal symbols, making up the characters of the produced strings, and non-terminal symbols, making up characters that may appear in derivations but not in the produced strings. A production is an inductive rule containing the aforementioned symbols. Every string in a given grammar begins with a start symbol. Finally, an alphabet is a nonempty finite set of symbols. All in all, a grammar (G) consists of a terminal alphabet (X), a non-terminal alphabet (V), a start symbol (S), and its productions (P). Strings that are produced using a grammar G in language L are well-formed in relation to that language. A useful example is given regarding a language of parentheses where every opened parenthesis should be closed. Here, G is defined as:

$$X = (, )$$

$$V = S$$

$$S = S$$

$$P = \{S \rightarrow (), S \rightarrow (S), S \rightarrow SS\}$$

Using G, the following string can be constructed:  $S \rightarrow SS \rightarrow (S)S \rightarrow (S)SS \rightarrow (())()()$

Also, an important division is made between context-free and context-sensitive languages. The former is a language that is generated using context-free grammar, meaning that in every production  $v \rightarrow w$ ,  $v$  is a single non-terminal symbol. For context-sensitive languages, this is not the case. Context-free definitions are widely utilized, like for programming languages.

### 2.4.2 Backus-Naur form

Before J. Backus formulated his metasyntax, programming languages like FORTRAN and IAL were defined by a combination of natural language components and some formative patterns. This method did not wield enough expressive power to consistently define simple control structures unambiguously, impeding downstream definition of more complex structural components like declarations [6]. After P. Naur altered Backus’s notation slightly to improve readability, BNF was created.

Taking figure 4 as an example, the following about BNF can be outlined. It consists of a set of terminal symbols, a set of non-terminal symbols, and a set of production rules in the form *Left Hand Side (LHS) ::= Right Hand Side (RHS)* where LHS is a single non-terminal symbol (ergo, context-free) and RHS is a sequence of symbols that may be terminal or non-terminal. ‘::=’ indicates that LHS must be replaced by the sequence RHS. The vertical bars indicate a choice between possible substitutions.

```

<adding operator> ::= + | -
<multiplying operator> ::= × | / | ↑
<primary> ::= <unsigned number> | <variable> | <function
designator> |
(<arithmetic expression>)
<factor> ::= <primary> | <factor> ↑ <primary>
<simple arithmetic expression> ::= <term> | <adding
operator><term> |
<simple arithmetic expression><adding operator><term>
<if clause> ::= if <Boolean expression> then
<arithmetic expression> ::= <simple arithmetic expression> |
<if clause><simple arithmetic expression> else <arithmetic
expression>

```

Figure 4: BNF definition of arithmetic expressions in the ALGOL 60 report. Source: [6]

Following the popularity of BNF, many slight variants were used in academic literature. In hopes of realizing a new standard, N. Wirth proposed a slightly extended variant, later named Extended BNF (EBNF). The additions did not improve BNF’s expressivity, rather they eased the expression process. Most notably, it facilitates an explicit iteration construct that obviates verbose use of recursion [31]. Wirth’s attempt was to no avail, as there are many dialects of EBNF and BNF in use today [34]. In this paper, the rule language will be defined using the ISO/IEC 14977 [22] EBNF standard.

### 3 Business rule taxonomy

Rules form the heart of this project, so a charting of the rule domain is essential. The ngUML solution is developed to ease the involvement of non-IT domain experts within the software engineering process. It, therefore, handles high-level business rules as well as their low-level implementation. In the following chapters, a taxonomy of rules is presented that exists on both levels.

The rules that govern an organization are defined in [32] as all rules that govern its conduct and procedures. This ruleset encompasses business rules, rules that can be altered by the business itself, as well as regulation, rules that are exogenously enforced by governing bodies. Furthermore, it's argued that even the laws of physics are to be considered when holistically modeling the ruleset that governs an organization: a timetabling application should enforce that every person can only be in one place at the same time. Also, a distinction is made between descriptive and prescriptive rules. Descriptive rules describe reality as it is ("the order total is the sum of all product prizes multiplied by their quantity"), while prescriptive rules articulate how it should be ("the order total should exceed 50 dollars"). Within this project, all described rules are prescriptive: they govern the behavior and state of existing classes and processes.

At the implementation level, rules are often named assertions. An assertion is a boolean expression, connecting to some point(s) in the program, that should evaluate to true at its connected point(s) in the program on execution. It is meant to formalize and guard assumptions, thus ensuring safe and reliable behavior [30].

Considering the applicability of rules within the scope of the ngUML project, two views can be differentiated. By utilizing both views, the presented taxonomy should ensure clarity through a logically divisible delineation and expressiveness through a variety of specifications.

In [32], a business process is defined as "an activity that is managed by an organization to produce results of value to that organization, its customers, its suppliers, and/or its partners" (p. 15). The 'rational view' of the following taxonomy outlines a higher dimensional representation of rules that are directly applicable to entire business processes or substantial components. It relates to high-level concepts like variability. For example, within the context of a casino, "a customer must be at least 21 years old" and "a customer may not order extra chips" should be handled differently. The former must always be satisfied, while the latter is instance-dependent: its applicability is evaluated using contextual parameters like a customer's account balance.

The 'specified view' represents a lower dimension that borders the implementation of rules: it defines rules that restrict, for example, what values an attribute can take or whether these values should be ordered. The rational view contains super-types of the rules in the specified view, thus exclusively the lowest sub-types in the latter view are instantiable.

#### 3.1 Taxonomy of rules: rational view

Figure 5 presents rules through the rational viewpoint. The UML 'generalize' relation is used to specify rule sub-types.

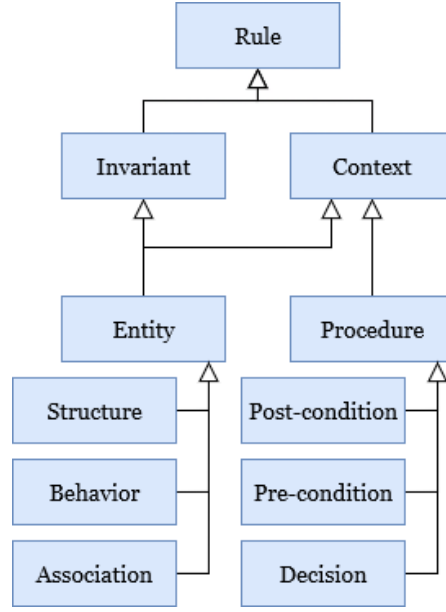


Figure 5: Taxonomy of rules: rational view

**Definition 1** (Rule). An assertion that must hold as specified in its definition.

**Definition 2** (Invariant rule). A rule that applies constantly.

An invariant rule must hold at all times. For example, “a customer should be at least 21 years of age.”

**Definition 3** (Context rule). A rule that only applies when its embedding context satisfies certain parameters.

A context rule applies conditionally in relation to its embedding context. Such a rule is formulated as “if  $X$ , then *rule*”; for example, “if a customer enters the casino, they should be at least 21 years of age.”

**Definition 4** (Entity rule). A rule that restricts the behavior or structure within an entity or the association between entities.

**Definition 5** (Structure rule). A rule that restricts what structural characteristics an entity can possess.

The aforementioned rule “a customer should be at least 21 years of age” is a structure rule that restricts the values that an entity’s property can take on. For a detailed outline on the variety of possible applications, see the rule taxonomy’s specified view in chapter 3.2.

**Definition 6** (Behavior rule). A rule that restricts which actions an entity can perform.

A behavior rule restricts what activities an entity can perform or participate in. For example, “only an administrator can alter the delivery schedule.”

**Definition 7** (Association rule). A rule that restricts the characteristics of relationships between entities.

Where the structure and behaviour rule restrict intra-entity characteristics, the association rule restricts on an inter-entity level. For example, a scrum team entity might consist of a scrum master entity, a business analyst entity, and multiple developer entities. An association rule would restrict the characteristics of such a ‘consist’ relationship (e.g., maximally one scrum master).

**Definition 8** (Procedure rule). A rule that takes sequentiality into account when considering its application.

Since sequentiality is evaluated relative to a point of reference (a trigger, another procedure, et cetera), it is always a specialization of a context rule.

**Definition 9** (Pre-condition rule). A rule that applies before a procedure is begun.

A pre-condition rule specifies the state that its embedding context should conform to before an activity commences. For example, “a user’s e-mail address should be confirmed before they can reset their password.”

**Definition 10** (Post-condition rule). A rule that applies after a procedure is completed.

Like a pre-condition rule, a post-condition rule specifies a state that its embedding context should conform to. In this case, the condition should hold after an activity is finished. For example, “after order checkout is completed, the payment should be received within two business days.”

**Definition 11** (Decision rule). A rule that restricts the set of subsequent activities.

A decision rule governs the possibility space when choosing subsequent activities; it controls the workflow. The fundamental difference between pre and post-condition rules is that a decision rule restricts the activity-space, rather than the state-space. E.g., “if the order total is above \$100,000, the sales officer should be notified.” So-called trigger rules or event rules fall within this definition.

### 3.2 Taxonomy of rules: specified view

The specified view outlines a lower-dimensional representation of rules: it further delineates rules to instantiable sub-types that border the technical implementation.

In figure 6, rules within a circle further specify the rational rule set shown underlined, meaning that they are the successors of the rules in this rule set. Like the rational taxonomical representation, the ‘generalize’ relation is used to specify rule sub-types. In the specified view, a division is made between invariant sub-types and context sub-types because they possess different successors. This division is indicated by the vertical line. The green circle presents the specified rule types with the rational structure, behavior, and association rules as predecessors, considering invariant sub-types exclusively. The blue and red circles show all specified rules for context sub-types. There is an overlap between the blue and red circle: the inclusive, exclusive, and order rules are successors from both rule set B and rule set C. Since the specified view further delineates the rational view, all rules exist in both representations. For example, the rule “a customer’s age should be at least 21” is



an invariant structure rule in the rational view and an inclusive demarcation rule specifying rule set A in the specified view.

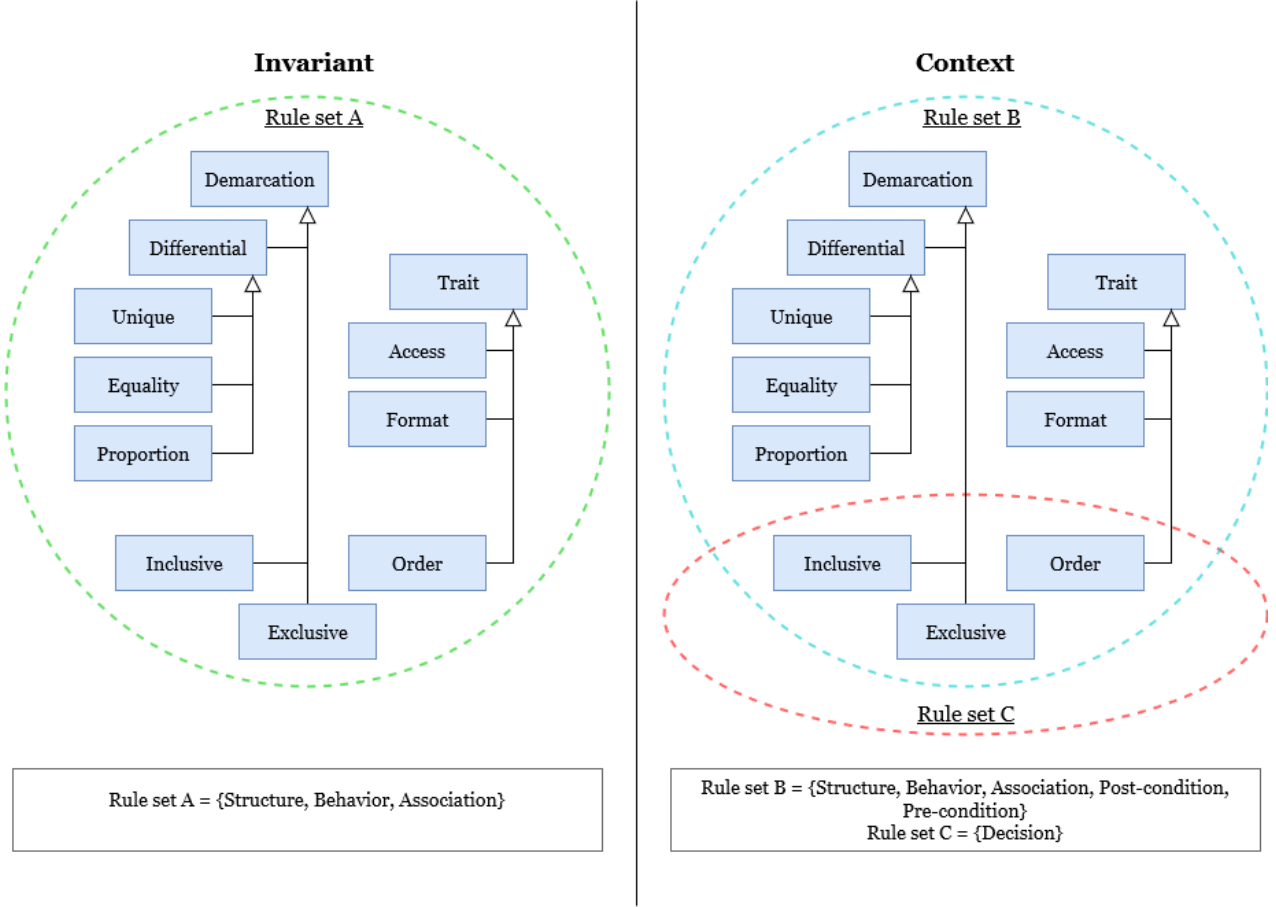


Figure 6: Taxonomy of rules: specified view

**Definition 12** (Demarcation rule). A rule that specifies what values can be assigned to an entity, set of entities, or to an association between entities.

Whether a demarcation rule sub-type specifies a structure rule or association rule depends on the characteristics of the entities that it restricts. It specifies the former if it restricts entities that are treated as properties within a single entity, it specifies the latter if it restricts otherwise independent entities that are connected through an association. For example, considering the rule “the name on a booking receipt should be the same as the name on the booker’s passport”, the characteristics of ‘booking receipt name’ and ‘booker’s passport name’ are the differentiator. It is a structure rule if they are treated as properties on a single booking entity, it is an association rule if they are treated as two separate entities that are related to a booking entity.

**Definition 13** (Differential rule). A rule that specifies what values can be assigned to an entity or set of entities in relation to another entity or set of entities.

The differential rule fundamentally differs from the inclusive and exclusive rule because it explicitly refers to another entity or set of entities when demarcating a set of legally assignable values.

**Definition 14** (Unique rule). A rule that specifies that the content of a data item or combination of data items must be unique in relation to the content of another data item or combination of data items.

For example, “the combination date of birth and username should be unique for every customer in relation to all other customers.”

**Definition 15** (Equality rule). A rule that specifies that the content of a data item or combination of data items must be equal or unequal to the content of another data item or combination of data items.

The aforementioned rule “the name on a booking receipt should be the same as the name on the booker’s passport” is an example of this

**Definition 16** (Proportion rule). A rule that specifies what numerical values can be assigned to an entity or set of entities in relation to another entity or set of entities.

For example, “the load of a truck should always be 50 pounds less than the capacity of the truck.”

**Definition 17** (Rule set A & B: Inclusive rule). A rule that specifies what set of values can be assigned.

The inclusive rule can implement a structure, behavior, association, post, or pre-condition rule, as well as a decision rule. The interpretation of this lower-level rule depends on the type of rational rule it implements. Within the context of rule set A and B, the inclusive rule specifies a required state: what values should be assigned to a corresponding entity or set of entities. An inclusive rule implementing a pre-condition is “a clerk should have at least five years of experience before they can be promoted to store manager.”

**Definition 18** (Rule set C: Inclusive rule). A rule that specifies what set of procedures can be executed.

With regard to rule set C, the inclusive rule restricts the state-space: what procedures can be executed considering the context. An inclusive rule implementing a decision rule is “after the customer selects order checkout, they can either opt for credit or debit card payment.”

**Definition 19** (Rule set A & B: Exclusive rule). A rule that specifies what set of values cannot be assigned.

Like the inclusive rule, an exclusive rule can implement a structure, behavior, association, post, or pre-condition rule, as well as a decision rule. The differentiating feature is that the exclusive rule defines what values or procedures do not belong to the possibility set.

**Definition 20** (Rule set C: Exclusive rule). A rule that specifies what set of procedures cannot be executed.

**Definition 21** (Trait rule). A rule that specifies what characteristics the values assigned to an entity, set of entities, or association between entities should have.

A trait rule specifies the characteristics of data items within a demarcated space of legal assignments. Like the demarcation rule, whether trait rule sub-types specify a structure or association rule depends on the autonomy of the restricted entities.

**Definition 22** (Access rule). A rule that specifies what types of access to an entity or set of entities are legal.

An access rule defines how the content of a data item or combination of data items can be accessed, for example, “only an administrator can alter the delivery schedule.”

**Definition 23** (Format rule). A rule that specifies how the content of a data item or a combination of data items must be formatted.

A format rule defines what format is required when handling, representing, or saving data. For example, “timestamps should be formatted as *MM/dd/yyyy HH:mm:ss.SSSS*.”

**Definition 24** (Rule set A & B: Order rule). A rule that specifies how a set of entities or the content within an entity or set of entities should be ordered.

When implementing rule set A and B, an order rule defines what order is required when handling, representing, or saving data. For example, “outgoing orders should be ordered descendingly on the due date.”

**Definition 25** (Rule set C: Order rule). A rule that specifies the sequential execution of procedures.

When implementing rule set C, an order rule defines the sequential traversal of the state-space. For example, “when stock is delivered, the products should be inspected before they are unloaded.”

## 4 System design

The goal of this project is to transform natural language input into a source code implementation of rules for enforcement in a run-time prototype application. In the following chapters, a software solution is presented that fulfills this goal. Under ‘Logical design’, a high-level overview of the implementation and its function within the greater ngUML project is outlined. Under ‘Technical design’, a low-level description of the implementation is presented.

### 4.1 Logical design

#### 4.1.1 System requirements

The software solution presented in this paper should realize the following requirements.

1. **The resulting implementation should be able to convert rules that are defined in a natural language into source code at run-time.** The goal of the ngUML project is to improve the accessibility of software development for non-IT domain experts, construction of models is therefore done using natural language input. Thus, the specification of rules governing such models should be possible using natural language as well. These submitted rules should result in source code that is applicable to prototypes at run-time.
2. **The resulting implementation should be able to implement expressive rules.** Business rules can be nuanced and complex. Thus, the software solution should show promise in handling such rules.
3. **The resulting implementation should ensure general user-friendly functionality for non-IT experts.** Accessibility is an essential target for this project. Thus, non-IT experts should experience the resulting implementation as easy to use.
4. **The resulting implementation should be extensible.** Many people have contributed to the ngUML project and many more will continue to do so. Therefore, the software solution that is delivered needs to be expandable to make it future-proof.

#### 4.1.2 System overview

The software solution presented in this paper transforms textual requirements into enforceable rules by converting constraints formulated in natural language to source code. An important pre-condition for this process is that the submitted rules apply to existing classes. As shown in figure 7, the workflow starts with a user submitting text or audio that is converted to metadata, which is saved in the metamodel. Only after this, rules can be formulated and applied. The natural language processing, formal rule generation, and rule object management are bundled in a single component that is named ‘Rule Manager’. The Rule Manager is implemented to handle a subset of the rules taxonomy, namely the invariant inclusive demarcation rule that specifies rule set A. In order to enable this functionality, a generic architecture is implemented that can be extended to handle many more rule types. Figure 7 depicts its position within the complete ngUML software’s workflow.

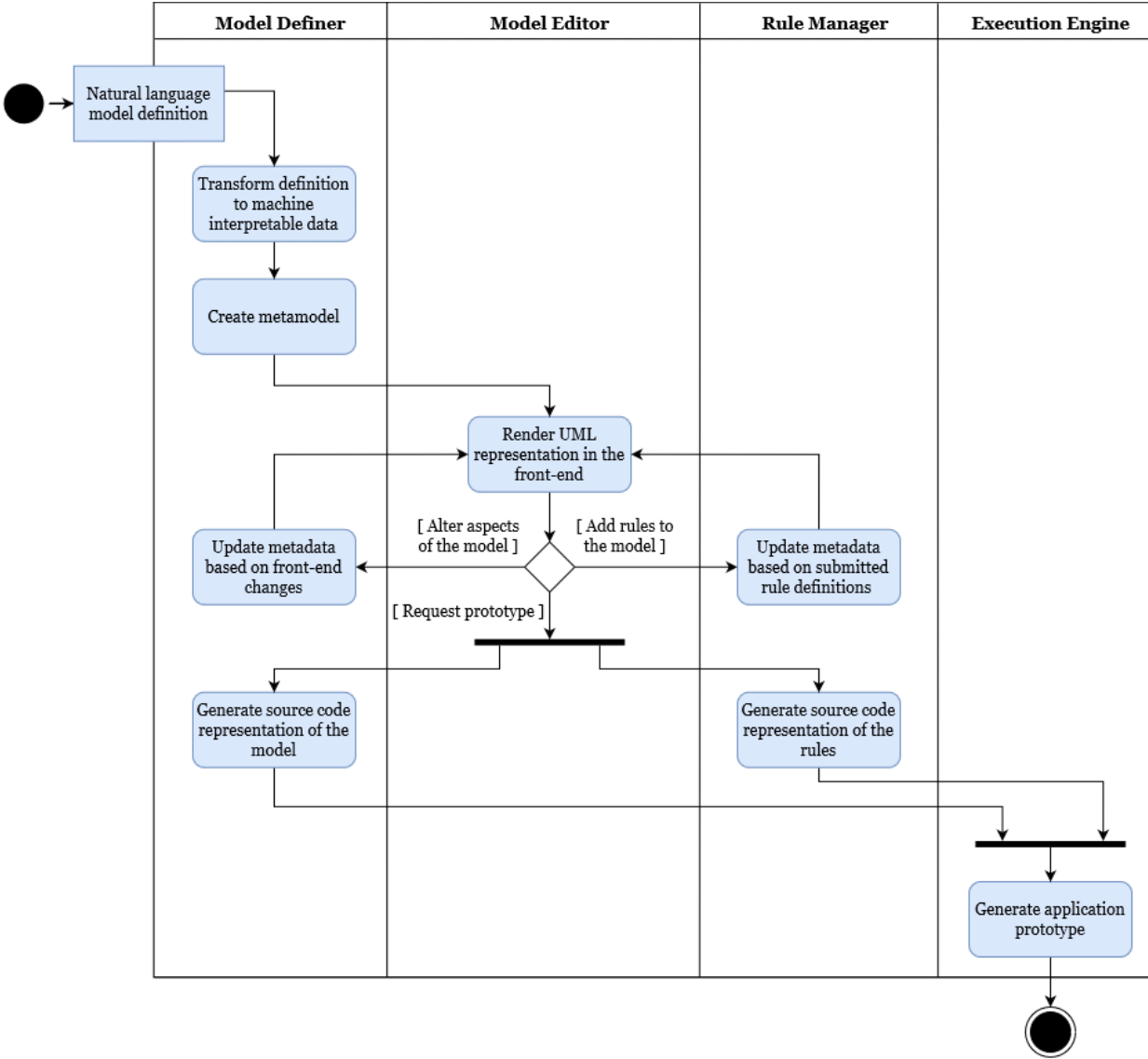


Figure 7: Activity diagram of the ngUML software's workflow

The diagram in figure 8 shows the relevant parts of the Rule Manager. The implementation is divided into three subsystems, each feeding data to the next one. The Rule Manager interacts with the existing ngUML code base at specific points. It fetches user input, metadata artifacts, and rule alterations in the front-end; it exports the structured rule strings, validator code, and a user view representation of rules.

The Rule Manager essentially forms an extra pipeline within the ngUML engine. It works end-to-end: taking in user input, altering the metamodel, and applying the resulting source code to the prototype (see figure 9). The NLP component requires an interface to the user where it can fetch input. After this, the input gets tokenized, lemmatized and parts of speech are tagged. The Formal Rule Generator maps metadata to the input object and validates if the result is correct and interpretable. If so, the object is articulated in a structured rule language, which is designed to act as an intermediary between the expressive capacity of natural language and the unambiguous

nature of programming languages (see 4.2.3). Finally, the Rule Object Manager transforms this structured rule string into Python code and enforces it on the metamodel.

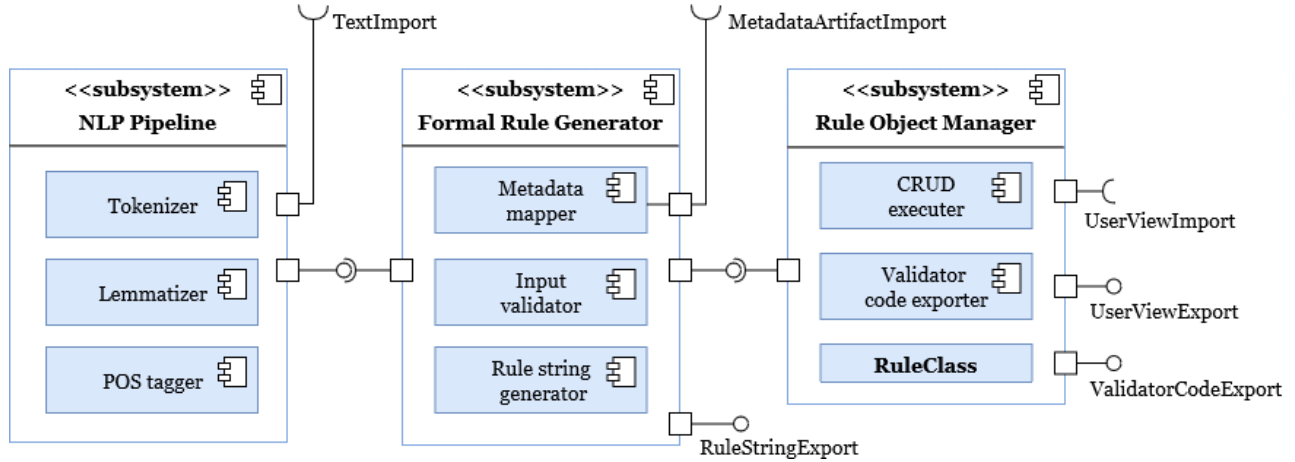


Figure 8: Component diagram of the Rule Manager

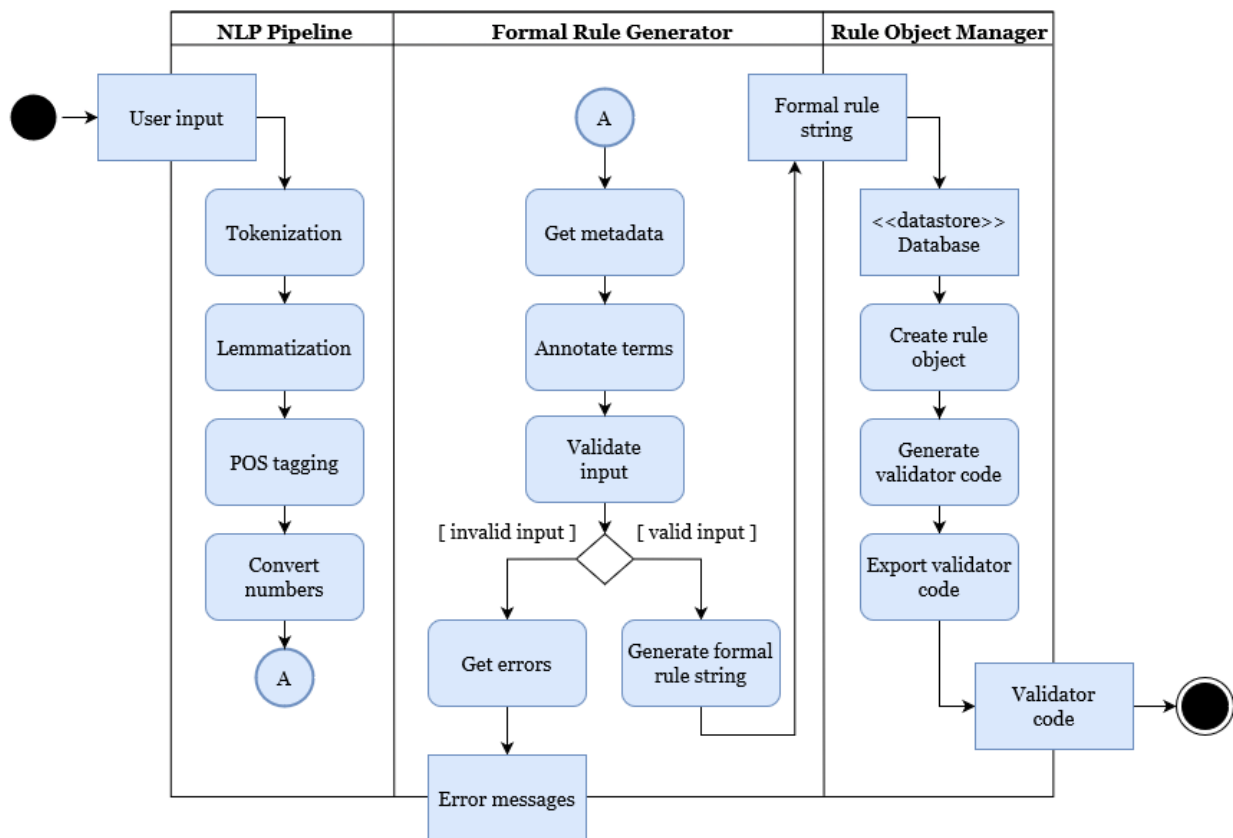


Figure 9: Activity diagram of the Rule Manager's workflow

## 4.2 Technical design

This section expands on the previously outlined conceptual view of the software solution by presenting technical implementation details. Three types of validator implementation are outlined and subsequently mapped to the rule taxonomy. Then, after defining the structured rule language for a subset of the rule taxonomy, some examples are showcased.

### 4.2.1 Validator implementation methods

At the source code level, a rule is an assertion that governs the properties and methods of class instances by throwing an error if it evaluates to false. The exact implementation of an assertion depends on several factors and is generalizable to three levels: atomic, intra-class, and inter-class. Since the back-end of the ngUML application is built using the web-framework Django, the following chapters outline the different validator implementations within the context of a Django application. However, the generic nature of this proposed division should enable usage at a conceptual level, independent of the implementation context.

Django is a python-based web-framework. As stated in [11], the Django philosophy is efficient development. It subscribes to the ‘Don’t repeat yourself’ (DRY) principle which postulates that “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system” ([21]). The tech community appears to appreciate its efficacy: it is used at Instagram [10], Mozilla [26], and Spotify [25].

Within Django, an object relational mapper mediates between data models and a relational database. All models are defined using Python classes (see listing 1). Django performs some validation out-of-the-box when an instance is saved to the database and offers several access points for customizing this process [14]. Within Django, assertions are called ‘validators’: functions that take a value and raise a `ValidationError` object if some criteria are not met [17]. Django performs validation in a three-step process that is coordinated by `full_clean()` [14]:

1. Atomic validation using `Model.clean_fields()`;
2. Intra-model validation using `Model.clean()`;
3. Field uniqueness validation using `Model.validate_unique()`.

The `full_clean()` method is not called by default when saving an instance. Thus, to enable the execution of custom validator functions, it needs to be added to the `save()` method of a class for nearly all use-cases in the three aforementioned levels of evaluation [14]. Within the `full_clean()` method, exceptions can be gathered to display in the front-end.

**Atomic validation** At the atomic level, an individual instance’s properties are evaluated. A basic level of such functionality executes by default. When an instance of the `Customer` object (as shown in listing 1) is saved, `MinValueValidator` and `MaxValueValidator` are implicitly called on the `age` property to enforce that the entered integer value falls within a domain that is safe to enter into the underlying database [13]. Custom rules can be enforced by adding validators to class properties. Listing 1 shows that the `min_balance` validator is linked to the `account_balance` property. When a `Customer` instance is saved to the database with an account balance value of less than \$5, a `ValidationError` is raised. In this way, many atomic rules are enforceable.

```

1 def min_balance(value):
2     if value < 5:
3         raise ValidationError(
4             _("Your account balance should be at least $5.")
5         )
6
7 class Customer(models.Model):
8     username = models.CharField(max_length=100)
9     account_balance = models.FloatField(validators=[min_balance])
10    age = models.IntegerField()
11    ...
12    def save(self, *args, **kwargs):
13        try:
14            self.full_clean()
15        except ValidationError as e:
16            ...
17        super().save(*args, **kwargs)

```

Listing 1: Atomic evaluation

**Intra-class validation** At the intra-class level, the states of multiple properties within a single class are evaluated to enforce constraints. To enable this type of custom validation, functionality needs to be added to a model’s corresponding `clean()` method. In listing 2, the rule “only a customer with a premium subscription may opt for drone delivery” is enforced.

```

1 class Customer(models.Model):
2     username = models.CharField(max_length=100)
3     subscription = models.CharField(max_length=100)
4     default_delivery = models.CharField(max_length=100)
5     ...
6     def clean(self):
7         if self.subscription != "premium" and self.default_delivery == "drone":
8             raise ValidationError(
9                 _("Drone delivery is only possible with a \
10                    premium subscription.")
11             )
12     ...
13    def save(self, *args, **kwargs):
14        try:
15            self.full_clean()
16        except ValidationError as e:
17            ...
18        super().save(*args, **kwargs)

```

Listing 2: Intra-class level evaluation

**Inter-class validation** At the inter-class level, the state of multiple classes – and possibly their properties – are evaluated to enforce constraints. Models can be related through a One-to-One, Many-to-One, or Many-to-Many relationship. Models can be directly connected, indirectly connected, or not connected at all. Directly connected means that two models are connected because either model has a foreign key relation to the other. Indirectly connected means that two models are connected through one or more intermediate models. Listing 3 shows a directly connected



Many-to-One relationship that enforces the rule “only customers with a premium subscription may opt for evening delivery”. This is again enabled by customizing the model’s `save()` function. The same method works when constraining a One-to-One relationship.

```
1 class Customer(models.Model):
2     username = models.CharField(max_length=100)
3     subscription = models.CharField(max_length=100)
4     default_delivery = models.CharField(max_length=100)
5
6
7 class Order(models.Model):
8     delivery_slot = models.CharField(max_length=100)
9     customer = models.ForeignKey(
10         "Customer", on_delete=models.CASCADE, null=True
11     )
12     ...
13     def clean(self):
14         if self.delivery_slot == "evening" and
15             self.customer.subscription != "premium":
16             raise ValidationError(
17                 _("Evening delivery is only possible with a \
18                 premium subscription.")
19             )
20     ...
21     def save(self, *args, **kwargs):
22         try:
23             self.full_clean()
24         except ValidationError as e:
25             ...
26         super().save(*args, **kwargs)
```

Listing 3: Inter-class Many-to-One evaluation

In the case of a Many-to-Many relationship, both instances have to be saved before they can be linked together using `add()`. However, `add()` does not call `save()` out of performance considerations [15]. Django does provide a hook to customize the process of saving Many-to-Many instances to the database through the `m2m_changed` signal (listing 4, line 23). This signal can be utilized to monitor changes concerning such relationships because it is called before and after saving any changes [16]. Since `m2m_changed` is called for every Many-to-Many alteration, a redirect function should check the involved instance-types and call their corresponding validation functions (line 1).

```
1 def m2m_redirect(sender, instance, action, model, **kwargs):
2     # Mentor added to Mentee
3     if model is Mentor and isinstance(instance, Mentee):
4         pass
5     # Mentee added to Mentor
6     if model is Mentee and isinstance(instance, Mentor):
7         Mentor_m2m_Mentee(instance)
8
9 def Mentor_m2m_Mentee(instance):
10     if instance.mentee_set.all() > 4:
11         raise ValidationError(
12             _("A mentor may have maximally 4 mentees.")
13         )
```

```

14
15 class Mentor(models.Model):
16     ...
17
18 class Mentee(models.Model):
19     ...
20     mentor = models.ManyToManyField(Mentor)
21
22
23 m2m_changed.connect(m2m_redirect)

```

Listing 4: Inter-class Many-to-Many evaluation

The aforementioned examples regarding the different levels of implementation represent typical applications. The atomic and intra-class examples could easily be transformed into an inter-class implementation if the database were queried for other metadata when enforcing a rule. In listing 5, the cash coverage ratio is calculated before a new customer instance is added, transforming the atomic validation from listing 1 to inter-class validation. In this way, rules can span many classes that are directly linked, indirectly linked, or not linked at all.

```

1 def min_coverage(value):
2     total_deposit = value
3     total_cash = request_cash_reserve()
4     res = Customer.objects.all()
5     for i in range(len(res)):
6         total_deposit += res[i].account_balance
7     if not (total_cash / total_deposit * 100 >= 80):
8         raise ValidationError(
9             _("Required cash coverage ratio is breached."))
10
11
12 class Customer(models.Model):
13     username = models.CharField(max_length=100)
14     account_balance = models.FloatField(validators=[min_coverage])
15     age = models.IntegerField()

```

Listing 5: Atomic evaluation transformed to inter-class evaluation

#### 4.2.2 Mapping rules to implementation

This section presents a mapping between the rules taxonomy and the levels of implementation. The charting of the rule domain, together with the following mapping, is meant to contribute to effective discourse on the implementation and assessment of the ngUML project’s rule handling capabilities.

There are many ways to enforce rules in Django because of the framework’s flexibility. While this is indeed useful, the following assertions are valuable because they enforce a pragmatic and qualitative angle on the implementation.

1. The input of a model’s property field is handled as a single value.
2. If an invariant rule can be stated as a context rule, this should be done.

Assertion 1 is not necessarily the case, because Django offers the possibility of creating custom property fields. For example, one could construct a ‘MultiEmailField’ that takes a comma-separated list of e-mails that can be parsed and treated as having multiple of values [12]. This could be of value in edge-cases, but it also enables mapping to impractical implementations of rules. For example, one could construct a validation field that implements an atomic context access rule. Taking input in the form *password:value*, the value would only be alterable when the entered password is correct. While this is a workable solution, it needlessly pollutes the input channel. Since assertion 1 enforces a pragmatic angle on these obscure edge-cases and keeps the mapping clear, it is worth the restriction.

Assertion 2 is valuable because it protects implementation quality by excluding the application of context rules as hard-coded invariant rules. For example, one could enforce the rule “only an administrator can book a meeting” in two ways. In the form of a context rule, it would first check the trigger if `meeting_schedule.altered == True` and then check the contextual parameter if `user.role != administrator`. When implemented as an invariant, it would be implemented as `if not (user.role != administrator) and meeting_schedule.altered == True`. The latter implementation needlessly combines the validation-trigger and the contextual requirement in a single enforcing statement, effectively hard-coding context into the rule. Ensuring this separation is crucial when the number of contextual parameters increases to the level of real-world business applications.

Figure 10 and 11 present the mapping of all rules in the rule taxonomy’s specified view to Django implementation types. Figure 10 shows the rules as invariant sub-types: meaning that the inclusive, exclusive and order rule can only implement rule set A (see figure 6). Figure 11 shows the rules as context rule sub-types, the ‘rule set B’ and ‘rule set C’ rows signify what rational rule super-types correspond to the subsequent inclusive, exclusive, and order rule rows.

Invariant			
Rule	Implementation type		
	Atomic	Intra-class	Inter-class
Unique		X	X
Equality		X	X
Proportion		X	X
Access	X		X
Format	X	X	X
Rule set A			
Inclusive	X		X
Exclusive	X		X
Order		X	X

Figure 10: Mapping of invariant rule types to implementation levels

Context			
Rule	Implementation type		
	Atomic	Intra-class	Inter-class
Unique		X	X
Equality		X	X
Proportion		X	X
Access		X	X
Format		X	X
Rule set B			
Inclusive		X	X
Exclusive		X	X
Order		X	X
Rule set C			
Inclusive			X
Exclusive			X
Order			X

Figure 11: Mapping of context rule types to implementation levels

#### 4.2.3 EBNF definition of the inclusive demarcation rule

In this section, the structured syntax of the invariant inclusive demarcation rule that specifies rule set A (i.e., A\_DEM\_INCL) is defined using Extended Backus-Naur Form. The definition is specified according to the ISO/IEC 14977 EBNF standard ([22]), and partitioned in components with a short explanation. When applying this rule type within an inter-class context, it constrains the multiplicity of relationships. Since One-to-One relationships fix their multiplicity at a database level, the use-case for this relationship type is excluded from the A\_DEM\_INCL syntax definition. The following operators are used:

- `=` splits the left hand side and the right hand side. The term on the LHS is a meta-identifier, this term can be substituted by any terminal or non-terminal symbol on the RHS. A symbol is a sequence of one or more characters.
- `|` indicates a choice between symbols, the meta-identifier on the LHS can be substituted by any of the choices on the RHS.
- `"` and `'` are used to enclose terminals.
- `[` and `]` indicate option, meaning that the enclosed symbols can be omitted or used once.
- `{` and `}` indicate repetition, meaning that the enclosed symbols can be omitted or repeated any number of times.
- `,` is used to concatenate symbols.
- `;` is used to terminate a sequence of symbols.

```

1 rule string =
2     "'", rule name , "---" , context information , "---" , constraints , "'" ;
3
4 rule name = "A_DEM_INCL" ;
5 context information = property information | direct relation information ;
6 """a + b = y + z"""
7 constraints =
8     a * [ "(" ] , block , y * [ ")" ] ,
9     { logical operator , b * [ "(" ] , block , z * [ ")" ] } ;

```

Listing 6: Rule definition 1/5

A rule string consists of three parts: the rule name tag, context information, and constraints. The invariant inclusive demarcation rule that implements rule set A is tagged with A\_DEM\_INCL. ‘Rule name’ is not fixed at A\_DEM\_INCL because this definition may be reusable for other rules, like the invariant exclusive demarcation rule implementing rule set A. The context information firstly defines if the rule is atomic or inter-class and secondly provides its application context. If a rule is atomic, the corresponding classifier and property name are passed. If the rule is inter-class, a query term and possibly a foreign key alias are passed (see listing 7). Constraints define the actual rule and are split into blocks. Brackets can enclose blocks to explicitly state the precedence of logical operators, it is required that the number of opening brackets equals the number of closing brackets (line 6). Since the logical operators get directly translated to Python (see listing 12, line 5), their precedence is interpreted as defined in the documentation [18].

```

1 direct relation information =
2     "rel." , many to one relation | many to many relation ;
3 many to one relation =
4     possessor , ".m2o." , possession , "." , foreign key alias , query term ;
5 many to many relation =
6     possessor , ".m2m." , possession , "." , query term ;
7 foreign key alias = identifier ;
8 query term = identifier ;
9
10 property information = "field." , class name , "." , property name ;
11 class name = identifier ;
12 property name = identifier ;

```

Listing 7: Rule definition 2/5

Direct relation information can either be of the type Many-to-One or Many-to-Many. A Many-to-One relationship is defined by assigning a foreign key value to a model’s property (see listing 3, line 9), the name of this property should therefore be passed as a foreign key alias. A query term is used to retrieve all related instances. Atomic rules are enforced on the properties of classes, therefore the assignment of a class name and a property name is required.

```

1 block = "|" , ( direct relation block | property block ) , "|" ;
2
3 property block =
4     type constraint , "=" |
5     "=", ( digit range constraint | letter range constraint ) |
6     type constraint , "=" , digit range constraint |
7     "*int=" , digit range constraint | "*let=" , letter range constraint ;
8

```

```
9 direct relation block = "=" , digit range constraint ;
```

Listing 8: Rule definition 3/5

A property block contains a `=` sign that separates the type constraint on the LHS from the digit range constraint on the RHS. A property block can be defined by passing both a LHS and RHS, constraining both the type and multiplicity of the target. Omitting one of the two sides is a legal formulation as well, resulting in the enforcement of the constraint on the side that's defined. One can only pass a digit range constraint in a direct relation block, since the type is already fixed at the related class. For example, when enforcing such a constraint on a Mentor-Mentee relation, only the multiplicity can be determined because the type is fixed at Mentor or Mentee. Instead of a type constraint, `*int` or `*let` can be passed. The former defines the actual integer value that a property should have, the latter defines the letter value (e.g., `*let="admin"`).

```
1  """a + b = y + z"""
2  type constraint =
3      a * [ "(" ] , type , y * [ ")" ] ,
4      { logical operator - "and" , b * [ "(" ] , type , z * [ ")" ] } ;
5
6  digit range constraint =
7      a * [ "(" ] , interval , y * [ ")" ] ,
8      { logical operator , b * [ "(" ] , interval , z * [ ")" ] } ;
9
10 letter range constraint = a * [ "(" ] , string , y * [ ")" ] ,
11    { logical operator , b * [ "(" ] , string , z * [ ")" ] } ;
```

Listing 9: Rule definition 4/5

The type, digit range, and letter range constraint follow the same structure as the meta-identifier constraints defined in listing 6, the differences being the terms that are enclosed by brackets and the number of legal logical operators. Currently, a type constraint is used to enforce that a class property is (a combination of) the type 'integer' or 'letter'. Since any single value cannot be both at once, the `and` operator is excluded from usage.

```
1  string = "" , sequence , "" ;
2  sequence = letter , { letter } ;
3  interval =
4      digit | lower bound , digit , "," | "," , digit , upper bound |
5      lower bound , digit , "," , digit , upper bound ;
6
7  identifier = letter , { letter | digit | "_" } ;
8  lower bound = "{" | "["
9  upper bound = "}" | "]"
10 type = "int" | "let"
11 logical operator = "or" | "and" | "xor"
12 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
13 letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
14         | "H" | "I" | "J" | "K" | "L" | "M" | "N"
15         | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
16         | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
17         | "c" | "d" | "e" | "f" | "g" | "h" | "i"
18         | "j" | "k" | "l" | "m" | "n" | "o" | "p"
19         | "q" | "r" | "s" | "t" | "u" | "v" | "w"
```

Listing 10: Rule definition 5/5

An interval can be defined as having a single bound, a lower and upper bound, or a constant value. Regarding the upper and lower bound: ‘{’ defines an exclusive bound and ‘[’ defines an inclusive bound.

#### 4.2.4 From natural language to structured rules

The rules in this project are constructed from natural language input. To enable a lexical interpretation of this input, an NLP pipeline is implemented. Some essential functionalities are realized with Stanza, an open-source Python NLP toolkit, developed by Qi et al. [3]. All functionality is based on neural network methodology, providing competitive results: it generally outperforms the widely used NLP toolkits UDPipe and spaCy (see figure 12).

Treebank	System	Tokens	Sents.	Words	UPOS	XPOS	UFeats	Lemmas	UAS	LAS
Overall (100 treebanks)	Stanza	99.09	86.05	98.63	92.49	91.80	89.93	92.78	80.45	75.68
Arabic-PADT	Stanza	<b>99.98</b>	80.43	<b>97.88</b>	<b>94.89</b>	<b>91.75</b>	<b>91.86</b>	<b>93.27</b>	<b>83.27</b>	<b>79.33</b>
	UDPipe	<b>99.98</b>	<b>82.09</b>	94.58	90.36	84.00	84.16	88.46	72.67	68.14
Chinese-GSD	Stanza	<b>92.83</b>	98.80	<b>92.83</b>	<b>89.12</b>	<b>88.93</b>	<b>92.11</b>	<b>92.83</b>	<b>72.88</b>	<b>69.82</b>
	UDPipe	90.27	<b>99.10</b>	90.27	84.13	84.04	89.05	90.26	61.60	57.81
English-EWT	Stanza	<b>99.01</b>	<b>81.13</b>	<b>99.01</b>	<b>95.40</b>	<b>95.12</b>	<b>96.11</b>	<b>97.21</b>	<b>86.22</b>	<b>83.59</b>
	UDPipe	98.90	77.40	98.90	93.26	92.75	94.23	95.45	80.22	77.03
	spaCy	97.30	61.19	97.30	86.72	90.83	–	87.05	–	–
French-GSD	Stanza	<b>99.68</b>	<b>94.92</b>	<b>99.48</b>	<b>97.30</b>	–	<b>96.72</b>	<b>97.64</b>	<b>91.38</b>	<b>89.05</b>
	UDPipe	<b>99.68</b>	93.59	98.81	95.85	–	95.55	96.61	87.14	84.26
	spaCy	98.34	77.30	94.15	86.82	–	–	87.29	67.46	60.60
Spanish-AnCora	Stanza	<b>99.98</b>	<b>99.07</b>	<b>99.98</b>	<b>98.78</b>	<b>98.67</b>	<b>98.59</b>	<b>99.19</b>	<b>92.21</b>	<b>90.01</b>
	UDPipe	99.97	98.32	99.95	98.32	98.13	98.13	98.48	88.22	85.10
	spaCy	99.47	97.59	98.95	94.04	–	–	79.63	86.63	84.13

Figure 12: Performance comparison on the Universal Dependencies test treebanks. Source: [3]

In figure 9, an overview of the rule generation process is given. Tokenization, lemmatization, and part-of-speech (POS) tagging is performed by the Stanza toolkit. The NLP Pipeline produces a Python list with tagged tokens for every input-sentence. The tag-set consists of:

- **log\_op**, mapped to tokens that specify a logical operator like ‘and’.
- **val\_op**, mapped to tokens that specify relationships between values like ‘less’ or ‘most’.
- **terminal**, mapped to tokens that specify a value like ‘digit’ or ‘letter’.
- **num**, mapped to numbers like ‘20’ or ‘three hundred’. Stanza’s POS tagging is used for this.
- **lit**, mapped to words that are enclosed by double-quotes.

JSON objects are used for the tagging of `log_op`, `val` and `terminal`. These JSON objects map tokens to their corresponding tag.

The list that is generated by the NLP Pipeline is passed to the Formal Rule Generator, which gathers all existing metadata and checks for every noun in the input-string if it matches classes or properties. Here, two more tags are added:

- **class**, mapped to words that correspond to an existing class name.
- **prop**, mapped to words that correspond to an existing property name.

This list, together with other input-related data, is bundled in a data structure and finally evaluated for correctness. If all tests are passed, the structured rule string is generated. A simple overview of the input conversion is shown in figure 13.

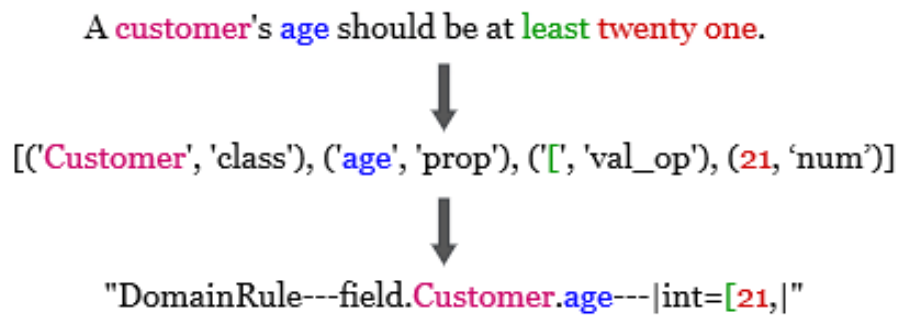


Figure 13: Conversion of natural language input to a structured rule

#### 4.2.5 From structured rules to code

Within this project, the invariant inclusive demarcation rule that specifies rule set A is implemented. As an invariant, the rule should always hold, so no precursory conditions are implemented. At the atomic level, the inclusive demarcation rule defines what values are legal for the property of a class. Possible examples are: “a customer should be at least 21 years old” or “a username should consist of at least five characters”. At the inter-class level, the rule defines requirements regarding the relation of entities. For example, in a class diagram with a Mentor and Mentee class, rules like “a mentor has at least five mentees” could be enforced. Such rules implement a refined multiplicity requirement on the Mentee and Mentor relation that was originally on a simple Many-to-One basis.

**Atomic** If a natural language sentence contains one classifier and one property, it is treated as atomic. The following are examples of A\_DEM\_INCL rule strings.

1. ‘A\_DEM\_INCL- - -field.Customer.account\_balance- - -|int=[5,500]|’
2. ‘A\_DEM\_INCL- - -field.Employee.department- - -|\*let="sales" or "IT" or "management"|’
3. ‘A\_DEM\_INCL- - -field.Employee.department- - -|\*let="admin"| or (|let or int={5,| and |=100|)’



Taking rule 2 as an example, `A_DEM_INCL` defines the rule type and `field.Employee.department` defines that its an atomic rule that applies to the class `Employee` and property `department`. The third part describes the actual rule that is to be implemented: the value of the property should either be “sales”, “IT” or “management”.

```

1 def A_DEM_INCL_Customer_account_balance(value):
2     results = [domain_Customer_account_balance_b1(value)]
3     if not (results[0][0]):
4         errors = []
5         for result in results:
6             if result[0] == 0:
7                 errors.append(result[1])
8             raise ValidationError(errors)
9
10 def A_DEM_INCL_Customer_account_balance_b1(value):
11     errors = []
12     for symbol in value:
13         if not(check_symbol_type(symbol, "int")):
14             errors.append(
15                 ValidationError(_("%(value) err_msg"),
16                                 params={"value": value},)
17             )
18             break
19     if not(5 <= len(value) <= 500):
20         errors.append(
21             ValidationError(_("%(value) err_msg"),
22                             params={"value": value},)
23         )
24     if len(errors) > 0:
25         return[0, errors]
26     return [1]

```

Listing 11: Atomic string example 1 to code

Listing 11 shows the validator source code as converted from the string in example 1. Such validators are constructed in a predefined structure: a coordinating parent validator and sub-validators. The first function retrieves all results (line 2) and combines the evaluations of all sub-validators to determine the final result (line 3). Finally, it gathers all errors and sends them to the front-end. The second function is a sub-validator. It first checks if all symbols are of the type integer (line 12-13), and secondly if the input length falls within the specified bounds. Any errors are aggregated and sent to the parent.

```

1 def A_DEM_INCL_Employee_department(value):
2     results = [A_DEM_INCL_Employee_department_b1(value),
3               A_DEM_INCL_Employee_department_b2(value),
4               A_DEM_INCL_Employee_department_b3(value)]
5     if not (results[0][0] or (results[1][0] and results[2][0])):
6         errors = []
7         for result in results:
8             if result[0] == 0:
9                 errors.append(result[1])
10            raise ValidationError(errors)
11
12 def A_DEM_INCL_Employee_department_b1(value):

```

```

13     ...
14     if not(check_input_type(value, "let")):
15         ...
16     if not(value == "admin"):
17         ...
18     if len(errors) > 0:
19         return[0, errors]
20     return [1]
21
22 def A_DEM_INCL_Employee_department_b2(value):
23     ...
24     for symbol in value:
25         if not(check_symbol_type(symbol, "let") or
26               check_symbol_type(symbol, "int")):
27             ...
28     if not(len(value) > 5):
29         ...
30     if len(errors) > 0:
31         return[0, errors]
32     return [1]
33
34 def A_DEM_INCL_Employee_department_b3(value):
35     ...
36     if not(len(value) == 100):
37         ...
38     if len(errors) > 0:
39         return[0, errors]
40     return [1]

```

Listing 12: Atomic string example 3 to code

Listing 12 shows the source code that corresponds to the string in example 3. Line 5 gives a clear example of how the parent validator combines the evaluation of all three sub-validators. In the first block of the rule-string, the `unsign` symbol is used (`*let`) to indicate that the whole input should consist of letters and that the value should equal ‘admin’. In sub-validator two, the `let` or `int` type constraint is enforced by iterating over every symbol. The final sub-validator checks exclusively if the length of the input-value equals one hundred. While this rule string is semantically not very useful, it shows the flexibility of the source code generator.

**Direct inter-class** If a natural language sentence contains two classifiers and zero properties, it is treated as direct inter-class. The following are examples of such inclusive demarcation rule strings.

1. ‘A\_DEM\_INCL- - -rel.Mentor.m2o.Mentee.mentor.mentee\_set- - -|={1,5}|’
2. ‘A\_DEM\_INCL- - -rel.Employee.m2m.PriorityTask.prioritytask\_set- - -|=2|’
3. ‘A\_DEM\_INCL- - -rel.PriorityTask.m2m.Employee.employees- - -|={10,}|’

Example 1 shows a rule string that enforces the rule “a mentor should have more than one and maximally five mentees” on a Many-to-One relationship. When splitting the context information at each dot, the second and fourth term specify the possessor and possession. This is essential

for determining the direction of a rule in Many-to-Many relationships. The fifth term passes the name of the property on the Mentee class that holds the foreign key reference to Mentor. The last term specifies the term that should be used to query all referenced Mentees on a Mentor instance. Examples 2 and 3 implement a constraint on the Many-to-Many relationship between Employee and PriorityTask. The former string regards Employee as the possessor and PriorityTask as the possession, implementing the rule “an employee should always be assigned two high priority tasks.” The latter string constrains the relation from PriorityTask to Employee. The last term is again the query term, its content depends on which class holds the foreign key reference.

```

1 def A_DEM_INCL_Mentor_m2o_Mentee(instance):
2     results = [A_DEM_INCL_Mentor_m2o_Mentee_b1(instance)]
3     if not (results[0][0]):
4         errors = []
5         for result in results:
6             if result[0] == 0:
7                 errors.append(result[1])
8             raise ValidationError(errors)
9
10 def A_DEM_INCL_Mentor_m2o_Mentee_b1(instance):
11     errors = []
12     if not(1 < len(instance.mentor.mentee_set.all()) <= 5):
13         errors.append(
14             ValidationError(_("%(instance) err_msg"),
15                             params={"instance": instance},)
16         )
17     if len(errors) > 0:
18         return[0, errors]
19     return [1]

```

Listing 13: Inter-class string example 1 to code

Listing 13 shows the implementation of rule string 1. The implementation is nearly identical to the atomic implementation, except for the evaluation (line 12). This similarity also holds for rule strings 2 and 3. The main difference between atomic and inter-class implementation is the method whereby the validator functions are linked to their corresponding model, as outlined in section 4.2.1.

## 5 Worked example

In this chapter, a worked example is outlined according to the user workflow in figure 7. While the creation of the model and the prototype application form important steps in this workflow, the incorporation of rules will be the point of focus. Figure 14 shows the creation of an information system for a university.

The screenshot shows the 'ngUML Requirements preprocessing' application. The top navigation bar includes 'Home', 'Manage requirements', and 'Runnable prototype'. On the left, a sidebar titled 'Generating a new model' lists three steps: 'Choose a project' (Step 1, selected), 'Create a new system' (Step 2), and 'Review extraction' (Step 3). The main content area is titled 'Create a new project' and contains the following sections:

- Project name:** A text input field containing 'University data store'.
- Project description:** A text area containing 'This application will store essential instance information and business rules about our university.'
- Write requirements:** A text area containing a sample requirement: 'A student has a username and GPA. Business Courses and Science Courses are types of courses. Each course consists of multiple lectures, and have a course name, code and date. A course coordinator organizes the courses. Each course has one or more lecturers, a location, time slot, and set of dates. Lecturers will give lectures, and administrators will make announcements that are for a particular course. Each course will have assignments and an exam. Students must attend to the lectures, complete the assignments and take the exam. Lecturers will give grades. A course grade consists of an assignment grade and exam grade. An exam is either a first exam or a retake. Students will receive their course grades by email.'
- Upload requirements:** A section with the text 'Max file size is 500kb. Only .txt or .wav files are supported.' and a dashed box for file upload. Below this is an 'OR' button.

Figure 14: Creating a project

In figure 15, the class diagram showcases a simple model of such an information system. Because Many-to-Many relationships cannot be implemented as source code yet, it is not possible to enforce Many-to-Many rules on metadata. Therefore, both relationships are defined as Many-to-One. A user can submit business rules in the rules menu on the right side. Successfully implemented rules are shown underneath the text-box, where they can be edited and deleted as desired. Taking figure 15 as an example, a variety of inclusive demarcation rules are currently implementable:

1. Rules that restrict the relation multiplicity of directly related classes. For example, “A lecturer should give at least four lectures”.
2. Rules that restrict the integer value of a property. For example, “The attendance of a lecture can maximally be one hundred and five”.
3. Rules that restrict the letter value of a property. For example, “A lecturer’s faculty can be “Economics” or “Science””.
4. Rules that restrict the number of symbols for a property. For example, “The username of an administrator should be at least six characters long”.

5. Rules that restrict the type of symbols for a property. The example at item 4 is applicable here. The possible types are: letters, characters and integers.

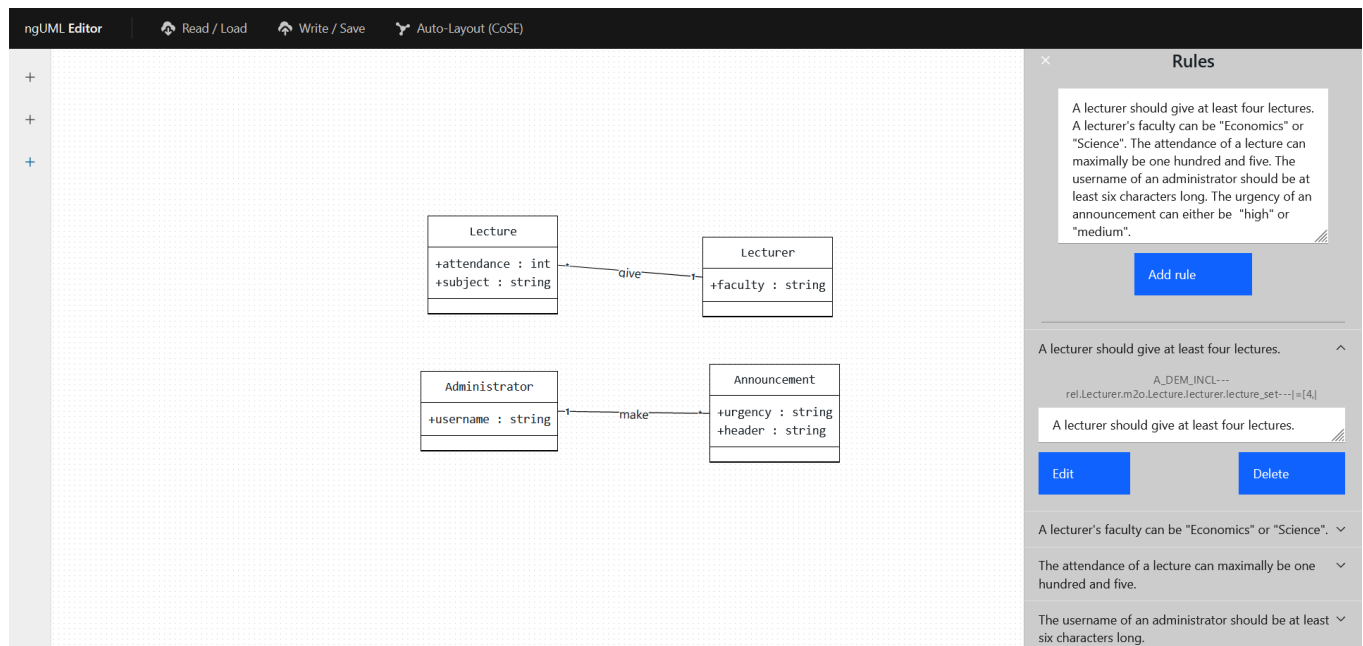


Figure 15: Implementing rules in a model

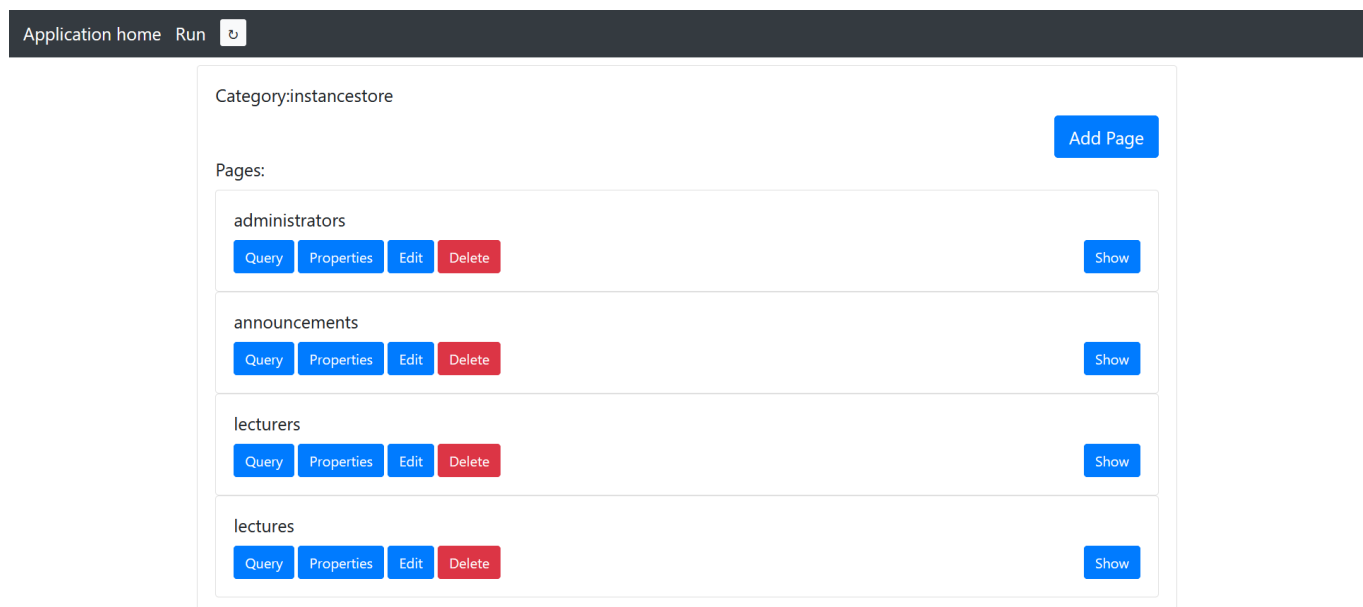


Figure 16: Creating an application prototype

In the current prototype generator, one can construct pages for creating class instances. Figure 16 shows a prototype application using the entities and rules that were defined in 15. In this example, a single page is created for every entity. The user creates these pages by selecting the relevant classes and incorporating input-fields for the class instance's properties. Figure 17 shows how the Rule Manager adds value to a prototype. When the user tries to create an announcement with an urgency-value other than “high” or “medium”, the corresponding rule is reflected back at them when attempting to save the instance. Figure 18 shows the enforced character minimum for an Administrator’s username. If no rules are broken, no rules are shown.

Figure 17: Creating an Announcement instance

Figure 18: Creating an Administrator instance

Lastly, figure 19 shows the enforcement of a direct relationship rule between the Lecturer and Lecture classes. All three Lecture instances are linked to a single Lecturer, this should be at least four.

Application home
Run

universitydatastore

-instancestore

->administrators

->announcements

->lecturers

->lectures

A lecturer should give at least four lectures.

save

Lecture - attendance77

Lecture - subjectIntroduction to Economics

Lecturer object (1) v

Lecture - attendance101

Lecture - subjectSupply and Demand

Lecturer object (1) v

Lecture - attendance67

Lecture - subjectThe Household

Lecturer object (1) v

Figure 19: Creating an Lecture instances

## 5.1 Error messaging

The NLP pipeline component boasts substantial input validation functionality. Currently, faulty rule sentences are filtered out of the input-stream and relevant error messages are generated. The error messages are not shown in the front-end yet. In this section, pairs of user input and their corresponding error messages are listed as a demonstration of the error messaging capability. All input-error pairs are based on the class model in figure 15.

The username of a lecturer and administrator may consist of letters and numbers.

`["Too many properties. If a rule contains two classes, it should contain no properties."]`

The Lecturer and Administrator classes, together with the username property, are mentioned in this rule. Since user input is restricted to rules with two classes and zero properties (direct relationship rule) or one class and one property (atomic rule) to ensure interpretability, the ‘too many properties’ error is appended.

A lecturer should give at most three announcements per month.

`["No direct connection. The classes ‘Lecturer’ and ‘Announcement’ were found. They do not share a direct connection, therefore no rules can be applied.", "Metadata found: Class: ‘Lecturer’. Class: ‘Announcement’."]`

When two classes are mentioned in a rule, this is interpreted as a direct relationship rule definition. Since Lecturer and Announcement are not connected, it appends the ‘no direct connection’ exception.

An announcement's subject should consist of letters only.

```
["Stray property found. Word 'subject' was linked to property 'subject'.",  
"Missing property. If a rule contains one class, it should also contain one  
property.', "Metadata found: Class: 'Announcement'. Stray properties found:  
'subject'."]
```

The subject property belongs to the Lecture class, not to Announcement. It is therefore marked as a 'stray property'. Because no correct property is mentioned, the 'missing property' exception is also appended.

A lecture should give at most two lecturers.

```
["m2o: wrong direction. Tried to enforce rule from 'Lecture' to 'Lecturer', but  
'Lecture' is the dependent class."]
```

Lastly, this example illustrates the validation of the directionality when submitting a rule. In a Many-to-One relationship, multiplicity rules can only be applied from the Many side, since Many-to-One already enforces that every dependent class instance can only be related to one independent class instance. Lecturer is the independent class in the model, therefore the directionality of the defined rule is indeed wrong.



## 6 Validation

To ascertain the value of the added rule handling capability within the ngUML solution, a user-experiment and survey were created. Firstly, respondents got a document that outlined the user’s workflow and explained all necessary information regarding the creation of a class model and a corresponding prototype application (see appendix A). Each respondent performed the experiment on my computer because that enabled me to fulfill some irrelevant technical prerequisites like starting the Docker containers and visiting the appropriate localhost addresses. The rest of the user workflow was performed by the users themselves. The experiment was targeted at respondents that know little to nothing about software development because the goal of the ngUML solution is to be accessible to this audience.

In the survey, each item consists of a statement. For every statement, the respondent could pick one of the following answers, indicating their agreement: “Strongly disagree”, “Somewhat disagree”, “Somewhat agree”, “Strongly agree”. In order, the statements were:

1. I am knowledgeable regarding software development
2. All steps of the validation workflow were clearly described
3. Managing rules in the model was intuitive
4. The added rules were sufficiently expressive
5. The rules were noticeably enforced in the prototype
6. The method whereby rules were shown to me was understandable
7. I was confronted with the rules at natural points in the prototype workflow
8. The addition of rule functionality adds value to the class model
9. The addition of rule functionality adds value to the prototype

### 6.1 Results

Figure 20 shows the results of the survey. The questions are numbered according to their order in the survey, as enumerated above.

Question	Number of responses per answer			
	Strongly disagree	Somewhat disagree	Somewhat agree	Strongly agree
1	1	-	-	3
2	-	-	2	2
3	-	-	1	3
4	-	1	3	-
5	-	-	-	4
6	-	-	1	3
7	-	-	3	1
8	-	-	3	1
9	-	-	1	3
Total respondents	4			

Figure 20: Survey results

Firstly, the table shows that most respondents indeed knew very little about software development. The responses to questions 3, 5, 6 and 7 show however that the management of rules in the class model and the incorporation of rules in the prototype application felt mostly natural to these inexperienced users. Not all respondents were as convinced about the expressiveness of the current rule handling capability (Q. 4), which is understandable when seen from the perspective of a user that purely cares for the usability of the end-product. Hopefully, now that the infrastructure for natural language rule conversion to Python source code is implemented, the marginal cost of adding implementable rule types is significantly reduced such that swift extension is feasible. Lastly, the responses to question 8 and 9 show that respondents were convinced of the value added by the implementation of rule handling capability to the ngUML solution, indicating the fruitfulness of future research on this front.

## 7 Further research

The ngUML research team’s objective to transform natural language requirements into software at run-time pushes boundaries and therefore yields many future objectives and challenges. The rule component is no exception. The following paragraphs will hopefully point to meaningful opportunities, together with recommended uses for this project’s deliverables.

**Adding implementable rule types** While sufficiently functioning as a proof-of-concept, it is clear that enforcing merely this subset of the rules domain does not begin to cover the real-world need for business rule expressiveness. Therefore, the functionality of the NLP Pipeline and the Formal Rule Generator components should be extended to handle a diverse range of rule types. The business rules taxonomy presented in this thesis will hopefully guide and structure future discourse on this front. Furthermore, by applying the described Django implementation types, together with their mapping to the rules taxonomy, much of the down payment has already been fulfilled, enabling expansion at a reduced marginal cost.

**Improving integration of the Rule Manager within application prototypes** Currently, the Rule Manager is not fully integrated into the application prototypes, resulting in limited functionality. Because the procedure whereby user-submitted data is converted to a new class instance was designed to efficiently handle a large variety of use-cases, Django’s recommended methods for validating such instances cannot easily be integrated. Thus, enabling full functionality of rule enforcement capabilities requires refactoring part of the Execution Engine (see figure 7). At the moment, rules are only displayed as a warning to the user in the prototype, the storage of non-compliant instances is not blocked.

**Extending the user’s rule editing capability** Because natural language is inherently ambiguous, the human-in-the-loop approach is an important component of ngUML’s vision. While existing rules can already be edited, this leaves a gap between the supplied user definition and its interpretation by the NLP Pipeline and Formal Rule Generator. It would thus be valuable to introduce a rule editing component that bypasses much of this interpretation by enabling lower-level access to the formal rule. To maintain accessibility, this tool should be visually driven to offset its more technical nature. It could include a selection menu of metadata entities, relationships (less than, unique compared to), features (invariant, context), and logical operators.

**Refining the business rule taxonomy** The goal of the presented rule taxonomy is pragmatic: it is meant to be a tool for structuring progress when extending the Rule Manager’s capabilities. While already being quite extensive, future progress on this front will lead to advanced insight and consequently call for refinement of the taxonomy to ensure maximal usability. It would therefore be wise to treat the taxonomy as a dynamic structure, involving it in the evolution of the ngUML project.

## 8 Conclusions

This paper delivers a proof-of-concept on the extensibility of the ngUML solution with rule handling capabilities. To realize this, a Rule Manager was built to apply business rules, defined in natural language, to the existing run-time prototype environment as Python source code. Important elements of this rule component are the NLP Pipeline and the Formal Rule Generator. Together, they form the transformation framework that maps user-submitted rules to a formal rule language that was defined to mediate between the expressiveness of natural language and the unequivocalness of source code. Lastly, the Rule Object Manager was implemented to handle rule instances and perform source code generation.

The implementation presented in this paper incorporates end-to-end rule handling capabilities into the ngUML solution, showing the viability of such an endeavour. There is still much room for improvement, and the validation results show that this is a fruitful avenue for future research. In an attempt to decrease the cost of functionality expansion, this paper presents a rules taxonomy, together with a mapping to implementation types in Django. Furthermore, the integral position of the formal rule language within the Rule Manager’s architecture is intended to function as a foundation for future extension, as the architecture to convert such rules to source code is already implemented.

This project was conducted as part of the Next Generation UML research group. By extending the functionality of their solution, the project contributes to their aspiration of improving the accessibility of software development for non-IT domain experts. Through this contribution, the project also addresses Model Driven Engineering’s challenged adoption in the software engineering community by contributing to the development of adequate tooling.

## References

- [1] A. Bucchiarone et al. “Grand challenges in model-driven engineering: an analysis of the state of the research”. In: *Lecture Notes in Computer Science* (2020), pp. 5–13. DOI: <https://doi.org/10.1007/s10270-019-00773-6>.
- [2] G. Mussbacker et al. “The Relevance of Model-Driven Engineering Thirty Years from Now”. In: *Lecture Notes in Computer Science* 8767 (2014), pp. 183–200. DOI: [https://doi.org/10.1007/3-540-47884-1\\_16](https://doi.org/10.1007/3-540-47884-1_16).
- [3] P. Qi et al. “Stanza: A Python Natural Language Processing Toolkit for Many Human Languages”. In: *Cornell University* (2020).
- [4] Vaswani et al. “Attention Is All You Need”. In: (2017). DOI: [10.48550/ARXIV.1706.03762](https://arxiv.org/abs/1706.03762). URL: <https://arxiv.org/abs/1706.03762>.
- [5] John Backus. “The History of FORTRAN I, II, and III”. In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 165–180. ISSN: 0362-1340. DOI: [10.1145/960118.808380](https://doi.org/10.1145/960118.808380). URL: <https://doi.org/10.1145/960118.808380>.
- [6] H.T. de Beer. *ALGOL, More than just ALGOL*. URL: [https://heerdebeer.org/History/Publications/ALGOL\\_more\\_than\\_just\\_ALGOL.pdf](https://heerdebeer.org/History/Publications/ALGOL_more_than_just_ALGOL.pdf).
- [7] Francis Bond. *HG4041 Theories of Grammar*. URL: <http://compling.hss.ntu.edu.sg/courses/hg4041/pdf/lec-11-1dd.pdf>.
- [8] R. Driessen. “UML Class Models as First-class Citizen: Metadata at Design-time and Runtime”. In: *Thesis Bachelor Informatica Economie* (2020).
- [9] Khaled El Emam and A. Günes Koru. “A Replicated Survey of IT Software Project Failures”. In: *IEEE Software* 25.5 (2008), pp. 84–90. DOI: [10.1109/MS.2008.107](https://doi.org/10.1109/MS.2008.107).
- [10] Instagram Engineering. *What Powers Instagram: Hundreds of Instances, Dozens of Technologies*. URL: <https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad>.
- [11] Django Software Foundation. *Design philosophies*. URL: <https://docs.djangoproject.com/en/2.0/misc/design-philosophies/>.
- [12] Django Software Foundation. *Form and field validation*. URL: <https://docs.djangoproject.com/en/4.0/ref/forms/validation/>.
- [13] Django Software Foundation. *Model field reference*. URL: <https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.IntegerField>.
- [14] Django Software Foundation. *Model instance reference*. URL: <https://docs.djangoproject.com/en/4.0/ref/models/instances/#validating-objects-1>.
- [15] Django Software Foundation. *Related objects reference*. URL: <https://docs.djangoproject.com/en/4.0/ref/models/relations/#related-objects-reference>.
- [16] Django Software Foundation. *Signals*. URL: <https://docs.djangoproject.com/en/4.0/ref/signals/#signals>.
- [17] Django Software Foundation. *Validators*. URL: <https://docs.djangoproject.com/en/4.0/ref/validators/#module-django.core.validators>.

- [18] Python Software Foundation. *Expressions*. URL: <https://docs.python.org/3/reference/expressions.html#operator-precedence>.
- [19] Object Management Group. *MDA Guide rev. 2.0*. URL: <https://www.omg.org/cgi-bin/doc?ormsc/14-06-01>.
- [20] Julia Hirschberg and Christopher D. Manning. “Advances in natural language processing”. In: *Science* 349.6245 (2015), pp. 261–266. DOI: [10.1126/science.aaa8685](https://doi.org/10.1126/science.aaa8685). URL: <https://www.science.org/doi/abs/10.1126/science.aaa8685>.
- [21] A. Hunt. *Dont Repeat Yourself*. URL: <https://wiki.c2.com/?DontRepeatYourself>.
- [22] ISO. *ISO/IEC 14977 : 1996(E)*. URL: <https://www.iso.org/standard/26153.html>.
- [23] S. Kent. “Model Driven Engineering”. In: *Lecture Notes in Computer Science* 2335 (2002), pp. 286–298. DOI: [https://doi.org/10.1007/3-540-47884-1\\_16](https://doi.org/10.1007/3-540-47884-1_16).
- [24] E.D. Liddy. “Natural Language Processing”. In: *Encyclopedia of Library and Information Science* (2001).
- [25] Geoff van der Meer. *How we use Python at Spotify*. URL: <https://engineering.atspotify.com/2013/03/how-we-use-python-at-spotify/>.
- [26] Mozilla. *Python*. URL: <https://web.archive.org/web/20120208002537/>.
- [27] Guus J. Ramackers et al. “From Prose to Prototype: Synthesising Executable UML Models from Natural Language”. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2021, pp. 380–389. DOI: [10.1109/MODELS-C53483.2021.00061](https://doi.org/10.1109/MODELS-C53483.2021.00061).
- [28] J.S. Reel. “Critical success factors in software projects”. In: *IEEE Software* 16.3 (1999), pp. 18–23. DOI: [10.1109/52.765782](https://doi.org/10.1109/52.765782).
- [29] A. J. Kfoury Robert N. Moll Michael A. Arbib. “An Introduction to Formal Language Theory”. In: (1988). DOI: <https://doi.org/10.1007/978-1-4613-9595-9>. URL: <https://link.springer.com/book/10.1007/978-1-4613-9595-9>.
- [30] D. S. Rosenblum. “A practical approach to programming with assertions”. In: *IEEE Transactions on Software Engineering* 21 (1995), pp. 19–31.
- [31] Niklaus Wirth. “What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?” In: 20.11 (1977), pp. 822–823. ISSN: 0001-0782. DOI: [10.1145/359863.359883](https://doi.org/10.1145/359863.359883). URL: <https://doi.org/10.1145/359863.359883>.
- [32] Graham Witt. “Writing Effective Business Rules”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. Elsevier, 2012. ISBN: 978-0-12-385051-5. URL: <https://doi.org/10.1016/C2010-0-66328-0>.
- [33] Tom Young et al. “Recent Trends in Deep Learning Based Natural Language Processing”. In: (2017). DOI: [10.48550/ARXIV.1708.02709](https://doi.org/10.48550/ARXIV.1708.02709). URL: <https://arxiv.org/abs/1708.02709>.
- [34] Vadim Zaytsev. “BNF Was Here: What Have We Done about the Unnecessary Diversity of Notation for Syntactic Definitions”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. Association for Computing Machinery, 2012, pp. 1910–1915. ISBN: 9781450308571. URL: <https://doi.org/10.1145/2245276.2232090>.

# A User-experiment document

## Validation of the ngUML rules component

This document is used to validate the functionality of the rules component that was added to the ngUML project. I will direct you to the front-end and prototype page. After this, please follow the directions in this document. First, the general workflow will be outlined to set up a model and prototype app. After this, examples are shown regarding the addition of rules to a model and how rules are shown in the prototype. At the end, you are asked to fill in a short survey. This process should take 15 minutes at most. Your participation is very much appreciated!

## Creating a model

On the home page, click on the blue button to create a model. As shown in figure 21, input a project name and description, and input the following model definition: "A student has a username and GPA. Business Courses and Science Courses are types of courses. Each course consists of multiple lectures, and have a course name, code and date. A course coordinator organizes the courses. Each course has one or more lecturers, a location, time slot, and set of dates. Lecturers will give lectures, and administrators will make announcements that are for a particular course. Each course will have assignments and an exam. Students must attend to the lectures, complete the assignments and take the exam. Lecturers will give grades. A course grade consists of an assignment grade and exam grade. An exam is either a first exam or a retake. Students will receive their course grades by email." Click on the submit button.

The screenshot shows the 'ngUML Requirements preprocessing' web application. The top navigation bar includes 'Home', 'Manage requirements', and 'Runnable prototype'. A sidebar on the left titled 'Generating a new model' has three steps: 'Choose a project Step 1' (selected), 'Create a new system Step 2', and 'Review extraction Step 3'. The main content area is titled 'Create a new project'. It contains a text box for 'Project name' with the value 'University data store'. Below it is a 'Project description' text area with the text: 'This application will store essential instance information and business rules about our university.' At the bottom, there are two sections: 'Write requirements' with a large text area containing a detailed model definition, and 'Upload requirements' with a file upload instruction: 'Max file size is 500kb. Only .txt or .wav files are supported.' and a dashed box for dragging and dropping a file. An 'OR' button is positioned between these two sections.

Figure 21: Creating the project

In the next screen, check all boxes and select the class model option, then click on the submit button. A class diagram will be rendered on the screen as in figure 22. Click on a class to display a menu on

the right side. The figure also shows how to add properties to a class and alter a relationship. Since only Many-to-One relationships can be implemented by the prototype generator at the moment, change all relationships to this type. It would be logical to define the Lecturer and Administrator classes on the One side of the relationships.

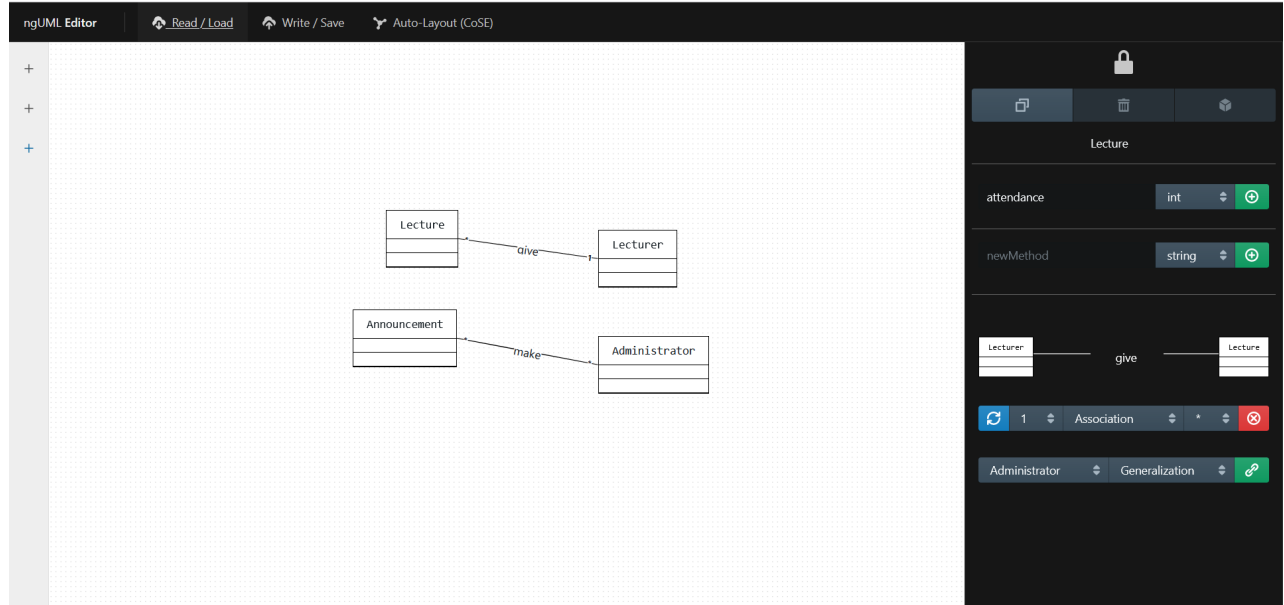


Figure 22: Adding properties and relationships to the class model

Add properties to the model, an example result is shown in 23. This figure also shows the next step: adding rules to the model. This thesis is a first proof-of-concept concerning the application of business rules to models, thus, not all rules are implementable yet. Some guidelines for creating rules:

1. Rules can only consist of two classes and zero properties, or one class and one property.
  - (a) Two classes and zero properties: these rules apply to direct relationships between classes. Taking figure 23 as an example, such rules concern the Administrator-Announcement relationship and the Lecturer-Lecture relationship.
  - (b) One class and one property: these rules apply to the property of a class. For example, the department of a Lecturer.
2. Rules constrict a domain.
  - (a) It constricts how many letters, characters or numbers a property can hold.
  - (b) It constricts what the numerical value of a property or relationship should be.
  - (c) It constricts the textual value that a property should hold, this is done by enclosing the intended value by double-quotes. For example, "A lecturer's faculty should be "Economics"."



3. Class-property rules can be restricted in all aforementioned ways. A direct relationship rule can only be restricted with regards to the numerical value (multiplicity), because the types are fixed at the corresponding classes. A direct relationship rule could be "A lecturer should give at least three lectures."
4. When a numerical value is mentioned without a type value, it is interpreted as the numerical value that a property or relationship should hold. For example, "A lecture's attendance should be at most one hundred" is interpreted as a value restriction only.
5. When a numerical value is mentioned together with a type value, it is interpreted as a type and length constriction on a property. For example, "An administrator's username should consist of at least 10 characters" restricts the quantity of input-symbols and what type they should be. Explicitly mentioning a type requirement together with a numerical value ensures that they are correctly linked, which is relevant when combining such statements using and's and or's.

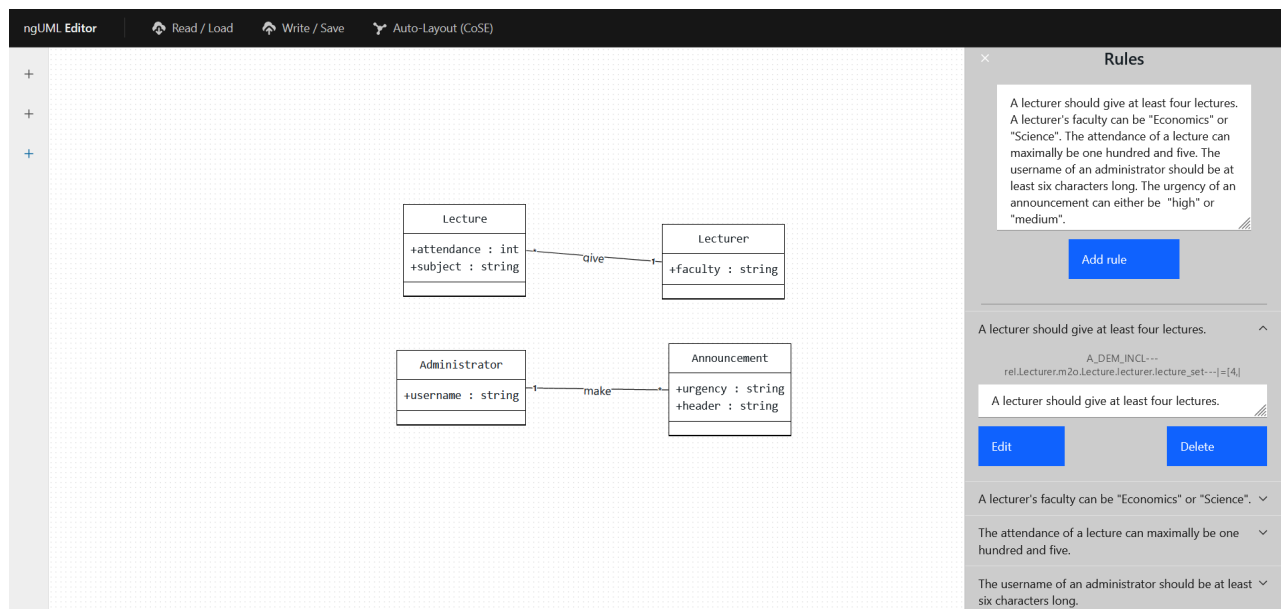


Figure 23: Adding rules to the class model

The rule menu can be opened by hovering over the plus symbol on the left side. In 23, the following rules specification is used: "A lecturer should give at least four lectures. A lecturer's faculty can be "Economics" or "Science". The attendance of a lecture can maximally be one hundred and five. The username of an administrator should be at least six characters long. The urgency of an announcement can either be "high" or "medium". After submitting the input, the rules are depicted in the rule menu. They can be altered and deleted as desired. When creating your own rules, they may not always show up in the menu. This is because faulty rules are filtered out of the input to ensure safe processing. Sadly, while the relevant error messages are already generated, they are not yet shown to the user.

## Rules in a prototype application

In this section, you will create a prototype application where the implemented rules are shown in action. On the home screen, click on the “Applications” button to create a new prototype. After giving it a name, add all classifiers by selecting them from the drop-down menu and clicking on “Add”. Then, click on “Add Category” and give it a name. Add pages to the prototype as shown in figure 24. Since the further construction of these pages is a bit difficult and irrelevant to the added rule functionality, I will do this for you.

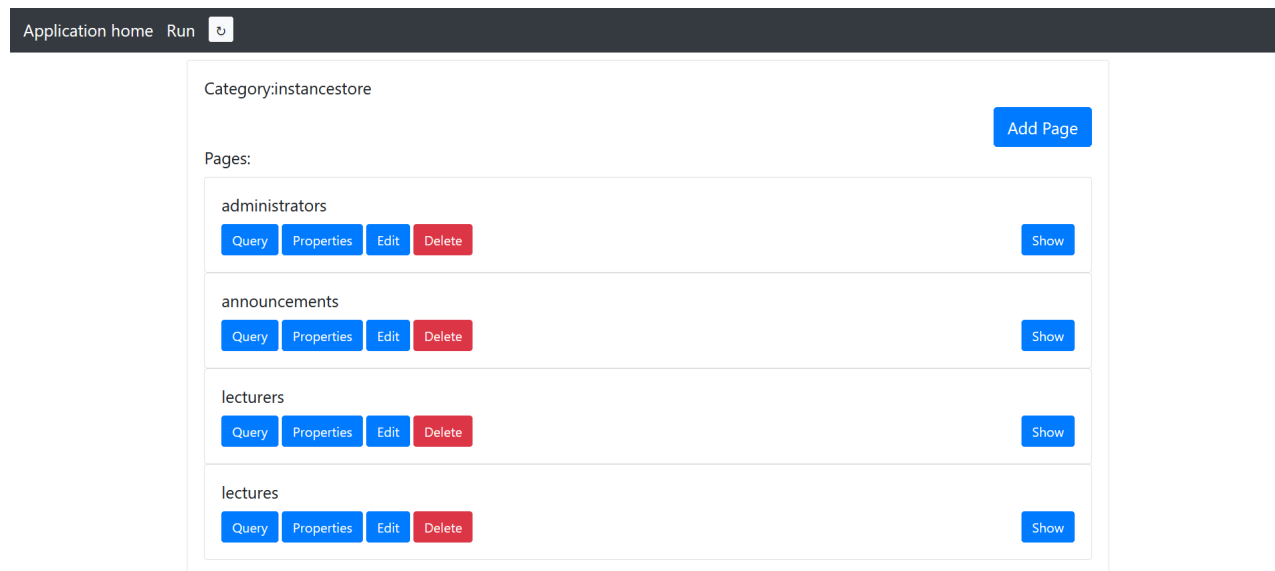



Figure 24: Adding pages to the prototype

Check out the pages using the “Show” button. It will serve a simple page where you can create class instances. For example, on the “announcements” page, you can create an announcement with a value for “header” and “urgency”. Check what happens when you break a rule. When testing the application of a direct-relationship rule, first create a Lecturer instance and then link this instance to the Lectures you create using the drop-down menu next to every Lecture instance. You will see that the quantity constraint between the Lecturer-Lecture classes will be shown when you break it, see figure 25 for an example.

Application home Run 

universitydatastore  
-instancestore  
->administrators  
->announcements  
->lecturers  
->lectures

A lecturer should give at least four lectures.

save

Lecture - attendance77

Lecture - subjectIntroduction to Economics

Lecturer object (1) ▾

Add more

Lecture - attendance101

Lecture - subjectSupply and Demand

Lecturer object (1) ▾

Add more

Lecture - attendance67

Lecture - subjectThe Household

Lecturer object (1) ▾

Add more

Figure 25: Adding Lectures to a Lecturer in the prototype

## Survey

That concludes the workflow. Please follow [this link](#) to fill in the survey. Again, thank you for participating!