PhD-FSTM-2022-85
The Faculty of Science, Technology and Medicine

DISSERTATION

Defense held on 16/09/2022 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Alvaro Mario VEIZAGA CAMPERO
Born on 28 July 1985 in Cochabamba, Bolivia

MODEL-BASED SPECIFICATION AND ANALYSIS OF NATURAL LANGUAGE REQUIREMENTS IN THE FINANCIAL DOMAIN

Dissertation defense committee

Dr. Lionel BRIAND, Dissertation Supervisor
Professor, University of Luxembourg

Dr. Domenico BIANCULLI, Chairman
Professor, University of Luxembourg

Dr. Seung Yeob SHIN, Vice Chairman
Research Scientist, University of Luxembourg

Dr. Alessio FERRARI, Member
Research Scientist, National Research Council of Italy (CNR)

Dr. Fabiano DALPIAZ, Member
Professor, Utrecht University, the Netherlands

Acknowledgement

It is a genuine pleasure to express my gratitude to my supervisor Prof. Dr. Lionel Briand, for accepting me into his research group and for teaching, guiding, and leading me in this process with such dedication and patience. I am honored to have had the opportunity to learn from his academic excellence and his remarkable experience.

I would like to thank my co-supervisor Seung Yeob Shin, for his insights, guidance, and valuable feedback while developing our research work. All the things I have learned from him, professionally and personally, have had a valuable impact on my persona.

A particular acknowledgment to my former co-advisors Prof. Dr. Mehrdad Sabetzadeh, Dr. Mauricio Alferez and Dr. Damiano Torre for the support they gave me throughout the process of my doctoral research.

I would like to thank Dr. Alessio Ferrari, Prof. Dr. Fabiano Dalpiaz, and Prof. Dr. Domenico Bianculli for having accepted to be part of the jury that evaluated my doctoral thesis.

I would also like to express my gratitude to the Investment Fund Services team of Clearstream Luxembourg for their valuable feedback to the research work and for providing information to develop case studies.

A special thank you goes to my parents, Zunilda and Mario, for being my strongest motivation and source of strength, not only for developing this doctoral thesis but also for my personal and professional growth. I thank my sister, Adriana, for her unconditional help and support.

Finally, I would like to thank my research group, Software Verification and Validation, for the friendly working environment. Thank you for the helpful advice, support, feedback and exchanges received from Jaekwon, Angelo, Fitash, and my other colleagues.

Abstract

Software requirements form an important part of the software development process. In many software projects conducted by companies in the financial sector, analysts specify software requirements using a combination of models and natural language (NL). Neither models nor NL requirements provide a complete picture of the information in the software system, and NL is highly prone to quality issues, such as vagueness, ambiguity, and incompleteness. Poorly written requirements are difficult to communicate and reduce the opportunity to process requirements automatically, particularly the automation of tedious and error-prone tasks, such as deriving acceptance criteria (AC). AC are conditions that a system must meet to be consistent with its requirements and be accepted by its stakeholders. AC are derived by developers and testers from requirement models. To obtain a precise AC, it is necessary to reconcile the information content in NL requirements and the requirement models.

In collaboration with an industrial partner from the financial domain, we first systematically developed and evaluated a controlled natural language (CNL) named Rimay to help analysts write functional requirements. We then proposed an approach that detects common syntactic and semantic errors in NL requirements. Our approach suggests Rimay patterns to fix errors and convert NL requirements into Rimay requirements. Based on our results, we propose a semiautomated approach that reconciles the content in the NL requirements with that in the requirement models. Our approach helps modelers enrich their models with information extracted from NL requirements. Finally, an existing test-specification derivation technique was applied to the enriched model to generate AC.

The first contribution of this dissertation is a qualitative methodology that can be used to systematically define a CNL for specifying functional requirements. This methodology was used to create Rimay, a CNL grammar, to specify functional requirements. This CNL was derived after an extensive qualitative analysis of a large number of industrial requirements and by following a systematic process using lexical resources. An empirical evaluation of our CNL (Rimay) in a realistic setting through an industrial case study demonstrated that 88% of the requirements used in our empirical evaluation were successfully rephrased using Rimay.

The second contribution of this dissertation is an automated approach that detects syntactic and semantic errors in unstructured NL requirements. We refer to these errors as smells. To this end, we first proposed a set of 10 common smells found in the NL requirements of financial applications. We then derived a set of 10 Rimay patterns as a suggestion to fix the smells. Finally, we developed an automatic approach that analyzes the syntax and semantics of NL requirements to detect any present smells and then

suggests a Rimay pattern to fix the smell. We evaluated our approach using an industrial case study that obtained promising results for detecting smells in NL requirements (precision 88%) and for suggesting Rimay patterns (precision 89%).

The last contribution of this dissertation was prompted by the observation that a reconciliation of the information content in the NL requirements and the associated models is necessary to obtain precise AC. To achieve this, we define a set of 13 information extraction rules that automatically extract AC-related information from NL requirements written in Rimay. Next, we propose a systematic method that generates recommendations for model enrichment based on the information extracted from the 13 extraction rules. Using a real case study from the financial domain, we evaluated the usefulness of the AC-related model enrichments recommended by our approach. The domain experts found that 89% of the recommended enrichments were relevant to AC, but absent from the original model (precision of 89%).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Software requirements have emerged as powerful instruments for software development. Requirements describe the functional capacities, features, qualities, and operational constraints of a system [65, 51]. Defining and identifying software requirements is an evolving process that occurs repeatedly throughout the lifecycle of a project until a consensus is reached among all interested parties.

The quality of software requirements has a great influence on the quality of a software system. Previous studies have established that poorly written requirements are one of the main causes of software project failures in the industry. Poorly written requirements are characterized as unclear, ambiguous, or incomplete and are difficult to communicate among project stakeholders [1, 28, 69]. In addition, it is well known that the cost of fixing problems related to requirements increases as the life cycle of software development progresses [11]. Therefore, it is paramount that requirement problems be identified in the early stages of a software development project.

In practice, analysts from companies in the financial sector express software requirements using natural language (NL), along with models usually represented in some type of unified modeling language (UML) language [45]. Requirement models, such as activity diagrams, class diagrams, and use-case diagrams, are used to describe the functionalities of a software system. NL requirements often provide fine-grained details about the software system that would not normally be included in the requirements models.

NL has been widely used to express software requirements. Existing research indicates that most users (61%) prefer to express requirements using NL [32] and 52% of software requirements are written in NL [45]. Despite the popularity of NL, common problems arise when using it to specify software requirements. These problems include poor testability, inappropriate implementation, wordiness, under-specification, incompleteness, duplication, omission, complexity, vagueness, and ambiguity [42, 22].

Requirement models have been increasingly used by large financial institutions, where requirements are modeled and expressed using UML language. In this process, analysts specify system behaviors using activity diagrams. The types and properties of data or objects manipulated in activity diagrams are

described using class diagrams. Analysts then define the actors in the use-case diagrams. These actors perform the actions described in the activity diagrams.

In this dissertation, we seek to address the barriers encountered by the financial sector when writing requirements. The current practice of companies in this sector is to write NL requirements using a general-purpose text editor without enforcing any requirement structure. The lack of structured requirements poses major difficulties in automating certain tasks in the software development process (e.g., generating acceptance criteria [AC]). AC are conditions derived by developers and testers based on requirement models, which a system must meet to be consistent with its requirements. To obtain precise AC, it is necessary to reconcile the information content of both NL requirements and models. To our knowledge, there is no existing approach that considers the above challenges as observed in companies in the financial domain. Existing approaches are currently limited to addressing these challenges in other domains.

Our industrial partner is Clearstream Services SA Luxembourg, a post-trade services provider owned by Deutsche Borse AG. Clearstream reported the following issues to us: (1) communication problems and delays in the software development process are caused by poorly expressed requirements; and (2) Clearstream wishes to leverage NL requirements information to enable automation of abstract test case generation.

Throughout this dissertation, we provide effective solutions to tackle the aforementioned needs. More concretely, we present solutions that improve the quality of requirements. High-quality requirements can enable task automation for tedious and error-prone activities performed on software development process requirements (e.g., the generation of AC). We believe that this dissertation makes a major contribution to research in the field of requirements engineering. In the following section, we describe our contributions in detail.

1.2 Research Contributions

In this dissertation, we propose solutions to improve the quality of NL requirements. Furthermore, we introduce a methodology that aims to reconcile the information content present in NL requirements and requirement models. Our solutions have been developed and empirically evaluated in close collaboration with our industrial partner, Clearstream. In this dissertation, we make several contributions.

Concerning the systematic building of a controlled NL for functional requirements, we developed and evaluated a controlled natural language (CNL) named Rimay to help analysts write functional requirements. We relied on Grounded Theory to build Rimay and followed well-known guidelines for conducting and reporting industrial case study research. The main contributions of this chapter are as follows: (1) a qualitative methodology to systematically define a CNL for functional requirements (intended for general use across information-system domains), (2) a CNL grammar to represent functional requirements (derived from our experience in the financial domain but applicable, possibly with adaptations, to other information-system domains), and (3) an empirical evaluation of our CNL (Rimay) through an industrial case study. These contributions have already been published [71] and are presented in Chapter 3.

In terms of requirement quality assurance, we proposed a set of 10 smells that represent the most common syntactic and semantic errors found in the NL requirements of some financial applications. We then derived 10 Rimay patterns. These patterns assist analysts in fixing the smells found in NL requirements. Our results led us to propose an automated approach that automatically detects the 10 smells in NL requirements and suggests Rimay patterns to analysts to improve the overall quality of NL

requirements. The results of a case study to assess the accuracy of the results obtained by our approach, as well as the aforementioned contributions, are outlined in Chapter 4.

Finally, concerning efforts to leverage NL requirements with the goal of deriving better acceptance criteria from models, our work was prompted by the observation that a reconciliation of the information content in NL requirements and models is necessary to obtain precise AC. We performed this reconciliation by devising an approach that automatically extracts AC-related information from NL requirements and helps modelers enrich their models with the extracted information. An existing AC derivation technique was then applied to the model, which has now been enriched by the information extracted from the NL requirements. Using a real case study from the financial domain, we evaluated the usefulness of the AC-related model enrichments recommended by our approach. These contributions have been published as a conference paper [72] and are presented in Chapter 5.

1.3 Dissertation Outline

Chapter 2 provides some fundamental background on controlled natural languages, natural language processing techniques, acceptance criteria, and the Unified Modeling Language.
Chapter 3 presents our methodology for defining controlled natural languages and describes the creation of our CNL called Rimay for writing functional requirements.
Chapter 4 introduces our smell detector tool that finds common syntactic and semantic errors in natural language requirements and guides analysts to fix the errors.
Chapter 5 describes our methodology that supports the model-based derivation of acceptance criteria (AC) by enriching requirements models with AC-related information in NL requirements.
Chapter 6 summarizes the dissertation contributions and discusses perspectives on future works.

Chapter 2

Background

This chapter introduces the technical background necessary to understand the work carried out in this dissertation. First, we introduce the notion of controlled natural languages (CNL). Next, we provide an analysis of natural language processing (NLP) techniques and lexical resources. Finally, we provide a background on the generation of acceptance criteria (AC) and the Unified Modeling Language (UML).

2.1 Controlled Natural Languages

A CNL is engineered to define a restricted natural language for a specific domain. Such languages restrict grammatical structures and provide language constructs that allow analysts to define precisely the syntax and semantics of a language [51]. A CNL is made up of (1) a set of predefined sentence structures that restrict the syntax of NL and precisely define the semantics of the statements written using this language, and (2) a lexicon that contains the allowed set of words and the domain terminology to be used in the language [51, 36].

CNLs are intended to solve communication gaps between stakeholders by using standard structures and terminology. These gaps often include misunderstandings about aspects of the underlying domain. However, CNLs tend to reduce the expressiveness of a language by imposing restrictions on structure and vocabulary. Furthermore, to apply CNLs, stakeholders need training to become familiar with the structures and terminology used by the language [51]. In requirements engineering, requirements written using CNLs have the following advantages when compared to requirements written using a natural language (NL) [51, 59]: (1) requirements are easy to understand, (2) requirements are less ambiguous because they have simplified grammar and predefined vocabulary with exact semantics, and (3) requirements are semantically verifiable due to formal grammar and predefined terms.

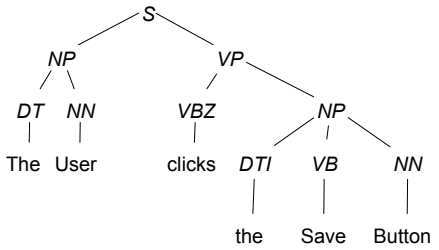
2.2 Natural Language Processing

NLP is a field of artificial intelligence that aims to understand, analyze, manipulate, and produce NL through computational techniques [40]. NLP techniques execute a pipeline to carry out multiple analyses,

such as tokenization, syntax analysis, and semantic analysis. These techniques have been widely used and are at the core of many applications that we use on a daily basis (e.g., chatbots, text translation, and speech-to-text conversion). In requirements engineering, NLP techniques have enabled, to some extent, the automation of several tasks, such as test automation [72, 2] and model derivation [5]. The following is a short description of NLP techniques and lexical resources used in this dissertation.

Constituency Parser

The constituency parser is a statistical parser that maps NL sentences into a constituency structure. The latter is a syntactic representation that recursively breaks down a sentence into smaller segments. These segments are classified, based on their internal structure, into noun phrases, verb phrases, etc. [30]. Constituency structures are normally represented using syntax trees. Figure 2.1 shows an example of a constituent structure for a requirement written in English.



Symbol	Meaning
A << B	A is an ancestor of B
A < B	A is the parent of B
A <, B	B is the first child of A
A >, B	A is the first child of B
A >: B	A is the only child of B

Figure 2.1: Example of a syntax tree and summary of Tregex operators

Syntax Tree

A syntax or a parse tree represents the syntactic structure of a grammatical sentence. It contains all the steps of sentence derivation from the root node. Figure 2.1 shows an example of a syntax tree. The tree has a top-down structure that displays the root of the tree at the top and the leaves at the bottom. All tree nodes are connected by links called edges, which capture the relationships between the nodes.

Tregex

Tregex is a tree query language used to define patterns that match the content of tree nodes with their hierarchical relationships [38]. The content of the tree nodes includes lemmas, POS tags, and characters. Figure 2.1 shows an excerpt of the Tregex operators provided by Tregex, along with an example of a syntax tree. In this example, the Tregex pattern "`VP` <, `VBZ`" matches the content "clicks" of the syntax tree example. The verb "clicks" is the first child of the verb phrase `VP`.

Part-of-Speech Tagging

Part-of-Speech (POS) Tagging is an NLP process in which the text is analyzed so that each word and other tokens of the text can be labeled with the correct part of speech [70]. Parts of speech include nouns, verbs, adjectives, and so on. The tag is assigned according to the definition of the word and its context. The POS tags used in this dissertation follow the Penn Treebank tag set [41].

Lexical Resources

WordNet

WordNet [46] is a domain-independent linguistic resource which provides, among several other things, more than 117000 *synsets*. Synsets are synonyms –words that denote the same concept and are interchangeable in many contexts– grouped into sets. Each synset contains (a) a brief definition ("gloss"), (b) the synset members, and, in most cases, (c) one or more short sentences illustrating the use of the synset members. Each synset member is a synonym sharing the same sense of the other members of the synset. Synset members use the format $word\#sense\ number$. For example, Table 2.1 shows the WordNet entry for the verb *create*. This entry has six synsets. The sixth synset contains the following information: (a) gloss, "create or manufacture a man-made product", (b) three synset members, *produce#2*, *make#6*, and *create#6* and (c) an example of how to use the synset member *produce#2*, "We produce more cars than we can sell".

Table 2.1: WordNet entry for the verb create

#	Gloss	Synset Members	Example
1	Make or cause to be or to become	make#3, create#1	"make a mess in one's office"
2	Bring into existence	create#2	"He created a new movement in painting"
3	Pursue a creative activity	create#3	"Don't disturb him–he is creating"
4	Invest with a new title, office, or rank	create#4	"Create one a peer"
5	Create by artistic means	create#5, make#9	"Schoenberg created twelve-tone music"
6	Create or manufacture a man-made product	produce#2 make#6, create#6	"We produce more cars than we can sell"

VerbNet

VerbNet [33] is a domain-independent, hierarchical verb lexicon of approximately 5800 English verbs. It clusters verbs into over 270 verb classes, based on their shared syntactic behaviors. Each verb in VerbNet is mapped to its corresponding synsets in WordNet, if the mapping exists. In VerbNet, a verb is always a member of a verb class and each verb class is identified by a unique code composed of a name and a suffix. The suffix reveals the hierarchical level of a verb class, e.g., two of the sub-classes of the root class *multiply-108* are *multiply-108-1* and *multiply-108-2*. In VerbNet, the sub-classes inherit features from the root class and specify further syntactic and semantic commonalities among their verb members. For example, each of the sub-classes of *multiply-108* uses the same syntactic structure which is defined as a noun phrase followed by a verb, a noun phrase, and a prepositional phrase. However, each sub-class uses different prepositions in the prepositional phrase. In particular, the subclass *multiply-108-1* has the verb members *divide* and *multiply* and uses the preposition *by* as in the phrase "I multiplied x by y". The subclass *multiply-108-2* has verb members such as *deduct*, *factor*, and *subtract* and uses the preposition

from as in the phrase “I subtracted x from y". Note that the verb *subtract* has no semantic similarity with the verb *multiply* that appears in the subclass name *multiply-108-2*. However, the verb *subtract* is a member of the subclass as they share syntactic behavior with the verbs *deduct* and *factor*.

2.2.1 UML

Unified Modeling Language (UML) is a general-purpose graphical language used to visualize, specify, construct, and document the artifacts of a software-intensive system. UML provides a standard way to write the blueprints of a system. This includes conceptual aspects such as business processes, system functions, and more concrete aspects such as programming language statements, database schemas, and reusable software components [49]. Diagrams modeled using UML can be divided into structural and behavioral types. Structural diagrams model the elements present in the software system, whereas behavioral diagrams describe various aspects of the system behavior. In this dissertation, we make use of three UML diagrams: activity, class, and use case diagrams.

Activity diagrams describe the behavior of a system. These diagrams provide a view of the workflows of the activities in the system. The activities of the activity diagram can be decomposed into sub-activities. Atomic activities are called actions.

Class diagrams provide a structural view of a software system by describing a system’s classes, objects, interfaces, attributes and operations, as well as the relationships between the classes [24].

Use case diagrams are widely used to capture the functional requirements of a system. They provide a description of the usage of the system by describing the interactions between the system and the users of the system. The information described in the use case includes the name of the use case, description, precondition(s), actors, dependencies, the flow of events, basic and alternative flows, and post-condition(s) [24].

2.2.2 Acceptance Testing

Acceptance testing is designed to determine whether completed software meets the needs of the customer. To do so, acceptance testing assesses software against requirements elicited in the early stage of the software development process (i.e., the requirements analysis phase). Requirements describe the functional capacities, features, qualities, and operational constraints of a system [65, 51]. Acceptance testing begins after conducting the system testing and involves stakeholders who have strong domain knowledge [3]. Figure 2.2 depicts the software development activities and testing levels.

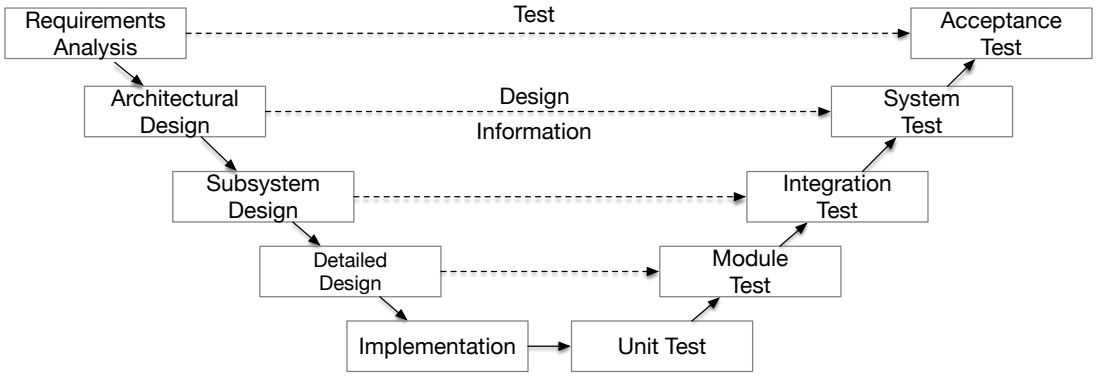


Figure 2.2: Software development activities and testing levels – the “V Model” [3]

Acceptance Criteria The definition of Acceptance Criteria (AC) is an important step in acceptance testing. AC distinguish the incorrect behaviors of the system from the correct ones. NL has been largely used to specify AC because it can be understood by all stakeholders (e.g., the development team, clients, and project management). In Behavior-Driven Development (BDD), AC are commonly expressed using the Gherkin language [76]. This language uses a *Given-When-Then* structure where *Given* describes the initial context, *When* describes an event or action, and *Then* describes the expected result. Here is an example Gherkin scenario: "*Given* order does not exist in System-A *When* System-A performs Create Order *Then* order exists in System-A". Furthermore, the syntax of the Gherkin language enables engineers to generate executable test cases by matching AC text to application programming interfaces (APIs) [76].

Chapter 3

On Systematically Building a CNL for Functional Requirements

3.1 Motivations and Contributions

Requirements are considered as one of the fundamental pillars of software development. For many systems in industry, requirements are predominantly expressed in natural language (NL). Natural language is advantageous in that it can be used in all application domains and understood virtually by all project stakeholders [51]. Supporting this statement, a study reported that 52% of software requirements specifications (SRSs) are written in NL [45]. Furthermore, Zhao et al. [82] posit that NL will continue to serve as the *lingua franca* for requirements in the future. Despite its pervasive use, undisciplined use of NL can bring about a variety of quality issues. Common problems with NL requirements include: poor testability, inappropriate implementation, wordiness, under-specification, incompleteness, duplication, omission, complexity, vagueness, and ambiguity [42, 22].

Further, requirements often change throughout a project's lifespan until a consensus is reached among stakeholders. Requirements changes lead to significant additional costs that vary according to the project phase [28]; it has long been known that the cost of fixing problems related to requirements increases rapidly when progressing through the software development phases [11].

The ultimate quality of a software system greatly depends on the quality of its requirements. Empirical evidence shows that the state of practice for acquiring and documenting requirements is still far from satisfactory [57, 63, 77]. Different studies have reported that one of the main causes of software project failures in industry is related to poorly written requirements, i.e., requirements that are unclear, ambiguous, or incomplete [1, 28, 69]. Poorly written requirements are difficult to communicate and reduce the opportunity to process requirements automatically, for example, to extract models [5] or derive test specifications [2].

The problem we address in this chapter was borne out of a practical need observed across many industrial domains. For example, in the financial domain, the current practice is to write system requirements

using a general-purpose text editor without enforcing any requirement structure. This is the case for our industrial partner, Clearstream Services SA Luxembourg – a post-trade services provider owned by Deutsche Borse AG. Clearstream reported that several communication problems and delays arise from requirements that are not stated precisely enough, particularly in situations where the project development tasks are divided across several teams in different countries. This problem is compounded by the fact that Clearstream typically has to deal with SRSs written in NL that are created by domain experts (from now on, we refer to them as "financial analysts"), who do not necessarily possess sufficient expertise in requirements elicitation and definition.

Furthermore, other stakeholders at different levels of the organization, e.g., customer service, also need to be able to process the requirements and validate them according to their specific needs [17]. As a result, there is a tension between the pressure to use NL in practice and the need to be more precise and resorting to formal languages [78]. Controlled natural languages (CNLs) strike a balance between the usability of NL on the one hand and the rigour of formal methods on the other. A CNL is a set of predefined sentence structures that restrict the syntax of NL and precisely define the semantics of the statements written using these predefined structures [51].

In this chapter, we concern ourselves with developing a CNL for writing requirements for financial applications. We have named our CNL Rimay, which means "language" in Quechua. We focus on *functional requirements*, noting that the vast majority of the requirements written by our industrial partner are functional, and that financial analysts find most of the ambiguity and imprecision issues in functional requirements. The functional requirements produced by Rimay are intended to replace unrestricted requirements and, as a result, enable the automation of certain tasks, such as the generation of acceptance-test criteria [72]. In our context, a functional requirement specifies what system response an actor is expected to receive when providing certain inputs, if certain conditions are met. We consider every other type of requirement to be non-functional.

While Rimay is grounded in requirements for financial applications, it also overlaps with other Requirements Engineering ontologies such as the Core Ontology for REquirements (CORE) [31], whose development was inspired by the work of Zave and Jackson [81]. In short, CORE aims to cover all the basic concerns that stakeholders communicate during the requirements elicitation process (beliefs, desires, intentions, and evaluations) by introducing a set of concepts (Goal, Plan, Domain assumption, and Evaluation). Each concept, except Plan, has subcategories. For instance, the goal concept has three subcategories in CORE: Functional goal, Quality constraint, and Softgoal. The condition structure and system response of Rimay correspond to the Functional goal concept of CORE.

Finally, although our work draws on the requirements of financial applications, this domain shares several common characteristics with other domains where (data-centric) information systems are developed. We therefore anticipate that our results, including our methodology, lessons learned, and Rimay itself, can be a useful stepping stone for building CNLs in other related domains. This said, we acknowledge that additional empirical work remains necessary to substantiate claims about usefulness beyond our current domain of investigation, i.e., finance.

Our investigation is guided by the following research questions (RQs):

- **RQ1: What information content should one account for in the requirements for financial applications?** In this RQ, we want to identify, in the requirements provided by our industrial partner, the

information content used by financial analysts. This information is a prerequisite for the design of the Rimay grammar.

- **RQ2: Considering the stakeholders, how can we represent the information content of requirements for financial applications?** After we identify the information content used by our industrial partner to represent requirements, we want to find out the structures of the requirements that our CNL should support. These structures follow recommended syntactic structures and define mandatory and optional information.

- **RQ3: How well can Rimay express the requirements of previously unseen documents?** After building our CNL grammar, we need to determine how well it can capture requirements in unseen SRSs.

- **RQ4: How quickly does Rimay converge towards a stable state?** Rimay reaches a stable state when it does not need to continuously evolve (i.e., no addition of new rules and no updates to the existing rules) in response to the analysis of new (unseen) SRSs. To assess stability, we use the notion of saturation. Saturation occurs, in a qualitative study, when no new information seems to emerge during coding.

In this chapter, we use a total of 15 SRSs written by financial analysts at Clearstream. These SRSs describe different projects that cover a range of activities: nine discuss the updating of existing applications, two concern the compliance of the applications with new regulations, two describe the creation of new applications, and the last two describe the migration of existing applications to more sophisticated technologies. Of the 15 SRSs, 11 are used in our qualitative study to answer RQ1 and RQ2, and the other four in our empirical evaluation to answer RQ3 and RQ4.

We use a combination of Grounded Theory and Case Study Research to address the four research questions posed above. The main contributions of this work can be summarized as follows:

(1) A qualitative methodology aimed at defining a CNL for functional requirements (RQ1). We rely on Grounded Theory for developing Rimay. Grounded Theory is a systematic methodology for building a theory from data. The goal of Grounded Theory is to generate theory rather than test or validate an existing theory [67]. Our methodology is general and can serve as a good guiding framework for building CNLs systematically. We rely on an analysis procedure named *protocol coding* [58], which aims at collecting qualitative data according to a pre-established theory, i.e., set of codes. Protocol coding allows additional codes to be defined when the set of pre-established codes is not sufficient. A code in qualitative data analysis is most often a word or short phrase that symbolically assigns a summative, salient, essence-capturing, and/or evocative attribute for a portion of language-based or visual data [58]. In the context of our chapter, a code identifies a group of verbs that share the same information content in an NL requirement. As explained in Section 3.3.3, most of the codes are pre-existing verb-class identifiers available in a well-known lexicon named VerbNet[1]. In addition, we use WordNet[2] to verify the verb senses of the requirements. The fact that we use domain-independent lexical resources and include no keywords specific to the financial domain in Rimay, makes our approach more likely to have wider applicability to information systems in general. We conduct our qualitative study on 11 SRSs that contain 2755 requirements in total.

[1] `https://verbs.colorado.edu/verbnet/` (last access on 17 August 2022)
[2] `https://wordnet.princeton.edu/` (last access on 17 August 2022)

(2) A CNL grammar (RQ2) targeting financial applications in particular and information systems in general. We apply restrictions on vocabulary, grammar, and semantics. The Rimay grammar accounts for a large variety of system responses and conditions, while following recommended syntactic structures for requirements (e.g., the use of active voice). Also, the Rimay grammar defines mandatory information content to enforce the completeness of functional requirements. In addition to the grammar, we generate a user-friendly and full-featured editor using the language engineering framework Xtext [3].

(3) An empirical evaluation of Rimay (RQ3 and RQ4). We report on a case study conducted within the financial domain. We evaluate Rimay on four SRSs containing 460 requirements to demonstrate the feasibility and benefits of applying Rimay in a realistic context. We use *saturation* to find the point in our evaluation where enough SRS content has been analyzed to ensure that Rimay is stable for specifying requirements for the financial domain. Furthermore, we use a *z-test for differences in proportions* to confirm that additional enhancements to Rimay are unlikely to bring significant benefits.

The chapter is structured as follows: Section 3.2 introduces the related work. Section 3.3 presents a qualitative study aimed at analyzing the information content in the requirements provided by Clearstream (our industrial partner). In Section 3.4, we describe the details of Rimay. Section 3.5 describes a case study that evaluates Rimay. Threats to the validity of our results are discussed in Section 3.6. Section 3.7 discusses practical considerations and, finally, our conclusions are provided in Section 3.8.

3.2 Related Work

Numerous studies have been conducted with a focus on NL requirements quality improvement. Pohl [51] presents three common techniques for improving the quality of NL requirements by reducing vagueness, incompleteness and ambiguity:

Glossaries. Requirements glossaries make explicit and provide definitions for the salient terms in a SRS. Requirements glossaries may further provide information about the synonyms, related terms, and example usages of the salient terms [7].

Patterns. They are pre-defined sentence structures that contain optional and mandatory components. Patterns restrict the syntax of the text and are meant to help stakeholders in writing more standardized NL requirements and thus circumventing frequent mistakes.

Controlled natural languages. They are considered an extension of the pattern category which, in addition to restricting the syntax (the grammatical structures), also provide language constructs with which it is possible to precisely define the semantics of NL requirements.

In this chapter, we build a CNL to represent functional requirements in the financial domain. However, given that Rimay does not rely on any domain-specific constructs (Sections 2.2 and 3.3), it could also be applied to other (data-centric) information systems in different domains.

Given our objective, we focus here on approaches and studies related to CNLs and patterns for expressing NL requirements. We searched relevant approaches and studies in four well-known digital libraries: ACM, IEEE, Springer, and ScienceDirect. In addition, we considered relevant surveys that discuss CNLs and patterns for expressing NL requirements. We selected 11 studies, directly relevant to our work, that focus on improving NL requirements through the use of patterns or CNLs. Table 3.1 outlines the main characteristics of these studies. The first column of the table provides a reference to each study.

[3] `https://www.eclipse.org/Xtext/` (last access on 17 August 2022)

The second column indicates the type of the approach, i.e., Pattern or CNL. In order to obtain a more thorough picture of the literature, although our work is focused on functional requirements, our analysis of the related work does not exclude references that exclusively address non-functional requirements. The third column shows the type of the requirements that the approach supports: Functional Requirements (FR), Non-Functional Requirements (NFR), or both. Additionally, the third column includes the domain in which the patterns and CNLs were created. There are two strands of work: domain-independent and domain-specific (i.e., automotive, business, healthcare, performance, embedded systems, and data-flow reactive systems).

The fourth column indicates whether an empirical study was conducted and evaluated in a systematic manner. The fifth column shows whether the proposed CNL or pattern was somehow evaluated. Finally, the sixth column reports on whether tool support was provided. We discuss the selected studies next.

Table 3.1: Summary of related work

Study Reference	Type of Approach	Type of Requirements	Systematic Study	Evaluation	Tool Support
Pohl and Rupp [52]	Pattern	FR (Domain - Independent)	No	No	No
Mavin et al. [44]	Pattern	FR (Domain - Independent)	No	Yes	Yes
Withall [74]	Pattern	Both (Business)	No	No	No
Riaz et al. [55]	Pattern	NFR (Healthcare)	No	No	Yes
Eckhardt et al. [19]	Pattern	NFR (Performance)	Yes	Yes	No
Denger et al.[16]	Pattern	FR (Embedded Systems)	No	Yes	No
Konrad and Cheng [35]	CNL	NFR (Automotive)	No	Yes	Yes
Fuchs et al. [25]	CNL	Both (Several)	No	No	Yes
Post et al. [54]	CNL	FR (Automotive)	No	Yes	Yes
Crapo et al. [15]	CNL	FR (Domain - Independent)	No	No	Yes
Carvalho et al. [13]	CNL	Both (Data - Flow Reactive systems)	No	No	Yes
Autili et al. [8]	CNL				

Patterns

Pohl and Rupp [52] discuss a single pattern to specify functional requirements. The authors claim that the requirements that comply to this pattern are explicit, complete and provide the necessary details to test such requirements.

Mavin et al. [44] define the Easy Approach to Requirements Syntax (EARS), which is a set of five patterns enabling analysts to describe system functions. The authors demonstrate, through a case study in the aviation domain, that using EARS leads to requirements which are easier to understand and which exhibit fewer quality problems, particularly in relation to ambiguity. Tool support for the EARS patterns was presented in a follow-up paper [39].

Withall [74] identifies 37 patterns to specify structured functional and non-functional requirements for the business domain. The study provides insights regarding the creation and extension of the patterns.

Riaz et al. [55] define a set of 19 functional security patterns. They provide a tool that assists the user in selecting the appropriate pattern based on the security information identified in the requirements.

Eckhardt et al. [19] propose patterns to specify performance requirements. The patterns were derived from a content model built from an existing performance classification. Eckhardt et al. [19] define the content elements that a performance requirement must contain to be considered complete.

Denger et al. [16] propose a set of patterns to describe requirements for embedded systems. The patterns were derived from a metamodel that captures several types of embedded-system requirements. The authors validate their patterns through a case study.

In contrast to the other four studies, Riaz et al. [55] and Mavin et al. [44] provide tool support to guide analysts in defining requirements. Eckhardt et al. [19] follow a systematic process to develop a framework for the creation of performance requirements patterns, and presented a well-defined evaluation of their approach.

Controlled Natural Languages

Konrad and Cheng [35] provide a restricted natural language for the automotive and appliance domains, enabling analysts to express precise qualitative and real-time properties of systems. They evaluated their approach through a case study and introduced their tool in a follow-up paper [34]. In recent work, Autili et al. [8] extended the language proposed by Konrad and Cheng [35], including 40 new patterns that allow users to specify real-world system properties.

The approach described by Fuchs et al. [25] was identified from the survey and classification of CNLs conducted by Kuhn [36].

Fuchs et al. [25] propose the Attempto Controlled English, which is a CNL that defines a subset of the English language intended to be used in different domains, such as software specification and the Semantic Web. Attempto can be automatically translated into first-order logic.

Post et al. [54] identify three new rules that extend the approach proposed by Konrad and Cheng [35] to express requirements in the automotive domain. They validated their rules through a case study, and described their tool in another paper [53].

Crapo et al. [15] propose the Semantic Application Design Requirements Language which is a controlled natural language in English for writing functional requirements. Their language supports the mapping to first-order logic. Carvalho et al. [13] propose a CNL called SysReq-CNL that allows analysts

to describe data-flow requirements. Their sentence rules are nonetheless not mapped onto any formal semantics. None of the above approaches have been empirically evaluated.

To summarize, no previous strand of work describes a systematic process to build CNL grammar rules. However, all the above approaches provide tool support to assist analysts with specifying requirements.

Differences Between the Related Work and Our Approach

No other work, in our knowledge, follows a systematic process for creating and evaluating a CNL to specify functional requirements, either in the financial domain (the main focus of our investigation) or any other domain. More precisely, our work differs from the existing work in the following respects: (a) we derive Rimay from the analysis of a large and significant number of requirements from the financial domain; (b) we create Rimay by following a rigorous and systematic process; (c) we evaluate Rimay through a case study based on industrial data while following empirical guidelines for conducting Case Study Research [56]; and (d) we fully operationalize Rimay through a usable prototype tool.

3.3 Qualitative Study

In this section, we report on a qualitative study aimed at characterizing the information content found in the functional NL requirements provided by Clearstream. In the following, every time we speak of "requirements", we mean functional NL requirements.

Other techniques, such as grammar induction [66], could have been used to learn the syntax of the functional requirements in an automated manner. However, we believe that the limited number of available requirements would not have resulted in a reliable learning model. Therefore, we opted to conduct a qualitative study to build a semi-automated strategy enabling the creation of the grammar rules in a precise manner.

First, we describe the context of the qualitative study along with the criteria used to select SRSs. Then, we present the analysis procedure of our qualitative study where we show the codes that identify different groups of requirements. Each group of requirements is characterized by different information content. In this work, information content refers to the meaning assigned to the text of the requirements.

The result of the analysis procedure is a grammar that defines the syntax of a CNL that is able to specify all the information content found in the analyzed requirements. A grammar is a set of controlled and structured syntax rules (also known as grammar rules) describing the form of the elements that are valid according to the language syntax [10]. In our context, our grammar controls the structure of functional requirements by applying syntax rules. Section 3.3.3 (Step 5.2) describes how we produce the Rimay grammar rules, and Section 3.4 describes all the grammar rules of Rimay.

3.3.1 Research Question

The goal of this qualitative study is to answer the following research question: ***RQ1: What information content should one account for in the requirements for financial applications?*** RQ1 aims to identify the mandatory and optional information content used by Clearstream to describe requirements. This is essential in order to design a CNL that will help financial analysts write requirements that are as complete and as unambiguous as possible.

3.3.2 Study Context and Data Selection

We conducted this study in collaboration with Clearstream Services SA Luxembourg, which is a securities services company with 2500 customers in 110 countries. More concretely, we worked with the Investment Fund Services (IFS) division. An Investment Fund is a capital that belongs to a number of investors and is used to collectively invest in stocks and bonds. Among other tasks, the IFS division takes care of (a) the development of new applications, (b) upgrading existing ones, and (c) the migration of applications to more sophisticated technologies to provide their clients with state-of-the-art solutions that comply with the regulations in force. The Clearstream units involved with IFS are project management, IFS and market operations, design, functional and business analysis, development, and testing.

Clearstream performs the aforementioned tasks following a methodology grounded in best practices and years of experience. For instance, financial analysts specify requirements using a combination of UML models and natural language requirements following the Rupp template [52]. Clearstream follows a carefully planned software development process [64] based on the V-Model, that is suitable for a heavily regulated industry, such as finance.

Clearstream is continuously delivering new software projects in the financial domain and employs English as the primary language for specifying requirements. Two members of our research team were embedded in the Clearstream - IFS to get familiar with the company's development process and its organizational culture for over a month before starting the project described in this chapter. Our members participated in training sessions and numerous meetings organized by Clearstream. Additionally, all the research team members have been interacting, both electronically and through face-to-face meetings, with the members of the IFS team for two years.

We validated our results and conclusions with a team of experts. The team was composed of eight financial analysts: (a) two were *senior* financial analysts with more than 20 years of experience in specifying requirements in the financial domain. Their areas of expertise are business analysis, functional design, functional architecture, requirements engineering, and project management; (b) Four of them were *mid-career* financial analysts with more than 10 (but less than 20) years of experience in business and functional analysis in the financial domain. One of the *mid-career* analyst had software programming and testing skills; and (c) two were *junior* financial analysts with two to five years of experience in business analysis. This validation activity was performed over a year in an iterative and incremental manner with face-to-face, bi-weekly sessions with the team of experts, with each of these sessions lasting between two to three hours. This activity was concluded when the experts did not have any additional suggestions for improving the clarity, completeness, or correctness of the requirements.

Among all those available in Clearstream, we selected SRSs which: (a) belong to recently concluded projects, (b) contain at least 15 requirements, (c) contain requirements written in English, and (d) are written by different financial analysts. The senior financial analysts from Clearstream selected 11 representative SRSs according to the four criteria defined above. Each one of the SRSs contained the following types of information: business context, goals and objectives, project scope, current and future overview, general information (e.g., glossary, related documentation, acronyms and abbreviations), and Unified Modeling Language (UML) diagrams for the high-level functional decomposition of the systems and requirements. In total, the 11 SRSs contained 2755 requirements.

3.3.3 Analysis Procedure

Figure 3.1 shows an overview of our semi-automated analysis procedure. In Step 1, we first extracted 2755 requirements from 11 SRSs. In Step 2, we identified a dictionary of 41 codes from the extracted requirements. For example, the code *send_11.1* identifies five verbs used in the extracted requirements: “return", “send", “forward", “pass", “export" and “import"(Table 3.4 and Table 3.5 shows the 41 codes and verbs identified in our qualitative study and the evaluation). Our analysis procedure for identifying the codes followed *protocol coding* [58], which is a method for collecting qualitative data according to a pre-established theory, i.e., a set of codes. As explained later in this section, our pre-established set of codes was identified from VerbNet. Using a coding system based on a predefined set of codes helps us to save analysis time and mitigate coding bias. In Step 3, two annotators (first and second researchers) labeled the extracted requirements with one or more of the codes discovered in the previous step. In Step 4, we grouped the extracted requirements by their labels. The purpose of grouping requirements is to ease the identification of common information content to create grammar rules. For example, all the requirements that use the verbs members of the code *send_11.1* share the semantic roles `INITIAL LOCATION` (a place where an event begins or a state becomes true) and `DESTINATION` (a place that is the end point of an action and exists independently of the event). In Step 5, we iteratively created and integrated the grammar rules into Rimay. Each of the five steps in Figure 3.1 shows one or two icons denoting whether a given step was carried out (1) automatically (i.e., the three gears icon), (2) manually (i.e., the human icon), or (3) semi-automatically (i.e., both icons). The next subsections describe in details Steps 1 to 5.

Extract Requirements (Step 1)

We read the 11 SRSs and extracted 2755 requirements. In our case, all the requirements were written in tables in which all the requirements were clearly identified and distinguished from other information. The structure of the SRSs clearly separates functional from non-functional requirements. Furthermore, we checked that no functional requirement was mistakenly placed in the non-functional requirements section. We verified that the content of the requirements presenting lists and tables was correctly captured by our automatic extraction algorithm. If there was any error, we manually corrected it. This step was automated using the Apache POI API [4], which is a well-known Java library for reading and writing files in Microsoft Office formats.

Table 3.2 shows three requirements extracted from a SRS. The column “Id” identifies the requirements, the column “Description" contains the original text of the requirements, and the column “Rationale" presents the reasoning behind the creation of a given requirement.

Identify Codes (Step 2)

The coding approach is intended to (1) obtain a number of codes that allow the language to be expressive enough for the financial domain, (2) be systematic to allow others to replicate the procedure, and (3) ensure that Rimay remains as broadly applicable as possible by minimizing reliance on domain-specific terms. The requirements specify the expected system behavior using verb phrases, e.g., “*send* a message" and “*create* an instruction". We used the verb lexicon named VerbNet (Section 2.2) to identify the codes from

[4] `https://poi.apache.org/` (last access on 17 August 2022)

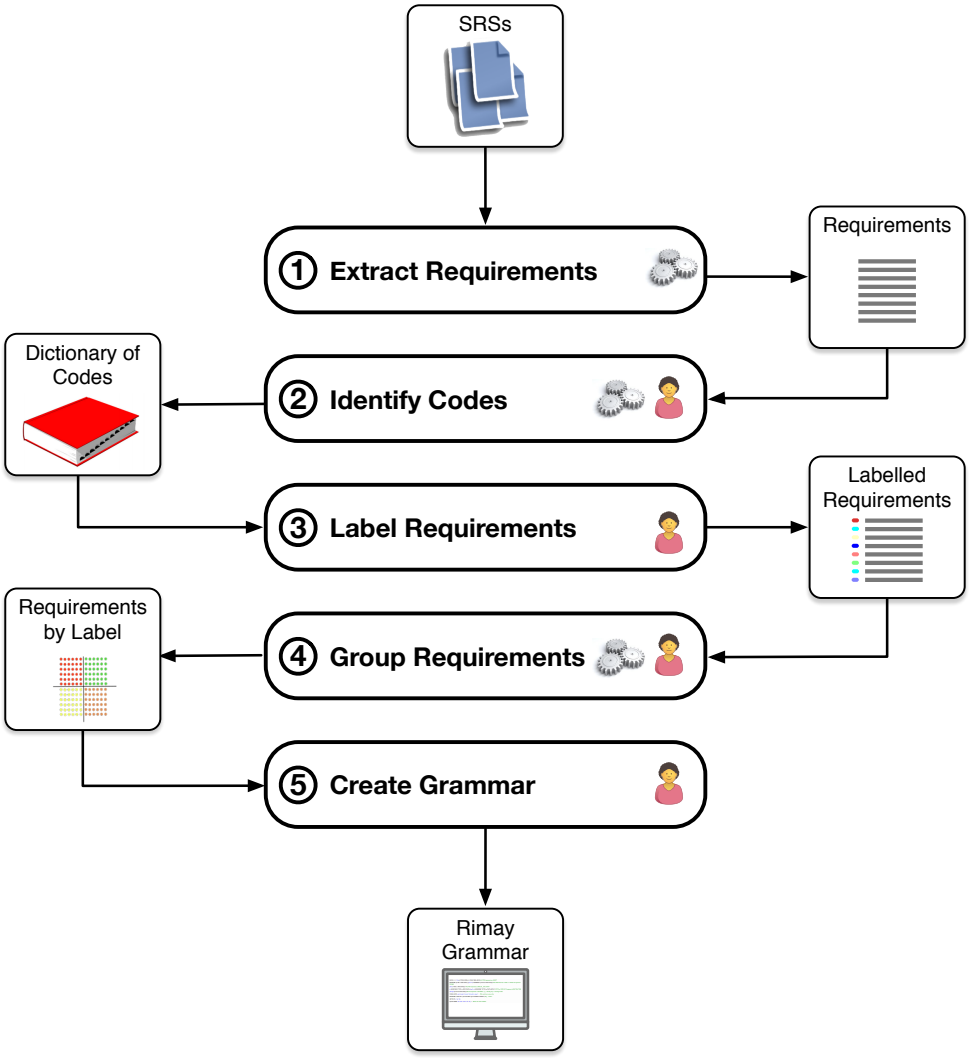


Figure 3.1: Overview of our analysis procedure

Table 3.2: Three requirements extracted from a SRS during Step 1 of Figure 3.1

Id	Description	Rationale
TNG.INPUT.010	If the message contains “FISN", then the System must ignore the message.	FISN is an official ISO Standard created to enhance the quality of financial messaging.
TRAN.0030	The System must regenerate the outbound XML according to the new XML specification “SR2017".	The previously created orders, which their status are activated, must be changed to comply with the new XML specification.
Data.SAA.060	The data of the System older than 13 months must be archived for at least 10 years.	This requirement complies to a legal rule.

our SRSs. Subsection 3.3.3 will explain in details how, by using verb classes, we obtain the grammar rules of Rimay.

We followed a semi-automated process to identify codes and their corresponding verbs. We automated some of the sub-steps of Step 2 by using the NLTK[5] library for Python. In the remainder of this section, we describe in detail which sub-steps of Step 2 were automated. From the 41 codes that we proposed in this qualitative study, 32 codes (78%) correspond to verb class ids from VerbNet (referred to thereafter as *VerbNet codes*), and nine (22%) are codes that we proposed because they were missing from VerbNet but were needed to analyze the requirements. We use below the following terms to describe this process:

- `REQS`: Set of requirements to analyze.
- `LEMMAS`: List of lemmas found in the action phrases of `REQS`.
- `CODES`: Dictionary of codes and their corresponding verb members found during our analysis procedure. There are two types of codes: VerbNet codes and codes proposed by us.
- `AUX`: Auxiliary list of the lemmas that are not members of any code in `CODES`.
- `SYNS`: Dictionary of lemmas and their corresponding applicable synonyms.
- `VN`: Read-only dictionary of all the publicly available VerbNet codes and their corresponding verb members.

In Figure 3.2, we show a running example of our process to identify the codes. The process steps are as follows:

Extract lemmas (Step 2.1). We extracted the verbs of each requirement in `REQS` (upper-left corner of Figure 3.2) to obtain lemmas. A lemma is the base form of the verb. For example, from “archived", the lemma is “archive". We stored the resulting lemmas in `LEMMAS`.

Separate lemmas that do not belong to any VerbNet code (Step 2.2). We retrieved for every lemma in `LEMMAS` its corresponding VerbNet codes from `VN`. We stored these VerbNet codes and their corresponding lemmas (including their sense number, depicted as a number after the symbol #) in `CODES`. For example, the key-value pair *{engender-27, generate#1}* in `CODES` of Figure 3.2 (Step 2.2) means that the lemma *generate* (Step 2.1 of Figure 3.2) with the sense number one (i.e., “bring into existence") is a member of the VerbNet code *engender-27*.

If a lemma in `LEMMAS` was not a member of any VerbNet code in `VN`, we added it to an auxiliary list of lemmas named `AUX`. For example, in Figure 3.2 (Step 2.2) we added to `AUX` the lemmas *ignore*, *regenerate* and *synchronize* that were not identified in `VN`, but were found in the analyzed requirements.

Identify new VerbNet codes by using synonyms (Step 2.3). We analyzed the synonyms and senses of the lemmas in `AUX` to discover new VerbNet codes that can be added to `CODES`. We describe this process in more details as follows:

[5] `https://www.nltk.org/` (last access on 17 August 2022)

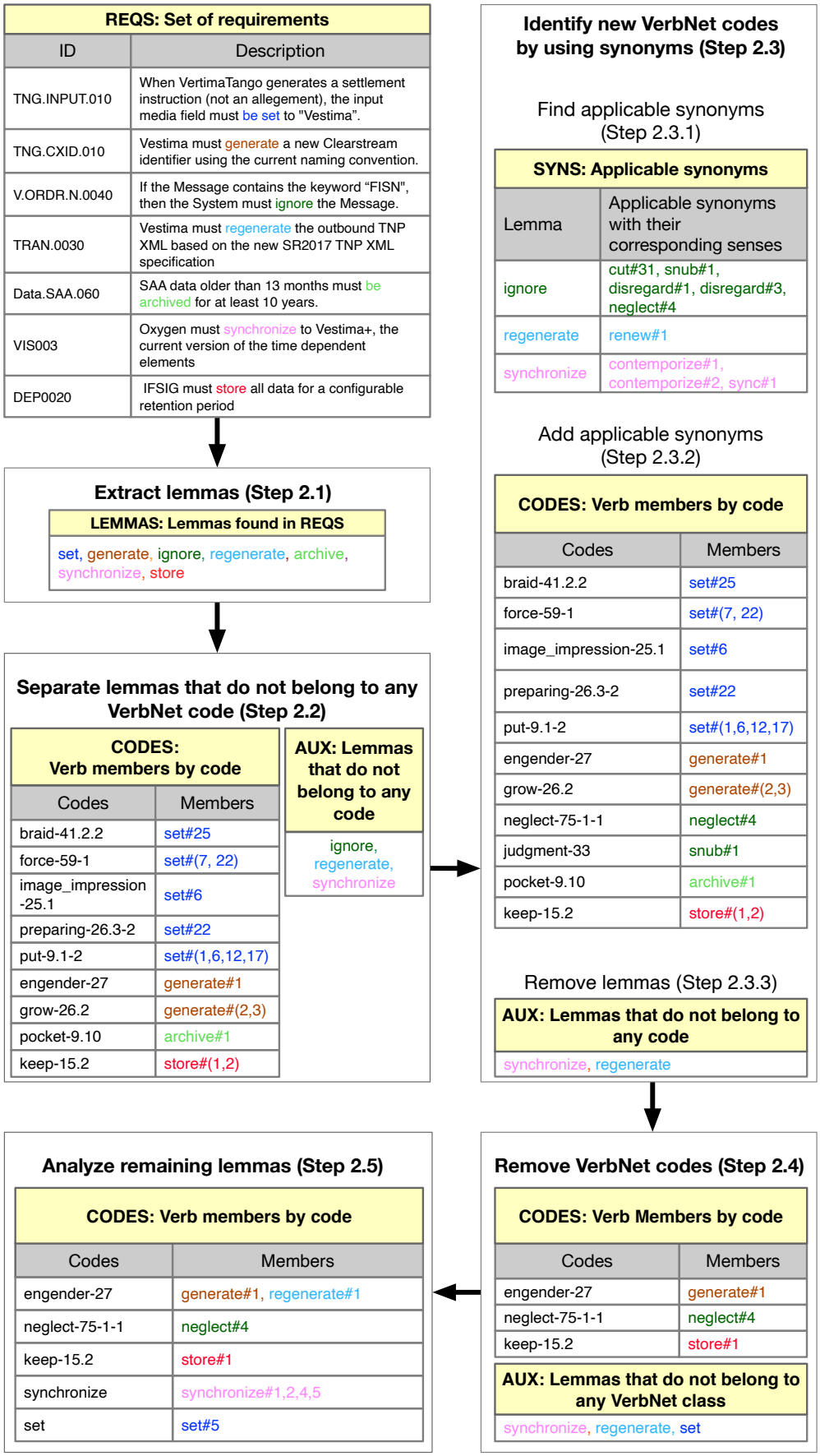


Figure 3.2: Identify codes (Step 2)

Find applicable synonyms (Step 2.3.1). We used WordNet (Section 2.2) to retrieve all the synonyms of each auxiliary lemma in AUX. We stored in SYNS only the synonyms whose senses match the sense of an auxiliary lemma as used in REQS.

As an example, Table 3.3 shows the list of synonyms of the lemma *regenerate*, which is one of the lemmas in AUX shown in Figure 3.2 (Step 2.2). The synonyms in Table 3.3 are grouped according to the sense numbers of the lemma *regenerate*, namely 1, 3, 4 and 9 (according to WordNet, the verb *regenerate* has nine senses, but Table 3.3 only shows the senses that have at least one synonym). From the four senses in Table 3.3, we chose the ones that match the sense of the verb *regenerate* used in REQS. In this case, we chose sense number 1 since it was the only sense that was applicable to the requirements. Finally, we store in SYNS the synonyms and their chosen sense numbers. In the case of the lemma *regenerate*, we only added *renew#1* to SYNS.

Table 3.3: Senses and synonyms of the verb *regenerate* retrieved from WordNet.

Sense Number	Sense Definition	Synonyms and Their Sense Number	Chosen Sense?
1	Reestablish on a new, usually improved, basis or make new or like new	renew#1	Yes
3	Bring, lead, or force to abandon a wrong or evil course of life, conduct, and adopt a right one	reform#2, reclaim#3, rectify#3	No
4	Return to life, get or give new life or energy	restore#2, rejuvenate#4	No
9	Restore strength	revitalize#1	No

Add applicable synonyms (Step 2.3.2). We retrieved, for every synonym in SYNS, its corresponding VerbNet codes from VN. Then, we stored the retrieved VerbNet codes and the corresponding synonym (including the sense number) in CODES. For example, given that the synonym *neglect* (Step 2.3.1 of Figure 3.2) with sense number four (i.e., *neglect#4*) is a member of the VerbNet code *neglect-75-1-1*, we created the key-value pair *{neglect-75-1-1, neglect#4}* in CODES (Step 2.3.2 of Figure 3.2). If none of the synonyms of a lemma is a member of any code in VN, then we move the lemma from SYNS to AUX. For example, if the synonym is *renew#1* and it is not a member of any VerbNet code in VN, if it is a synonym of *regenerate* we then move *regenerate* from SYNS to AUX.

Remove lemmas (Step 2.3.3). We updated AUX by removing the lemmas whose synonyms were added to CODES in Step 2.3.2. In Figure 3.2, we removed the lemma *ignore* from AUX. Furthermore, the verb *synchronize* was the only verb whose synonyms were not members of any VerbNet verb class in CODES. Therefore, the verb *synchronize* remained in AUX.

Remove VerbNet codes (Step 2.4). In this step, our goal is to remove the VerbNet codes (from CODES) that are either not relevant to the SRSs in the financial domain or redundant. We performed this step during several offline validation sessions. Each session was attended by three to four financial analysts with the presence of at least one senior and one mid-career financial analyst.

At the end of Step 2.4 (Figure 3.2), we went from 11 to three VerbNet codes (i.e., a reduction of 72,7%). Considering all the VerbNet codes used during this qualitative study, not only the 11 VerbNet codes shown in Step 2.4 in Figure 3.2, we decreased the number of VerbNet codes from 158 to 32 (i.e., a reduction of 79,7%). The two strategies that we employed to reduce VerbNet codes are as follows:

- *Strategy 1. Discard redundant verbs.* For example, between the verbs *archive* and *store*, we discard the verb *archive* because the verb *store* is more frequent and both verbs are semantically similar.

- *Strategy 2. Discard verbs that do not have applicable senses.* For example, the VerbNet code *image_impression-25.1* (Step 2.3.2 of Figure 3.2) involves only the member *set#6* whose sense is defined by WordNet as: “a relatively permanent inclination to react in a particular way". Since this latter sense is not used in `REQS`, we finally discarded *image_impression-25.1* from `CODES`. After applying this strategy, if a verb was discarded from `CODES`, we added only its lemma to `AUX` for further manual analysis as we explain next in Step 2.5. For example, given that the verb *set* was discarded from `CODES`, we added its lemma (e.g., only the word *set* without sense#) to `AUX`.

Analyze remaining lemmas (Step 2.5). In this step, we manually checked in WordNet if the senses of the remaining lemmas in `AUX` could be included in `CODES`. This step was carried out with the help of two senior and two mid-career financial analysts from Clearstream. We updated `CODES` when we identified an appropriate sense in WordNet that referred to one of the remaining lemmas. For example, in Figure 3.2, we created the code *set* with a member *set#5* whose sense is used in `REQS`, and updated the VerbNet code *engender-27* with the member *regenerate#1*.

Coding results. Tables 3.4 and 3.5 present the resulting codes identified during our qualitative study described in Section 3.3.3 (“Identify Codes" (Step 2)). We finally obtained 41 codes, where 32 were obtained from VerbNet and nine were proposed by us.

Table 3.4 provides the 32 VerbNet codes and their members. The first column of the table lists the codes, where each code is composed of a class name and a hierarchy level (Section 2.2). Note that the class names are not always verbs, e.g., *reflexive appearance*. The second column shows the verb members related to the code. Table 3.5 shows the nine codes that we proposed. The first column of the table lists the codes and the second column provides the verb members associated to the code.

Label Requirements (Step 3)

In Step 3 (Figure 3.1), two annotators (the first and second researchers) manually labeled the requirements extracted in Step 1 with one or more of the codes identified in Step 2. The labeling process required to (a) read the requirements and identify the verbs used in the system response of the requirements, (b) attempt to match the identified verbs with members of the codes found in Step 2, and (c) when there is a match, label the requirement with the corresponding code. This task required expert knowledge to abstract the main action verbs of the requirement and assign the correct code(s) to it. Because this activity can be challenging due to the polysemy of the main action verb, it was conducted by both annotators. We divided the set of 2755 requirements, used in our qualitative study, into two equal parts. All the requirements of the first part were annotated by the first annotator and reviewed by the second annotator

Table 3.4: VerbNet codes identified during our qualitative study

Codes		**Members**
Class name	**Hierarchy Level**	
admit	65	exclude
advise	37.9-1	instruct
allow	64.1	allow, authorize
beg	58.2	request
begin	55.1-1	begin
concealment	16-1	hide
contribute	13.2	restore
create	26.4	compute, publish
enforce	63	enforce
engender	27	create, generate
exchange	13.6	replace
forbid	67	prevent
herd	47.5.2	aggregate
involve	107	include
keep	15.2	store
limit	76	limit, restrict, reduce
mix	22.1-2	add
mix	22.1-2-1	link
neglect	75-1-1	neglect, ignore
obtain	13.5.2	accept, receive, retrieve
other_cos	45.4	close
put	9.1	insert
reflexive appearance	48.1.2	display, show
remove	10.1	extract, remove, delete
say	37.7-1	report, propose
see	30.1-1	detect
send	11.1	return, send, forward, pass
shake	22.3-2-1	concatenate
throw	17.1	discard
transcribe	25.4	copy
turn	26.6.1	convert, change, transform
use	105	apply
Total: 32		

Table 3.5: Codes proposed during the qualitative study

Codes	Members
cancel	cancel
enable disable	enable, disable
get from	download
interrupt	interrupt
migrate	migrate
select unselect	select, unselect
synchronize	synchronize
update	update
validate	validate, check
Total: 9	

and vice versa. If there was disagreement between annotators, we consulted a financial analyst to reach an agreement using a consensus-based decision-making strategy [12].

We describe below the three activities of the labeling process for requirement DEP0020 in `REQS` shown in Figure 3.2:“IFSIG must store all data for a configurable retention period". Specifically, (a) we identified that the verb used in the system response is *store*, (b) we detected that *store* matches one of the members of the VerbNet code *keep-15.2*, and (c) we labeled the requirement with the VerbNet code *keep-15.2*.

Group Requirements (Step 4)

In Step 4 (Figure 3.1), we grouped and copied the labeled requirements to different spreadsheets based on their labels. The purpose of having the requirements grouped by label is to make it easier for us to identify common information content among them.

Create Grammar (Step 5)

In Step 5 (Figure 3.1) we created the grammar of Rimay to capture relevant information content from the requirements. Figure 3.3 shows the steps that we carried out to create grammar rules for the VerbNet code *Send 11.1* (Table 3.4). The box in the upper-right corner of Figure 3.3 shows four examples of requirements related to the VerbNet code *Send 11.1* that will be used to illustrate this step. The same sub-steps (i.e., from 5.1 to 5.6) were carried out for the rest of the codes presented in Table 3.4 and Table 3.5.

Identify content in the requirements (Step 5.1). In this step we identify semantic roles and keywords in the requirements. VerbNet provides the syntax and the examples that show most of the semantic roles and the keywords (e.g., the prepositions) related to the VerbNet codes in Table 3.4. For example, the box in the upper-left corner of Figure 3.3 shows the syntax and examples related to the VerbNet code *Send 11.1*. The syntax contains the prepositions `from` and `to`, and the semantic roles `AGENT` (a participant that initiates an action), `THEME` (an entity which is moved by an action, or whose location is described), `INITIAL_LOCATION` (a place where an event begins or a state becomes true) and `DESTINATION` (a place that is the end point of an action and exists independently of the event).

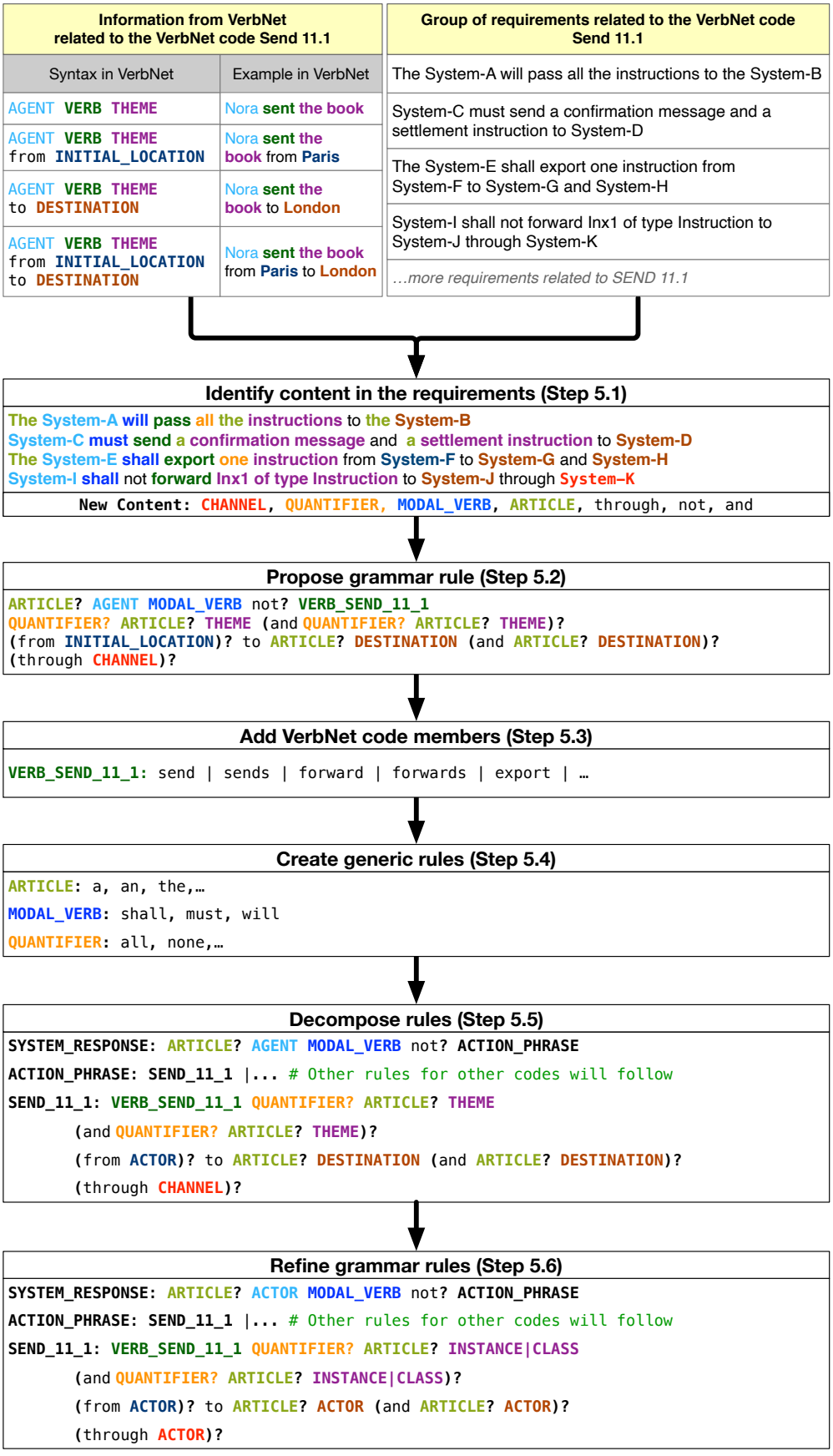


Figure 3.3: Obtaining CNL grammar rules from requirements related to the VerbNet code *Send 11.1*

In Figure 3.3, we use different colors to show the correspondence between the semantic roles and the parts of the requirements that represent the semantic roles. When some content in the requirements was not related to any VerbNet semantic role, we proposed a new semantic role to identify that content. For example, in Step 5.1 of Figure 3.3, we proposed the new semantic role `CHANNEL` to identify the content in the phrase “through System-K".

Propose grammar rule (Step 5.2). Based on the syntax provided by VerbNet, we defined the order of appearance of the content, and its repetition in Rimay. The symbols `?`, `*` and + indicate that the users of Rimay can repeat what is before the symbol at most once, any number of times, and at least once, respectively. Step 5.2 in Figure 3.3 shows that the grammar rule for the VerbNet code *Send 11.1* contains keywords such as (i) connectors (*and* and *or*), (ii) prepositions shown in the VerbNet syntax (*from* and *to*), (iii) prepositions related to new content (*through*) and (iv) the negation of a modal verb (*not*).

Add VerbNet code members (Step 5.3). We added a complete list of all the members of each VerbNet code related to its corresponding rule. For example, *forward* and *send* are two of the members of the VerbNet code *Send 11.1* that we added to its corresponding rule `VERB_SEND_11_1`. We also added the conjugated forms of the verbs to the rule (e.g., *forwards*, *sends*).

Create generic rules (Step 5.4). We created the rules related to the generic English grammar, e.g., we created the rules `ARTICLE`, `MODAL_VERB`, and `QUANTIFIER`.

Decompose rules (Step 5.5). We decomposed the grammar rules created in Step 5.2 to make them easier to understand and reuse. For example, we decomposed the example rule in Step 5.2 into three rules: `SYSTEM_RESPONSE`, `ACTION_PHRASE`, and `SEND_11_1`.

Refine grammar rules (Step 5.6). With the help of four financial analysts (including one senior and one mid-career financial analyst), we replaced some of the semantic role names with other ones that were more familiar to both financial analysts and engineers. In our case, financial analysts and engineers working for Clearstream were familiar with the UML [49]. For example, in the grammar rules `SYSTEM_RESPONSE` and `SEND_11_1` (Step 5.4 in Figure 3.3), we chose to replace the role `AGENT` with `ACTOR`, because an agent can be represented as an UML actor, i.e., a role played by a human user or a system who initiates and carries out an event or action.

Method. The method that we used to create Rimay was iterative and incremental. This means that we first followed Steps 5.1 to 5.6 in Figure 3.3 to create the grammar rules related to one of the groups of requirements produced in Step 4 of Figure 3.1. Second, we generated a requirements editor using Xtext. Third, we used the generated editor to rephrase the requirements in the first requirements group to test the grammar and its corresponding editor. We tested that our grammar and the editor were expressive enough to allow us to write all the information content for the first group of requirements. If the grammar was not expressive enough, we analyzed and extended the grammar, regenerated the editor and verified the requirements until there were no errors in all the rephrased requirements. For each remaining requirements groups produced in Step 4 (Figure 3.1), we repeated Steps 5.1 to 5.6 as performed for the first requirements group.

Answer to RQ1: Following a systematic and repeatable process, we identified 41 codes, which in our context, are groups of verbs that convey the same information in NL requirements. We created grammar rules for all the codes identified, thus covering all the information content found in a large and representative set of functional requirements in financial applications. We anticipate that our approach, being general in nature, should be applicable to other domains as well, while acknowledging that more empirical investigation is necessary to conclusively validate this claim.

3.4 Controlled Natural Language for Functional Requirements

In this section, we describe how a requirement is structured in Rimay in order to answer ***RQ2: "Considering the stakeholders, how can we represent the information content of requirements for financial applications?"***.

In recent years, different patterns have been increasingly used by the industry to improve the quality of the requirements. Patterns like EARS [44] and Rupp [52] provide general constructs and concepts to specify requirements (Section 3.2). However, these templates are not amenable to the type of analyses enabling task automation because they allow the introduction of unstructured text. On the other hand, CNLs provide structures with more specialized concepts and constructs, enabling automated analysis. As we report in our recent work, Rimay enables the generation of abstract test cases (Chapter 5). Since we could not find any comparable work in the financial domain, we applied Grounded Theory analysis for building Rimay. However, as we explain below, some constructs and concepts of Rimay are inspired by the EARS template.

The rule `REQUIREMENT` shown in Listing 3.1 provides the overall syntax for a requirement in Rimay. The rule shows that the presence of the `SCOPE` and `CONDITION_STRUCTURES` is optional, but the presence of an `ACTOR`, `MODAL_VERB` and a `SYSTEM_RESPONSE` is mandatory in all requirements.

```
REQUIREMENT: SCOPE? CONDITION_STRUCTURES? ARTICLE? ACTOR MODAL_VERB not?  SYSTEM_RESPONSE.
CONDITION_STRUCTURES: CONDITION_STRUCTURE (,? (and|or) CONDITION_STRUCTURE)*, then?
```

Listing 3.1: Overall syntax of Rimay

In a requirement, an actor is expected to achieve a system response if some conditions are true. An actor is a role played by an entity that interacts with the system by exchanging signals, data or information [49]. Moreover, requirements written in Rimay may have a scope to delimit the effects of the system response. One example of a requirement in Rimay is: "`For all the depositories, System-A must create a MT530 transaction processing command`". The requirement has a scope (`For all the depositories`), does not have any conditions, and has an actor (`System-A`) and a system response (`create a MT530 transaction processing command`).

Throughout this section, we simplify the description of Rimay by considering that the keywords are not case-sensitive. Also, we use grammar rules that are common in English such as `MODAL_VERB` (e.g., shall, must) and `MODIFIER` that includes articles (e.g., a, an, the) and quantifiers (e.g., all, none, only one, any). Subsections 3.4.1 and 3.4.3 will explain the `CONDITION_STRUCTURES` and `SYSTEM_RESPONSE`, respectively.

3.4.1 Condition Structures

The grammar rule named CONDITION_STRUCTURE shown in Listing 3.2 defines different ways to use system states, triggering events, and features, to express conditions that must hold for the system responses to be triggered.

```
CONDITION_STRUCTURE: WHILE_STRUCTURE|WHEN_STRUCTURE|WHERE_STRUCTURE|IF_STRUCTURE|
    TEMPORAL_STRUCTURE
WHILE_STRUCTURE: While PRECONDITION_STRUCTURE
WHEN_STRUCTURE: When TRIGGER
WHERE_STRUCTURE: Where TEXT #TEXT is a feature expression
IF_STRUCTURE: If PRECONDITION_STRUCTURE|TRIGGER
TEMPORAL_STRUCTURE: (Before|After) TRIGGER
```

Listing 3.2: Condition structures

The condition structures WHILE, WHEN, WHERE and IF that we use in our grammar are inspired by the EARS template [44]. EARS is considered by practitioners as beneficial due to the low training overhead and the quality and readability of the resultant requirements [43]. Additionally, we proposed the rule TEMPORAL_STRUCTURE to be used when the system responses are triggered before or after a triggering event. Below, we describe the types of CONDITION_STRUCTURE used in Rimay:

- The WHILE_STRUCTURE is used for system responses that are triggered while the system is in one or more specific states.
- The WHEN_STRUCTURE is used when a specific triggering event is detected at the system boundary.
- The WHERE_STRUCTURE is used for system responses that are triggered only when a system includes particular features. In our context, a feature is a unit of the functionality of the system [4]. The features are described in free form using the rule TEXT.
- The IF_STRUCTURE is used when a specific triggering event happens or a system state should be hold at the system boundary before triggering any system responses.

The rule CONDITION_STRUCTURE shown in Listing 3.2 allows combining condition structures using logical operators. We can, for example, combine the IF and WHEN structures using the operator and in the structure "If PRECONDITION_STRUCTURE and when TRIGGER" to separate the conditions in which the requirement can be invoked (i.e., the preconditions) and the event that initiates the requirement (i.e., the trigger).

Figure 3.4 depicts examples of the WHEN_STRUCTURE, TEMPORAL_STRUCTURE, and IF__STRUCTURE.

Listing 3.3 shows the grammar rules TRIGGER and PRECONDITION_STRUCTURE referenced in Listing 3.2.

The rule TRIGGER in Listing 3.3 defines that a triggering event is always caused by an ACTOR that performs some actions. The actions performed by the actor are defined by the rule ACTIONS_EXPRESSION which enables the combination of any number of actions using logic connectors to express complex system events. The WHEN_STRUCTURE in Figure 3.4 shows an example of a trigger composed of an actor and an action expression: "System-B receives an email alert from System-A".

```
TRIGGER: MODIFIER? ACTOR ACTIONS_EXPRESSION
ACTIONS_EXPRESSION: ACTION ((,|,and|and) ACTION)+
ACTION: ((do|does) not )? ACTION_PHRASE
PRECONDITION_STRUCTURE: ITEMIZED_CONDITIONS | CONDITIONS_EXPRESSION
ITEMIZED_CONDITIONS: the following conditions are satisfied:
        HYPHEN CONDITION((,|,and|and) HYPHEN CONDITION)+
CONDITIONS_EXPRESSION: CONDITION((,|, and|, or|and|or)? CONDITION)+
```

Listing 3.3: Trigger and precondition structure

The rule PRECONDITION_STRUCTURE in Listing 3.3 gives freedom for the users to decide how to describe conditions. The rule ITEMIZED_CONDITIONS (Listing 3.3) is appropriate for writing long lists of conditions that must evaluate to True. Conversely, the rule CONDITIONS_EXPRESSION (Listing 3.3) is suitable for only one condition, multiple conditions combined with logical operators, or parentheses that denote priority in the evaluation order of operations. The IF_STRUCTURE in Figure 3.4 shows examples of non-itemized and itemized conditions.

3.4.2 Conditions

In the previous subsection, we introduced the rule PRECONDITION_STRUCTURE to specify conditions. This rule is composed of operands and operators which are described as follows.

Operands

The operands are represented by the rules ACTOR, CLASS, PROPERTY, INSTANCE, ELEMENT and TEXT. The meaning of the operands is the same as in the UML [49], therefore an *Actor* specifies a role played by the user or another system that interacts with our system. The *Class* represents a domain concept (e.g., Instruction). A *Property* represents the attributes of the *Class*. An *Instance* represents a specific realization of a *Class* and an *Element* is a constituent of a model.

The users of Rimay can use the dot notation to refer to a property of a class, e.g.,"`Instruction .Settlement_Date`". In the cases where there is only one instance of a class in a requirement, the users do not need to declare any instance. For example, given that in Figure 3.4 there is only one instance of an instruction, we used "`Instruction`" instead of "`Inx1 of type Instruction`".

Operators

Rimay uses the following families of operators and its negative forms:

- COMPARE, such as "`equals to`", "`less or equal to`", etc.,
- CONTAINS such as "`has`", "`contains`", etc.,
- OTHER OPERATORS such as "`is available`"

An example of a condition that conforms to Rimay is: "`Inx1 of type Settlement_Instruction has Status and Status is equal to Valid`". This condition uses operators of type CONTAINS and COMPARE.

Condition Rule

The operators and operands defined in the previous subsections are used in the five grammar rules shown in Listing 3.4 conditions such as the ones shown in Figure 3.4.

The types of conditions are described as follows:

(1) `INSTANCE OR CLASS HAS PROPERTIES` evaluates if the instance of a class, or a class itself defines one or more specific properties. The properties can be defined in a document (e.g., "`Instruction has the properties described in the Section 1.b`"), or directly in the requirement (e.g., "`Instruction has the properties: Owner, Status and Settlement_Date`").

(2) `CONVENTION` checks if a property conforms to a format or standard, e.g., "`Instruction.Settlement_Date conforms to the standard ISO-8601`".

(3) `CLASS OR PROPERTY OPERATOR VALUE` is a condition composed of an operand-1, an operator and an operand-2. The operand-1 is a reference to a `CLASS` or `PROPERTY`. The auxiliary rule `OPERATOR VALUES EXPR` defines the operator and the operand-2 of the condition, e.g., "`the Transaction.Amount is less than or equal to 20000 Euros`". The operand-2 is any type of operand described in Section 3.4.2.

(4) `INSTANCE OR PROPERTY OPERATOR VALUE` is an operand-operator-value condition. The operand is a reference to an `INSTANCE` or `PROPERTY` and the value represent any literal or number. An example of this type of condition is: "`Transaction Type of Settlement Request is equal to Z-Value`".

(5) `UI COMPONENT INSTANCE OPERATOR ELEMENT` is a condition composed by an operand-1, operator, and operand-2 for a requirement related to the user interface (UI). The operand-1 is an instance of a UI component identified by a free form `TEXT` followed by a reference to the type of `UI COMPONENT`. Rimay contains a list of common UI component types to help the user to create the requirements (e.g., tab, page, bar, field, calendar, checkbox, menu, message). The auxiliary rule `OPERATOR VALUES EXPR` defines the operator and the operand-2 of the condition. An example that displays this type of condition is: "`the Account Number field contains 0000`".

```
INSTANCE_OR_CLASS_HAS_PROPERTIES: MODIFIER? (INSTANCE|CLASS) CONTAINS (the properties
    described in TEXT)|(the (property|properties): PROPERTIES)

CONVENTION: ((TEXT (of CLASS)? | PROPERTY) (conforms|conform|comply|complies) (to|with)
    MODIFIER? (format|convention|standard) TEXT

CLASS_OR_PROPERTY_OPERATOR_VALUE: MODIFIER? CLASS|PROPERTY OPERATOR_VALUES_EXPR

INSTANCE_OR_PROPERTY_OPERATOR_VALUE:    MODIFIER? TEXT OF_CLASS_OR_REFERENCE_TO_LABEL? Label?
    OPERATOR_VALUES_EXPR

UI_COMPONENT_INSTANCE_OPERATOR_ELEMENT: MODIFIER? TEXT UI_COMPONENT OPERATOR_VALUES_EXPR

OPERATOR_VALUES_EXPR: (COMPARE|CONTAINS|OTHER_OPERATORS) MODIFIER? MULTI_VALUES_EXPR TEXT?
```

Listing 3.4: Conditions rules

```
                                    TRIGGER
                          ACTOR     ACTIONS_EXPRESSION
WHEN STRUCTURE: When SystemB receives an "email alert" from SystemA
                                    TRIGGER
TEMPORAL STRUCTURE: Before SystemA sends an "Instruction" to SystemB

                                    CONDITIONS_EXPRESSION
                          CONDITION                          CONDITION
IF STRUCTURE: If Instruction.description contains a "Keyword" or Instruction.record is "Live" #CLASS_OR_PROPERTY_OPERATOR_VALUE
(NON ITEMIZED)


IF STRUCTURE: If the following conditions are satisfied:
(ITEMIZED)    HYPHEN                                                CONDITION
              -  the "Instruction" has the properties described in "Section Y", #INSTANCE_OR_CLASS_HAS_PROPERTIES
              -  the "Instruction" has the properties: "Owner, Status and Settlement Date", #INSTANCE_OR_CLASS_HAS_PROPERTIES
              -  the Instruction.Settlement_Date conforms to the standard "ISO8601", #CONVENTION
              -  the Transaction.Amount is less than or equal to "Y Value", #CLASS_OR_PROPERTY_OPERATOR_VALUE
              -  the "Transaction Type" of Settlement_Request is equal to "Z Value" and #INSTANCE_OR_PROPERTY_OPERATOR_VALUE
              -  the "Account Number" field  contains "0000" #UI_COMPONENT_INSTANCE_OPERATOR_ELEMENT

                        ACTOR          SYSTEM RESPONSE
ATOMIC SYSTEM       The User must upload the "excel file" to the "SystemA"
RESPONSE:
                        ACTOR                                  SYSTEM RESPONSE
SYSTEM RESPONSE     The SystemA must create an "Confirmation Message"(referred to as MsgI) and send MsgI to SystemB
EXPRESSION:
                        ACTOR
RESPONSE BLOCK      The SystemA must do the following actions in sequence:
ITEMIZED:           INT          SYSTEM RESPONSE
                    1  create an "Instruction"
                    2  send "Instruction" to SystemB
```

Figure 3.4: Examples of condition structures and system responses

3.4.3 System Response

The rule SYSTEM_RESPONSE in Listing 3.5 allows the user to express the behavior of the system in two manners using the rules: (a) RESPONSE_BLOCK_ITEMIZED, that is suitable for writing lists of actions; and (b) SYSTEM_RESPONSE_EXPRESSION, that is appropriate for writing one or multiple actions combined with logical operators, or parentheses that denote the priority of the actions. The previous rules include the rule ATOMIC_SYSTEM_RESPONSE and logical operators. Each ATOMIC_SYSTEM_RESPONSE contains an ACTION_PHRASE and optionally, a frequency (e.g., every 3 seconds). Figure 3.4 depicts examples of the ATOMIC_SYSTEM_RESPONSE as well as more complex examples, such as SYSTEM_RESPONSE_EXPRESSION and RESPONSE_BLOCK_ITEMIZED.

```
SYSTEM_RESPONSE: SYSTEM_RESPONSE_EXPRESSION | RESPONSE_BLOCK_ITEMIZED
SYSTEM_RESPONSE_EXPRESSION: ATOMIC_SYSTEM_RESPONSE ((,|, and|, or|and|or)?
        ATOMIC_SYSTEM_RESPONSE)*
ATOMIC_SYSTEM_RESPONSE: ACTION_PHRASE (every TEXT )?
RESPONSE_BLOCK_ITEMIZED: do the following actions (in sequence)? :
        BULLET ATOMIC_SYSTEM_RESPONSE ((,|, and|, or|and|or)? BULLET ATOMIC_SYSTEM_RESPONSE)*
```

Listing 3.5: System response

All the types of ACTION_PHRASE rules are available in Appendix A. The rule OBTAIN_13_5_2 in Table 3.6 is one type of ACTION_PHRASE rule. The column “Grammar Rule Name" shows the name of the grammar rule related to the code *obtain 13.5.2* that we discovered during the qualitative study (Tables 3.4 and 3.5). The column “Grammar Rule Summary" describes the syntax of OBTAIN_13_5_2, and the column “Examples" shows requirements that conform to that syntax.

Table 3.6: Grammar rule: OBTAIN_13_5_2

Grammar Rule Name	Grammar Rule Summary	Examples
OBTAIN_13_5_2	accept|receive|retrieve|reject MODIFIER? INSTANCE | CLASS (from ELEMENTS)? (through ACTORS)? (in compliance with TEXT (described in TEXT)?)?	Example 1: receive a DA_file from CFCL_IT Example 2: reject the "Message" in compliance with "current validation rules"

Rimay Editor

We developed the Rimay editor using the Xtext language engineering framework [10] which enables the development of textual domain-specific languages. We integrated the Rimay editor into an existing and widely known modeling and code-generation tool: Sparx Systems Enterprise Architect[6]. Enterprise Architect was already being used at Clearstream. In particular, we created a form composed of the Rimay editor, and fields related to key properties of a requirement, such as “Requirement ID", “Rationale", and “Examples". Figure 3.5 shows a screenshot of the form.

[6] https://sparxsystems.com/products/ea/ (last access on 26 January 2021)

To operationalize our technology-independent grammar (created in Step 5), we need to enhance it with some additional information. In particular, Xtext requires one to declare the name of the language, and further, import reusable terminals such as *INT*, *STRING* and *ID* for the syntax of integers, text, and identifiers, respectively.

The input that we provided to Xtext is an EBNF-like grammar composed of rules that are similar to the ones that we discussed in this section. Xtext automatically generates a web-based editor with the following helpful features [10]: (a) syntax highlighting, it allows to have the requirements colored and formatted with different visual styles according to the elements of the language; (b) error markers, when the tool automatically highlights the parts of the requirements indicating errors; and (c) content assist, a feature that automatically, or on demand, provides suggestions to the financial analysts on how to complete the statement/expression. In practice, these features are important to facilitate the adoption of Rimay by financial analysts. The implementation of our grammar and its editor are available online [7].

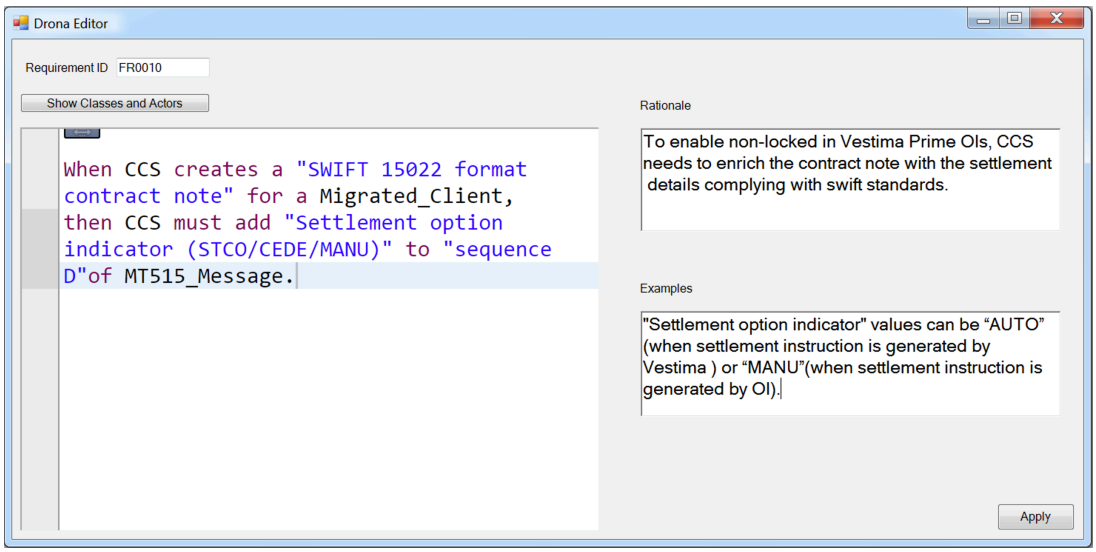


Figure 3.5: Screenshot of the requirements entry dialog box in the Rimay editor

Answer to RQ2: We operationalized the grammar of Rimay developed in Section 3.3 into a full-featured editor using Xtext. Nevertheless, Rimay is independent of any language engineering framework. Our grammar offers broad coverage of system response and condition types, following recommended syntactic structures for requirements (e.g., the use of active voice).

3.5 Empirical Evaluation

In this section, we describe a case study that evaluates Rimay developed in Sections 3.3 and 3.4. Throughout the section, we follow best practices for reporting on Case Study Research in Software Engineering [56].

[7] `https://gitlab.uni.lu/aveizaga/dsl_rimay/` (last access on 17 August 2022)

3.5.1 Case Study Design

As stated in the introduction, our evaluation aims to answer the following research questions:

- ***RQ3: How well can Rimay express the requirements of previously unseen documents?***
- ***RQ4: How quickly does Rimay converge towards a stable state?***

Figure 3.6 shows the iterative process that we follow in order to answer these two questions. To evaluate our approach, we needed to collect new SRSs that had not been used for the construction of Rimay. We applied the four steps presented in Figure 3.6 to collect new SRSs and examine the expressiveness and stability of Rimay using them: **(Step 1)** The financial analysts, on an opportunistic basis, gave us a new SRS that we had not seen before; we extracted from the given SRS its NL requirements ("Extract Requirements", Section 3.5.1). **(Step 2)** We attempted to rephrase the extracted requirements using the rules of Rimay, keeping the intent of the original requirements and ensuring that we did not lose any information content. In this step, we had to keep track of the requirements, if any, that were *non-representable* as well as the causes for such limitations ("Rephrase Requirements Using Rimay", Section 3.5.1). **(Step 3)** We analyzed the requirements that were marked as *non-representable* and enhanced Rimay to make these requirements *representable* ("Improve Rimay", Section 3.5.1). **(Step 4)** We checked whether there was a significant change in Rimay's ability to capture previously unseen content. As we argue in Section 3.5.4, it turned out that with four SRSs (i.e., four iterations of the process in Figure 3.6), we were able to reach saturation. At that point, we stopped analyzing more SRSs ("Check Rimay's Stability", Section 3.5.1). In the remainder of this section, we will not repeatedly be stating that these four SRSs were collected and analyzed iteratively and in a sequence. Instead, for succinctness, we refer to these four SRSs collectively when it is more convenient to do so.

With regard to our research questions, Step 1 and Step 2 of the process in Figure 3.6 answer RQ3, as these two steps provide information about the expressiveness of Rimay, i.e., the requirements that were *representable* or *non-representable* with Rimay. Step 3 and Step 4 of the process address RQ4, as these steps provide information about the improvements necessary for maturing Rimay to a stable state.

Extract Requirements (Step 1 of Figure 3.6)

In Step 1 of Figure 3.6, we extract the requirements from our four new, previously unseen SRSs. These SRSs were selected by senior financial analysts from Clearstream according to the criteria described in Section 3.3.2. The selected SRSs did not contain any requirement that was already analyzed while building Rimay's grammar in the qualitative study of Section 3.3.

Rephrase Requirements Using Rimay (Step 2 of Figure 3.6)

This rephrasing activity was performed in an iterative manner. Rephrasing the requirements of the four SRSs into Rimay took four iterations over two months, with each iteration requiring approximately two weeks. Each iteration was interleaved with a face-to-face session of two to three hours with at least six financial analysts (including one senior and one mid-career financial analyst). During the face-to-face validation sessions, the financial analysts checked that the intent of the requirements expressed in Rimay did not deviate from their original intent. A team composed of two annotators (the first and second researchers) rephrased the requirements using Rimay. Both annotators rephrased together the

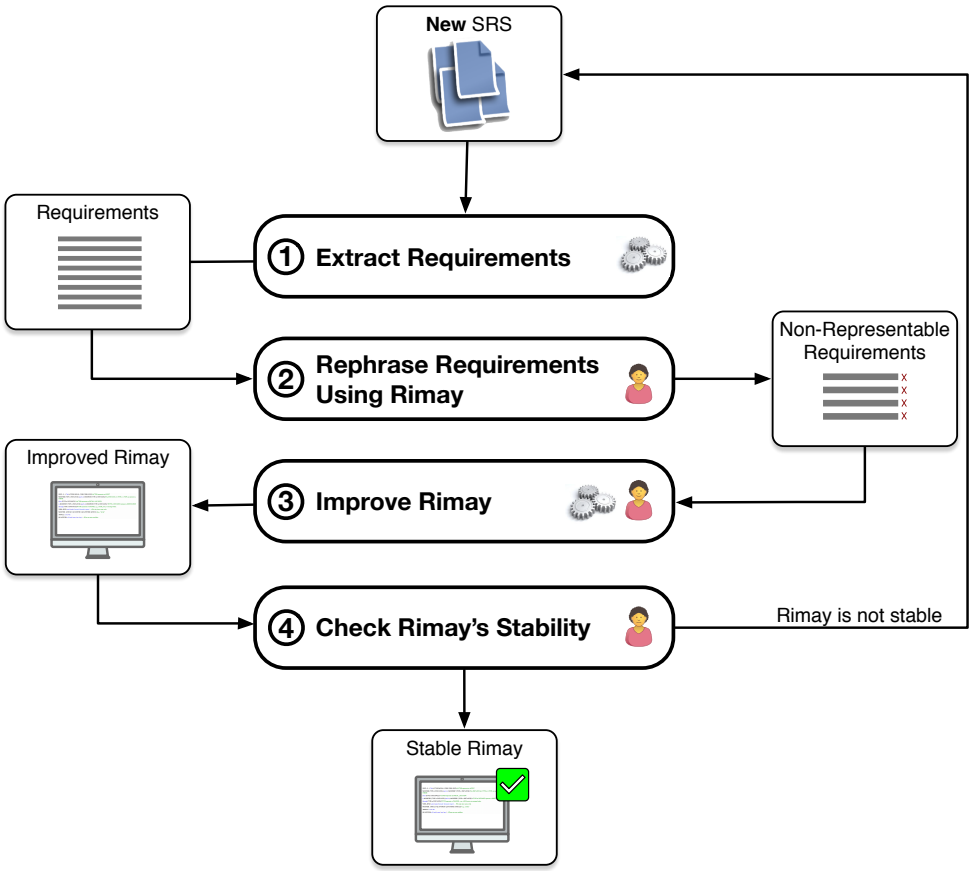


Figure 3.6: Case study design

first 20% of the requirements (i.e., 92 requirements) in order to internalize a clear procedure for (1) rephrasing a requirement into Rimay and (2) collecting the appropriate data from each requirement (i.e., representability of a requirement and possible causes of non-representability). Having a systematic procedure for rephrasing the requirements alongside the experience that the annotators had already gained while conducting our qualitative study helped ensure the quality of the rephrasing activity over the remaining 80%, i.e., the 368 (460-92) of the requirements that were rephrased by the first annotator.

A requirement can be composed of a scope, pre-conditions, an actor, and a system response. The scope and pre-conditions are optional, but the presence of at least one system response and one actor is mandatory.

Step 2 considers a requirement to be *non-representable* when some information content of the requirement cannot be captured using Rimay. A requirement is considered *representable*, otherwise. A requirement that is *non-representable* is annotated with one of following three causes:

- *Cause 1*. The requirement contains a verb that is not supported by Rimay rules. Therefore, we can either extend a Rimay rule with the verb or create a new rule.

- *Cause 2*. Part of the requirement (excluding the verb) includes information content that is not supported by Rimay. For example, the rule Send 11.1 initially defines the following information content: an AGENT who can move a THEME (e.g., data) from an INITIAL LOCATION to a DESTINATION. If a given requirement involves some information content not considered

by Send 11.1 (e.g., the CHANNEL through which the THEME is sent), then we consider that requirement to not be representable according to Cause 2.

- *Cause 3*. The meaning of the requirement is unclear and no financial analyst could clarify it.

Improve Rimay (Step 3 of Figure 3.6)

To improve Rimay, we analyzed the causes for requirements marked as *non-representable*. Concretely, we enhanced Rimay grammar by: (a) creating a new grammar rule when such requirement was marked with $Cause$ 1. To create a new grammar rule, we first identified, for each requirement, the codes according to the steps described in Section 3.3.3. The resulting codes were either identified from VerbNet or proposed by us. We then created the grammar rules following the steps described in Section 3.3.3; and (b) updating an existing grammar rule created in Section 3.3 to include either a new verb of a requirement labeled with $Cause$ 1 or missing content of a requirement labeled with $Cause$ 2.

Requirements labeled with $Cause$ 3 were not addressed in Rimay. We discuss such requirements in Section 3.6, dedicated to threats to validity.

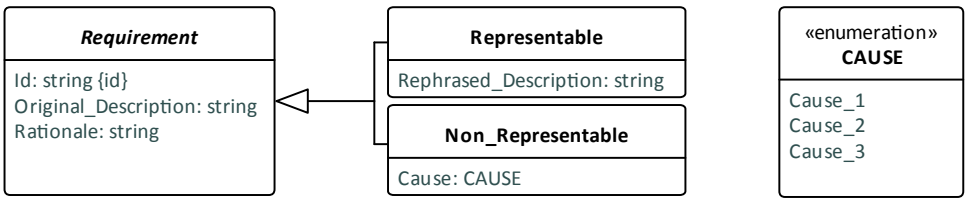
Check Rimay's Stability (Step 4 of Figure 3.6)

This step verifies whether there was a significant change in Rimay's capacity to capture the content of previously unseen NL requirements. If there is no significant change, we say that Rimay is stable, and we stop the evaluation process. Otherwise, we iterate over Step 1 to Step 4 using a new SRS until Rimay becomes stable. We refer to the notion of *saturation* to determine the point where Rimay is stable. Saturation is defined mathematically for capturing, in a simple way, when to stop our evaluation. In other words, we stop our evaluation when Rimay is expressive enough to capture all the verbs in the NL requirements of a SRS (i.e., the number of errors due to Cause 1 is zero). In our case study, we reached the saturation point during the evaluation of SRS 4.

3.5.2 Data Collection

We answered RQ3 and RQ4 by collecting data from the execution of the four steps described in Section 3.5.1. Figure 3.7 shows the data model of the requirements collected during the empirical evaluation. In our data model, a `Requirement` has an `Id` which is a unique code assigned to each requirement, an `Original_Description` and a `Rationale`. A requirement is either `Representable` or `Non_Representable`. If the requirement is `Representable`, we recorded its `Rephrased_Description`. If the requirement is `Non_Representable`, we recorded the `CAUSE` (i.e., `Cause_1`, `Cause_2` or `Cause_3`).

Figure 3.7: Data model of the collected requirements



In total, we collected 460 requirements from the four SRSs used in our evaluation. We improved the grammar rules after rephrasing one SRS and assessed the improved grammar on the next.

3.5.3 Collecting Evidence and Results

This section describes the execution and the raw data collected from our case study. The case study required the work of two annotators for two months, adding up to approximately 200 person-hours. In Section 3.5.1, we describe how the two annotators performed this task.

Table 3.7 provides the data for each of the four SRSs. For each SRS, we present the number of requirements that can and cannot be represented using Rimay. For example, the second row of Table 3.7 shows that 65 (74,7%) out of 87 of the requirements of the first SRS are *representable* in Rimay.

Table 3.7: Percentage of *representable* requirements and frequencies of causes for *non-representable* requirements

Requirements	***SRS***				***Total***
	1	***2***	***3***	***4***	
Representable and Non-representable	87	113	192	68	460
Representable	65	96	180	64	405
Non-representable	22	17	12	4	55
Non-representable - Cause 1	11	6	2	-	19
Non-representable - Cause 2	9	8	7	4	28
Non-representable - Cause 3	2	3	3	-	8

Table 3.7 shows, for the four SRSs, the frequency of the three causes (described in Section 3.5.1) in the requirements labeled as *non-representable*. For example, the second column of Table 3.7, for SRS 1, shows that for 11 requirements, the verb was not supported by Rimay (*Cause 1*). For nine requirements, some other content was not supported by Rimay (*Cause 2*). Two requirements were unclear and no financial analyst could clarify them (*Cause 3*). In total, 22 out of 87 requirements (25,3%) in SRS 1 were *non-representable.*

Next, we provide examples of *non-representable* requirements for each of the causes described above.

- Cause 1 - SRS 2: “On receipt of a request from System-A to update positions, System-B must recalculate all positions impacted by the confirmed order". Rimay does not have any grammar rule that has the verb recalculate.

- Cause 2 - SRS 1: “When the Market Calendar does not exist in the System, the System must add a record about the missing Market Calendar to the exception log". The grammar rule Mix-22.1-2, that contains the verb “add" does not support the following information content “about missing market calendar".

- Cause 3 - SRS 3: “System-A must be able to process System-B´s instructions with input media INPUT". The requirement is vague since the verb “process" is not precise enough [20].

Finally, we improved Rimay by addressing the *non-representable* requirements labeled with Causes 1 and 2, as explained in Section 3.5.1.

Coding Results.

Tables 3.8 and 3.9 show the codes and their verb members identified during our empirical evaluation. Recall from Section 3.3 that a code represents a group of verbs that convey the same information in NL requirements. The structures of Table 3.8 and Table 3.9 are the same as the structures of Table 3.4 and Table 3.5 reporting the coding results of our qualitative study discussed in Section 3.3.

Seven out of 13 codes in Tables 3.4 and 3.5 were found during our empirical evaluation. We placed the symbol "*" before the seven new codes to differentiate them from the codes that we had already identified in the qualitative study. For each new code, we created a new grammar rule. Considering that, in total, we found 48 codes during the qualitative study and the empirical evaluation, the seven (14,6%) new codes found in the empirical evaluation did not prompt drastic modifications to Rimay.

Table 3.8: VerbNet codes identified during our empirical evaluation

Codes		**Members**
Class name	**Hierarchy Level**	
begin	55.1-1	start
∗establish	55.5-1	establish
other_cos	45.4	reverse
remove	10.1	deduct
∗search	35.2	search
send	11.1	export
∗stop	55.4	stop
use	105	use
Total: 8		

Table 3.9: Codes proposed during our empirical evaluation

Codes	**Members**
∗calculate	calculate, recalculate
∗split	split
∗subscribe	subscribe
∗upload	upload
update	set
Total: 5	

3.5.4 Analysis of Collected Data

In this section, we analyze the collected data and answer RQ3 and RQ4.

Performance of Rimay on Previously Unseen SRSs (RQ3)

Table 3.7 shows that 405 out of 460 requirements (88%) across all four SRSs can be expressed using Rimay. For SRS 1, we use the version of Rimay resulting from our qualitative study while, for the following SRSs (second to fourth), we use a version of Rimay that includes the improvements made based on the previous SRS(s).

With regard to SRS 1, we note that we found five occurrences of a new verb, "use", which we had not encountered during our qualitative study. The relatively low expressiveness in this first SRS is largely

explained by the high frequency of appearance of this single verb. As one can see from Table 3.7, most requirements can be represented in Rimay across all SRSs. The improvements to the expressiveness of Rimay are brought about by small changes to Rimay. In other words, while the expressiveness of our grammar did improve as the result of analyzing more SRSs, we did not have to make major changes to the grammar. Our changes involved only the introduction of a few new verbs (as shown in Tables 3.8 and 3.9), and the enhancements of a small number of grammar rules created during our qualitative study (Section 3.3).

The most common causes for a requirement to be *non-representable*, in order of prevalence, are *Cause 2* with 28 occurrences (50.9%), followed by *Cause 1* with 19 occurrences (34.5%), and, finally *Cause 3* with 8 occurrences (14.5%). We conjecture that the main reason why *Cause 2* turns out to be the most frequent cause is that VerbNet – the lexicon we use for deriving our grammar rules – is domain-independent and may not contain certain information content that is specific to the financial domain. During our qualitative study, we identified some new content and extended the grammar rules accordingly. For example, the syntax for the rule `Send_11.1` in VerbNet specifies that an `AGENT` can move a `THEME` (e.g., data) from an `INITIAL_LOCATION` to a `DESTINATION`. Then, during the qualitative study, we identified new information content such as the temporal structure (e.g., "Before 1h00 CET") used at the beginning of requirements. Furthermore, in the evaluation, we identified extra information content such as a valid channel to send the `THEME` (e.g., a subsystem that encrypts the data).

Answer to RQ3: Rimay performed well in expressing the requirements of unseen SRSs. In particular, 405 out of the 460 requirements (i.e., 88%) used in our empirical evaluation were successfully rephrased using Rimay. The expressiveness of Rimay did steadily improve and converged to 94% in the last SRS. The rephrased requirements maintained their original intent with no information loss. We observed that improving the expressiveness of Rimay involved only small changes to its grammar. This suggests that the version of Rimay obtained from our qualitative study (Section 3.3) did not require drastic changes to maximize expressiveness.

Ensuring the Stability of Rimay (RQ4)

We refer to the notion of *saturation* to determine the point in our evaluation where we have been through enough SRSs to be confident that the updated version of Rimay is as expressive as possible to specify requirements for the financial domain. To determine if a statistically significant change is observed in the percentage of *representable* requirements, we conduct z-tests for differences in proportions of representability across different SRSs.

Saturation. Usually, saturation is reached in a qualitative study when "no new information seems to emerge during coding, i.e., when no new properties, dimensions, conditions, actions/interactions, or consequences are seen in the data" [27]. In our evaluation, the saturation point is reached when all the verbs analyzed in a SRS are already considered by Rimay (i.e., when *Cause 1* is not triggered). Specifically, as shown in Table 3.7, SRS 4 was the only SRS where no requirement was classified as *non-representable* due to *Cause 1*.

As can be seen from Table 3.7, the increment in the percentage of requirements that can be written in Rimay is tangible evidence that the changes made to Rimay were beneficial (although not extensive).

Z-test. The z-test is a standard statistical test used for checking the difference between two proportions [18]. We run one-tailed z-tests to check if the proportion (p_1) of *representable* requirements in one SRS (SRS i) is larger than or equal to the proportion (p_2) of *representable* requirements in another SRS (SRS j) analyzed thereafter. Our null and alternative hypotheses are as follows:

$$H_0 : p_1 \geq p_2$$

$$H_1 : p_1 < p_2$$

- H_0 : The percentage of *representable* requirements does not increase from SRS i to SRS j.
- H_1 : The percentage of *representable* requirements increases from SRS i to SRS j.

Each sample contains more than 30 independent data points and, though sample sizes are not equal, they are not drastically different, thus allowing the use of z-tests [83]. In total, we run six z-tests, at a level of significance of 0.05. The SRS pairs covered by these tests, alongside their corresponding proportions, are shown in Table 3.10. For example, the first row of Table 3.10 shows the input for performing a z-test over the (SRS 1, SRS 2) pair. SRS 1 contains 65 requirements that are *representable* with Rimay out of 87 requirements, and SRS 2 contains 96 requirements that are *representable* with Rimay out of 113 requirements.

Test	**Input**				
	Document Pair SRS i, SRS j	**Sample Size in SRS i**	**Sample Size in SRS j**	***Representable* Requirements in SRS i (p_1)**	***Representable* Requirements in SRS j (p_2)**
1	SRS 1, SRS 2	87	113	65	96
2	SRS 1, SRS 3	87	192	65	180
3	SRS 1, SRS 4	87	68	65	64
4	SRS 2, SRS 3	113	192	96	180
5	SRS 2, SRS 4	113	68	96	64
6	SRS 3, SRS 4	192	68	180	64

Table 3.10: Z-tests inputs

The z-scores and p-values for the z-tests are shown in Table 3.11. We conclude that the null hypothesis, H_0, is *rejected in the first five z-tests*. Therefore, there is significant evidence to claim that proportion p_1 is less than proportion p_2 at the 0.05 significance level for the first five document pairs. Concretely, this means that the proportion of *representable* requirements in SRS 2, SRS 3, and SRS 4 are significantly better than that of SRS 1. Similarly, the proportion of *representable* requirements in SRS 3 and SRS 4 are significantly better than that of SRS 2. However, the null hypothesis *cannot be rejected in the last z-test*. Therefore, the proportion of *representable* requirements in SRS 4 is not significantly better than that of SRS 3. We therefore concluded our analysis of new SRSs after completing SRS 4.

Answer to RQ4: We reached a stable version of our grammar after analyzing SRS 3 in our evaluation set. During the analysis of SRS 4, no new verbs emerged; we therefore concluded that we had reached saturation. Statistical tests confirmed that, after analyzing SRS 3, changes to Rimay did not bring about significant improvements in expressiveness.

Test	Document Pair SRS i, SRS j	z	$p - value$
1	SRS 1, SRS 2	−1,81	0,03
2	SRS 1, SRS 3	−4,50	3,35 E-06
3	SRS 1, SRS 4	−3,21	6,67 E-4
4	SRS 2, SRS 3	−2,53	0,01
5	SRS 2, SRS 4	−1,86	0,03
6	SRS 3, SRS 4	−0,11	0,46

Table 3.11: Z-test results

3.6 Threats to Validity

In the following subsections, we analyze potential threats to the validity of our empirical work according to the categories suggested by Wohlin et al. [75] and adapted by Runeson et al. [56] for case studies in software engineering.

3.6.1 Construct Validity

Construct validity reflects to what extent the operational measures that are studied really represent what the researcher has in mind and what is investigated according to the research questions [56].

We measured the percentages of the requirements that can be represented with Rimay according to the grammar rules we identified. If the criteria that we used to assess whether a requirement is *representable* are incomplete or too strict, this could constitute a threat. We therefore proposed three criteria (named *Causes*) that alleviate the risk of introducing inadequate information content into Rimay. We analyzed the *Causes* of the requirements marked as *non-representable* in order to enhance the Rimay grammar by (a) creating new grammar rules (i.e., *Cause 1*); (b) updating grammar rules to include some missing content (i.e., *Cause 1* and *Cause 2*), and (c) not considering incomplete, ambiguous or unclear information content (i.e., *Cause 3*). *Cause 1* and *Cause 2* are meant to capture missing parts that need to be included in the Rimay grammar. On the other hand, *Cause 3* focuses on the requirements that describe incorrect information content that we do not want to include in Rimay. To be sure that no important information was excluded from Rimay, we looked at the eight *non-representable* requirements labelled with *Cause 3* (Table 3.7) with the senior financial analysts from Clearstream, who agreed with our decision to discard them.

A second threat to construct validity is related to potential biases in the interpretation of requirements and the application of the qualitative codes while conducting Step 3 (i.e., Label Requirements) in Section 3.3. Ideally, to prevent biases in the coding process, one could have involved third parties in carrying out the step. However, we did not do so for two main reasons: (1) the confidentiality agreement with our industrial partner did not allow us to share the requirements with external parties, and (2) it was infeasible to identify third parties that had the specialized knowledge required for the coding process driven by linguistic resources, notably, VerbNet and WordNet. Despite not having third parties involved in this activity, we were able to mitigate potential biases and ensure the quality of the results by primarily relying on linguistic resources (VerbNet and WordNet, as noted above). Furthermore, whenever we were unable to conclusively interpret a requirement, we escalated the case to our collaborating financial analysts for deciding about the interpretation.

3.6.2 Internal Validity

Internal validity is of concern when causal relations are examined [56].

The results and the conclusions of our study strongly rely on two key activities that were performed manually: (1) the identification of codes (carried out by using protocol coding) and their members, and (2) the transformation process of requirements into Rimay. This can represent an important threat to the internal validity of our study. To mitigate biases, these two activities were systematically performed by a pair of researchers (the first and second researchers). Afterward, a third researcher reviewed and challenged some of the results of these activities. We finally improved steps (1) and (2) upon reaching an agreement between these three researchers.

Another threat to the internal validity is related to the assumption that all the requirements in SRSs should be used to create Rimay. If all the requirements in SRSs are used, incomplete and unclear requirements might be easily misinterpreted and as a consequence, incorrect information content might be included in Rimay. To tackle this threat, in Step 2 “Rephrase Requirements Using Rimay " (Figure 3.6), we first classified as *non-representable due to Cause 3* the requirements that contained either incomplete or unclear information and we then discarded those requirements.

3.6.3 External validity

External validity is concerned with the extent to which it is possible to generalize the study findings, and to what extent the findings are of interest to other people outside the investigated case [56].

The generalizability of our results is subject to certain limitations. For instance, by design, Rimay is focused on and applicable to functional requirements in the financial domain. In addition, overfitting is a potential threat because of the similarity in background among the eight financial analysts involved in the creation and validation of Rimay (Section 3.3.2). To mitigate this threat, we designed our analysis procedure (Section 3.3.3) by minimizing reliance on domain-specific terms from the financial domain. In particular, the fact that our procedure is rooted in domain-independent lexical resources (i.e., VerbNet and WordNet) significantly reduces the risk of overfitting. For this reason, we conjecture that many of our findings can be generalized to information systems in other similar domains.

A company who would want to reuse Rimay should first assess how complete Rimay is in capturing all their requirements; second, it should identify the changes required to our methodology to achieve a satisfactory degree of completeness in their given domain.

3.6.4 Reliability Validity

Reliability validity is concerned with the extent to which the data and the analysis are dependent on the specific researchers involved [56]. In order to achieve acceptable reliability, research steps must be repeatable, i.e., other researchers have to be able to replicate our results [9].

It is impossible to build a CNL that is able to represent all software requirements, and as we already acknowledged, some requirements could not be represented with Rimay. The main issues that may constitute a threat to reliability are related to how we built our CNL to be as expressive as possible. To mitigate this threat, we described in details the steps of our qualitative study and empirical evaluation following a systematic process. This process was performed by the first and second researchers and monitored by the other researchers.

3.7 Practical Considerations

In this section, we present some practical considerations for the different audiences who may be interested in the work reported in this chapter. These considerations are based on both our experience and our interactions with our industrial partner.

Considerations for CNL builders. The creation of a language editor entails a significant level of effort because there are many tasks to support, such as auto-completion and syntax highlighting. Mature language engineering frameworks make these tasks less complicated or even fully automated. For instance, we used Xtext to generate a basic editor based only on the grammar of Rimay. For us, the most challenging part of defining a grammar was to understand how to model nested expressions. The effort to customize the generic behavior of the editor generated by Xtext should be considered. In our case, we use the generic editor for our evaluation, but we are in the process of customizing the editor to further improve usability. In particular, we are simplifying the error messages shown by Rimay's editor, since they are difficult to understand for people without technical knowledge.

Considerations for companies investing into a CNL. Additional effort is to be anticipated for integrating a CNL with existing software development tools. In our case, our industrial partner uses Sparx Systems Enterprise Architect for modeling UML Use Case, Class, and Activity Diagrams. A key consideration for our partner was therefore to be able to reference (from requirements) the elements of UML models in Enterprise Architect. To provide such functionality, Rimay's editor dynamically tracks the model elements that need to be referenceable from requirements. This allows Rimay's editor to provide context-sensitive auto-completion assistance as analysts type in their requirements. Furthermore, if an analyst introduces in a requirement an element that does not already exist in the UML model, our editor will notify the analyst, asking whether the new element should be added to the UML model.

Whether an organization should invest into a CNL for requirements also depends on how requirements are elaborated and used within the organization. Generic text editing tools may suffice for analysts working on small projects. In our case, the types of projects our industrial partner is engaged in justified the construction of a CNL; the projects are not only large and complex but also involve multiple analysts from geographically dispersed locations. Systematic requirements writing practices that help mitigate incompleteness and ambiguity are thus key for our partner. In addition, organizations are interested in extracting accurate information from the requirements as a prerequisite step for automating such tasks as consistency checking between models and (textual) requirements, as well as generating test cases from requirements. Working toward such automation objectives would be very difficult without structured requirements, thus further justifying investment into a CNL. In more recent work (Chapter 5), our partners recognized that generating acceptance criteria exclusively from models would miss critical information that is available only in NL requirements. In that work, we elaborate on how acceptance-criteria-relevant information in NL requirements expressed via Rimay can be used for enriching requirements models and subsequently obtaining more precise and complete acceptance criteria.

Extending Rimay to other domains. In this chapter, we focused on the financial domain. However, Rimay may be adapted for use in other domains. We recommend the following steps to adapt Rimay to a given organization:

1. *Select requirements.* The organization selects functional requirements that are representative of commonly used conditions and action phrases.

2. *Rephrase requirements using Rimay.* The organization first rewrites the requirements selected in the previous step using Rimay, and second, labels each *non-representable* requirement with one of the three causes described in Section 3.5.1. Domain experts must ensure that the intents of the requirements written in Rimay do not deviate from the original ones.
3. *Improve Rimay.* For each *non-representable* requirement, the organization should enhance Rimay's grammar by either updating the existing grammar rules or creating new ones. The organization must follow the methodology described in Section 3.5.1 to perform this step.
4. *Generate and integrate Rimay's editor.* Once the organization has enhanced Rimay's grammar to support previously *non-representable* requirements, it generates and integrates the extended version of Rimay's editor into the modeling and development tool used within the organization, if available. If the editor is created using the Xtext language engineering framework, it can be used as an Eclipse-based plugin or integrated into web applications.

The time required for an organization to extend Rimay is difficult to estimate since doing so depends on several factors: (1) the number of requirements to be rephrased using Rimay, (2) the degree of access to engineers who know Rimay's methodology and have a background in language engineering, and (3) sufficient access to domain experts.

Since there is currently no extension of Rimay, to gain insights into the time required to extend Rimay, we discuss relevant aspects of the evaluation and refinement of Rimay presented in Section 3.5. The evaluation of Rimay included (1) a set of 460 functional requirements, (2) two engineers (first and second researchers), and (3) six domain experts. The entire evaluation and refinement process required 200 hours from the engineers and eight hours from domain experts over a span of two months (Section 3.5.3). The (approximate) distribution of effort observed across the four steps of our approach was as follows: Select requirements (10%), Rephrase requirements using Rimay (60%), Improve Rimay (25%), and Generate and integrate Rimay's editor (5%).

3.8 Conclusions

In this chapter, we proposed a rigorous methodology to define controlled natural languages (CNLs) for requirements specifications. We applied this methodology to develop a CNL, which we named Rimay, for expressing functional requirements in the financial domain. Rimay's grammar was derived from a qualitative study based on the analysis of 2755 requirements from 11 distinct projects. In this qualitative study, we identified the information content that financial analysts should account for in the requirements of financial applications.

We conducted an empirical evaluation of Rimay in a realistic setting. This evaluation measured the percentage of requirements that can be represented using Rimay. We observed that, on average, 88% of the requirements that we evaluated in our case study (405 out of 460) could be expressed using Rimay. Additionally, we analyzed how quickly Rimay would converge and stabilize to even higher percentages when refined after each new requirements specification was analyzed.

To a large extent, because it was specifically designed to be domain independent, we believe that Rimay can address the broader domain of data-intensive information systems. That said, future investigations remain necessary to determine whether and how Rimay can be specialized for other domains.

While CNLs and requirements patterns have generated a lot of attention in recent years as a vehicle for improving the quality of natural-language requirements, to our knowledge, no previous study has proposed and evaluated a CNL based on a qualitative analysis of a large number of industrial requirements and following a systematic process using lexical resources. A significant portion of this chapter was dedicated to developing and discussing such a systematic process with the goal of making this process repeatable; this way, other researchers and practitioners interested in developing their own CNLs can benefit from our proposed process and possibly even use Rimay as a starting point.

For future work, we intend to conduct a user study on the usefulness of Rimay. This would assess in a more conclusive manner whether financial analysts benefit from using Rimay for specifying functional requirements.

Chapter 4

Quality Assurance on Requirements

4.1 Introduction

Requirements are important for software development processes. Requirements are generally expressed using natural language (NL), which is widely used in many industrial applications. NL is ubiquitous in defining software requirement specifications (SRSs) because it can be used in all application domains and understood by all stakeholders in a software development project [51]. A recent study reports that 61% of users prefer to express requirements using NL [32]. Yet, despite its popularity, NL is highly prone to quality problems, such as under-specification, vagueness, ambiguity, complexity, and incompleteness [42, 22].

The main causes of software project failures in the industry are requirement quality problems [1, 28]. When such problems are not fixed in the early development phases, they carry over to subsequent development phases, and fixing them in later phases of development is a costly and time-consuming process. Improving the quality of NL requirements by identifying quality problems at early software development stages is quickly becoming a pivotal need in industry applications.

Our industrial partner, Clearstream Luxembourg, reported elevated costs associated with in-house processes to improve requirement quality, which involve several manual iterations and are thus prone to errors. A tool that automatically detects problems in NL requirements and guides analysts to improve the quality of NL requirements is highly desirable.

Various approaches have been proposed to improve the quality of NL requirements by detecting semantic and syntactic problems [50, 20, 26, 60, 48]. However, these approaches do not provide recommendations to analysts for improving the quality of the requirements. Furthermore, existing works do not account for many of the recurrent problems introduced by analysts. They include non-atomic requirements (multiple actions in the system response), incomplete content information in the segments of requirements (e.g., missing actors, verbs in conditions), missing segments in requirements (e.g., a requirement misses a system response), and incorrect order of segments in requirements (e.g., conditions after the system response).

In collaboration with our industrial partner, we developed a tool that addresses quality problems in requirements. Throughout this chapter, the term “smells” refers to quality problems that can lead

to defects. A smell has a precise location and detection mechanism that facilitates its inspection and identification [21]. Our approach detected 10 smells that are commonly present in requirements in the financial domain. We relied on natural language processing (NLP) techniques to analyze the information content of functional requirements to detect smells. This chapter provides suggestions for fixing smells and improving requirement quality. To accomplish this, we derived requirement patterns from an existing controlled natural language (CNL): Rimay (Chapter 3). Our study is guided by the following research questions (RQs):

- **RQ1) What NL requirement smells are commonly found in the financial domain?** We answer RQ1 by proposing 10 smells in NL requirements. The smells violate the expected requirement quality categories of completeness, clarity, atomic requirements, and correctness. The 10 smells were derived after analyzing a set of 384 requirements in the financial domain. They are able to detect quality problems in individual segments of the requirement (segment level) and in the requirement as a whole (requirement level).

- **RQ2) How can smells be detected?** We answer RQ2 by proposing an automated approach that relies on NLP methods to detect the smells in NL requirements. These methods include Tregex extraction rules, structural patterns, and glossary search.

- **RQ3) How can we suggest templates to improve requirement quality?** We answer RQ3 by proposing an automated approach that analyzes the overall syntax of an NL requirement. According to this syntax, our approach matches and suggests a suitable Rimay pattern. Following our suggested patterns increases the chance of fixing smells.

- **RQ4) Can our approach correctly indicate the occurrence of smells?** To answer RQ4, we evaluated the smell-identifying accuracy of our tool. To accomplish this, we conducted a case study using SRSs from financial applications. We compared our results against a handcrafted ground truth. Our evaluation results suggest that our approach accurately detects smells with a precision of 88% and a recall of 74%.

- **RQ5) How accurate is our approach to recommending requirement templates to fix smells?**

  RQ5 assesses how well our approach suggests appropriate templates to fix smells in NL requirements. We compared the performance of our approach against a handcrafted ground truth composed of a set of NL requirements from financial applications. Our approach accurately suggested an appropriate Rimay pattern with a precision of 89% and a recall of 82%.

The main contributions of this work are (1) a catalog of smells, (2) an automated approach that detects smells on SRSs and suggests Rimay patterns to fix smells, and (3) an industrial case study in the financial domain to measure the performance of our application to detect smells and suggest patterns.

4.2 Requirements Smells and Rimay Patterns

This section describes the process we followed to derive requirements smells. We propose a catalog of smells that describes the syntactic and semantic errors commonly found in requirements. Furthermore, we describe the procedure we followed to derive the Rimay patterns. These patterns will be shown as suggestions for the analysts to fix smells in the requirements.

4.2.1 Requirements Smells

This section aims to answer RQ1: **What are the commonly found smells in NL requirements in the financial domain**? To do so, we first analyze the concepts and constructs of the Rimay language to derive the quality attributes that Rimay (Section 3.4) enforces through its grammar to minimize the risk of having poorly written requirements. The quality attributes we derived are as follows: (1) **Completeness** refers to the presence of all the information required for the requirement to be complete. Rimay achieves completeness by having constructs that ensure the presence of certain contents. For example, Rimay warns analysts when a requirement does not have a system response. (2) **Clarity** refers to the usage of structures, phrases, and words that are free of ambiguity. Rimay achieves clarity by providing a set of predefined structures and a fixed vocabulary. (3) **Atomic requirements** refer to a natural language statement that describes a single system function. The Rimay language recommends analysts not have more than one system response in a requirement. (4) **Correctness** refers to enforcing the presence of correct information content in the correct order of appearance. The constructs of Rimay minimize the risk of having incorrect information content in the segments of a requirement. For example, Rimay does not allow the use of a modal verb in conditions.

We use Rimay because (1) there is a match between the domain where Rimay is applicable and our case study context, and (2) the Rimay concepts and constructs are characterized by providing precise syntax and semantics that minimize the risks of having poor quality requirements.

Next, to derive the smells, we analyzed a set of 384 NL requirements provided by our industrial partner to find NL requirements that violated the quality attributes of completeness, clarity, atomic requirements, and correctness. Table 4.1 shows the 10 smells we identified. The first column shows the name of the smell. The second column provides a description of the smell. The third column indicates the quality attribute that the smell violates. The proposed smells were validated by our industrial partner, who agreed that the smells describe errors frequently made by analysts when writing NL requirements.

4.2.2 Rimay Patterns

This section describes the process conducted to derive requirements patterns from the Rimay language. Rimay provides structures with specialized concepts and constructs to specify functional requirements. However, Rimay does not provide patterns that guides the analysts on how to write a requirement.

To derive the Rimay patterns, we first created a conceptual model for the concepts underlying the Rimay language. Figure 4.1 shows a high level of abstraction of the Rimay concepts. The model defines five concepts for "Requirement". "Condition_Structure" further defines five concepts. Moreover, the "Action_Phrase" defines 58 concepts (Section 3.4). Figure 4.1 shows only a few concepts for "Action_Phrase". Furthermore, the model defines the concept "Condition" used by the concepts "While_Structure", "If_Structure", and "Temporal_Structure". The concept "Trigger" is used by the concepts "If_Structure", "When_Structure", and "Temporal_Structure". The concept "Free_Expression" is used by the concepts "When_Structure", "Where_Structure" and "Temporal_Structural". The concept "Time_Adverb" is used by "Temporal_Structure" (Section 3.4).

Next, we derived possible combinations of Rimay concepts to create requirement patterns. These combinations represent valid sequences of Rimay concepts to write requirements.

Table 4.1: Catalogue of 10 smells

Smell name	Description	Quality Attributes Violated
Non-atomic	Non-atomic happens when there is more than one action in the system response	Atomic requirement
Incomplete requirement	Incomplete requirement happens when the requirement does not have a system response but has other optional segments, i.e., condition and scope	Completeness
Incorrect order requirement	Incorrect order occurs when a condition is located after the system response. This construct leads to a vague interpretation of the occurrence of the condition	Correctness
Coordination ambiguity	Coordination ambiguity happens when a requirement has two or more conditions and these conditions are connected by a coordinated conjunction "or"	Clarity
Not a requirement	Not a requirement happens when the requirement does not contain any part, i.e., scope, condition, and system response	Correctness
Incomplete condition	Incomplete condition happens when the condition lacks of either actor or verb	Completeness
Incomplete system response	Incomplete system response happens when the system response lacks of either the actor, the modal or the verb	Completeness
Incomplete scope	Incomplete scope happens when the scope misses a noun	Completeness
Passive voice	Passive voice happens when either the condition or system response are described in the passive voice and the description misses the subject	Completeness
Not a precise verb	Not precise verb happens when either the verb of the condition or system response is not precise enough. The verb misses a precise action. The list of our not precise verbs includes: "accomplish", "account", "come", "consider", "default", "define", "do", "get", "make", "perform", "process", "propose", "make", "raise", "read", "support", and "want". This list was curated by the analysts from Clearstream	Clarity

Table 4.2 outlines the 10 Rimay patterns derived from the combinations of the concepts in the conceptual model of the Rimay language. The first column shows the name of the pattern. The second column specifies the pattern. Finally, the third column indicates the combination of the concepts of the Rimay language used to derive the pattern. Table 4.2 does not include all the keywords from the Rimay concepts and does not include the templates for the Action-Phrases. Refer to Chapter 3 for a complete reference to the concepts and constructs of the Rimay language.

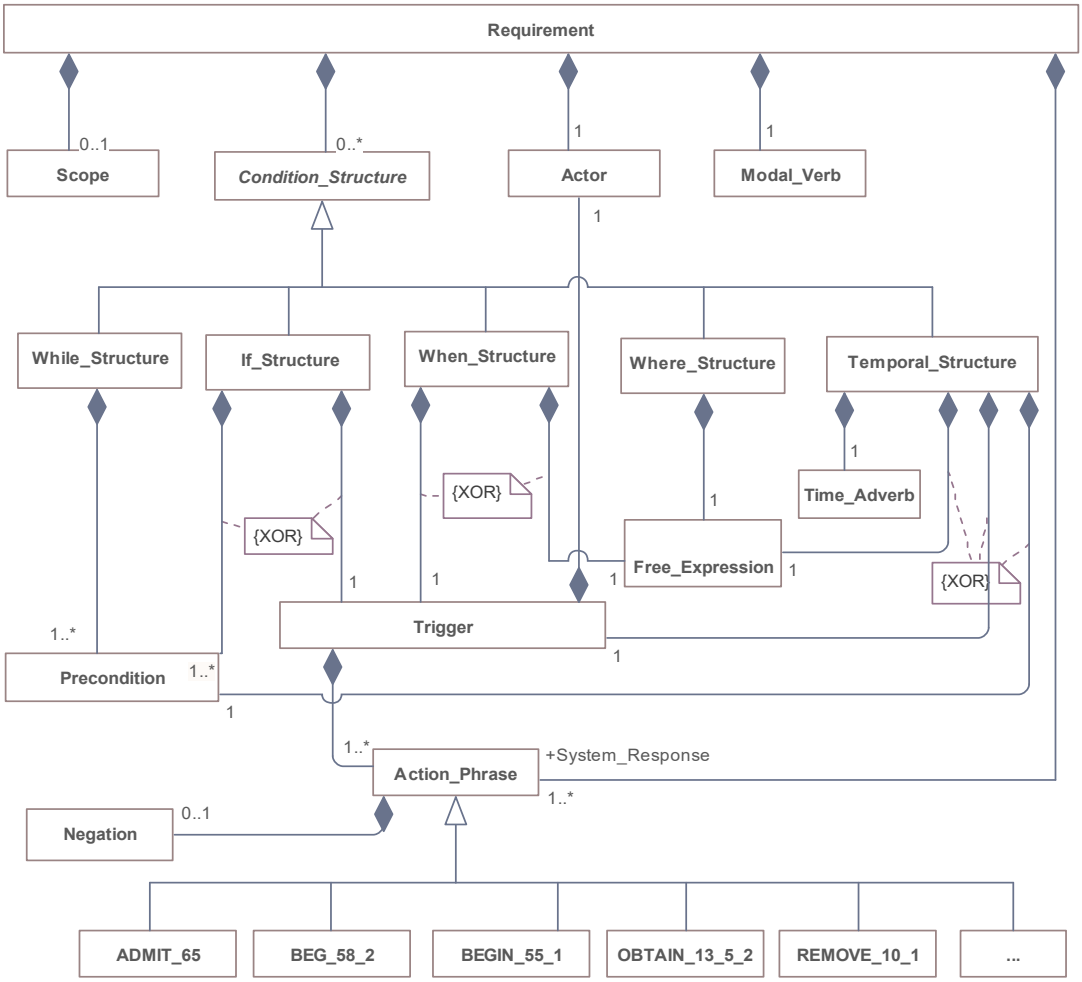


Figure 4.1: Rimay conceptual model

The derived patterns will be used by our approach to provide suggestions to analysts when our approach detects smell(s) in NL requirements. If analysts follow our suggestions and rewrite the requirements using the suggested patterns, the presence of errors can be minimized, and the quality of requirements can improve.

It is important to note that the proposed Rimay patterns aim to provide a point of reference to the sequence of Rimay concepts to start writing Rimay requirements. We assume that the analysts possess prior knowledge of the concepts and constructs of the language. Recall from Section 3.4 that the Rimay editor provides useful features (syntax highlighting, error markers, and content assist) that help analysts write requirements.

4.3 Approach

Figure 4.2 provides an overview of the four steps of our approach. The inputs are (1) software requirements specifications (SRSs), (2) Rimay patterns (Section 4.2.2), and (3) a catalog of 10 smells commonly found in NL requirements (Section 4.2.1). The SRSs contain a set of requirements. A requirement in our context specifies what the system response an actor is expected to receive when providing certain inputs if certain conditions are met.

Table 4.2: Rimay patterns

Pattern Name	Rimay Pattern	Mapping to Conceptual Model
1. Scope and system response	`For each|all|... "Text",|then the? Actor must <Action> (every "Text")?.`	Scope, Actor, Modal_Verb, and Action_Phrase
2. Scope, condition (precondition), and system response	`For each|all|... "Text", if <Property> is equal to | is less or equal to |... "Value",|then the? Actor must <Action> (every "Text")?.`	Scope, Precondition, Actor, Modal_Verb, and Action_Phrase
3. Scope, condition (trigger), and system response	`For each|all|... "Text", when the? Actor <Action> (every "Frequency")? ,|then the? Actor must <Action> (every "Text")?.`	Scope, Trigger, Actor, Modal_Verb, and Action_Phrase
4. Scope, condition (time), and system response	`For each|all|... "Text", after| before "Text",|then the? Actor must <Action> (every "Text")?.`	Scope, Time_Adverb, Actor, Modal_Verb, and Action_Phrase
5. System response	`The? Actor must <Action> (every "Text")?.`	Actor, Modal_Verb, and Action_Phrase
6. Condition (precondition) and system response	`If <Property> is equal to | is less or equal to |... "Value",|then the? Actor must <Action> (every "Text")?.`	Precondition, Actor, Modal_Verb, and Action_Phrase
7. Condition (trigger) and system response	`When the? Actor <Action> (every "Frequency")? ,|then the? Actor must <Action> (every "Text")?.`	Trigger, Actor, Modal_Verb, and Action_Phrase
8. Condition (time) and system response	`After|Before "Text",|then the? Actor must <Action> (every "Text")?.`	Time_Adverb, Actor, Modal_Verb and Action_Phrase
9. Scope, multiple conditions, and system response	`For each|all|... "Text", if <Property> is equal to | is less or equal to |... "Value",|and when the? Actor <Action> (every "Frequency")? ,|then the? Actor must <Action> (every "Text")?.`	Scope, Condition Structure (two or more), Actor, Modal_Verb, and Action_Phrase
10. Multiple conditions and system response	`If <Property> is equal to | is less or equal to |... "Value",|and when the? Actor <Action> (every "Frequency") ,|then the? Actor must <Action> (every "Text")?.`	Condition Structure (two or more), Actor, Modal_Verb, and Action_Phrase

In Step 1, we apply preprocessing steps to the NL requirements extracted from the SRSs. In Step 2, our approach separates requirements into segments (i.e, scope, condition, and system response). Our approach relies on patterns and a segment splitter to split requirements into segments. The patterns were written using Tregex, which is a language for defining patterns in text syntax trees. In Step 3, our approach detects smells in NL requirements by applying several techniques, such as structural patterns, Tregex patterns, rules, and glossary search. In our context, a structural pattern refers to a pattern that checks the sequence of words in a requirement. Finally, in Step 4, our approach suggests a pattern for analysts to fix the requirement and convert it into Rimay. Rimay helps decrease the risk of quality problems in requirements since it has precise syntax and semantics. Throughout this section, we provide examples of all the steps of our approach using the running example shown in Figure 4.3.

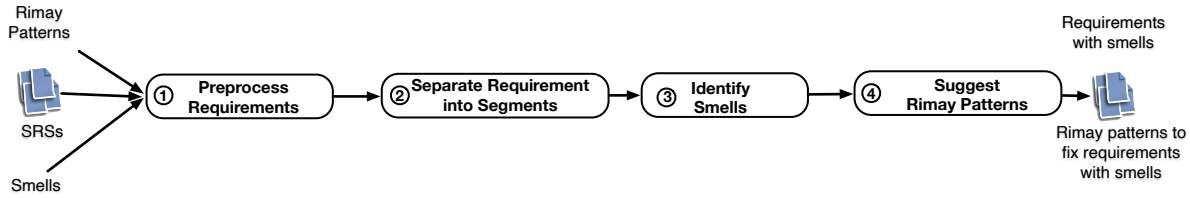


Figure 4.2: Approach overview

4.3.1 Step 1: Preprocess Requirements

We apply a set of preprocessing steps to the NL requirements extracted from SRSs, including tokenization (dividing the text of the requirement into tokens, such as punctuation marks and words), post-tagging (assigning part of speech tags to tokens, such as pronouns, verbs, and adjectives), and constituency parsing (a process that identifies the structural units of sentences, e.g., clause, noun phrase, verb phrase).

We also remove single and double quotes and maintain the MS Word metadata. These metadata include line breaks (a point at which text is split into two lines) and bullet points (an item on a list). In our context, these metadata are useful for detecting multiple conditions and system responses.
Figure 4.3 shows an example of a requirement that was preprocessed by our approach. Our approach removed the double quotes since we observed that single and double quotation marks prevented us from correctly identifying the structural units of the sentences.

4.3.2 Step 2: Separate Requirement into Segments

This step is intended to automatically separate the NL requirement into segments (i.e., scope, condition, and system response). We automatically separate segments from NL requirements to (1) analyze each segment of the requirement independently with the purpose of finding smells (Step 3) and (2) determine the overall syntax of the requirements with the purpose of suggesting a precise Rimay pattern (Step 4). We created an automated procedure to separate requirements into segments using the following methods: (1) Tregex patterns and (2) segment splitters.

Tregex patterns. The Tregex query language allows users to define regular expression-like patterns in tree structures [38]. Tregex is designed to match patterns that involve the content of the tree nodes and the hierarchical relations among the tree nodes of the syntax tree of the requirements. To separate a requirement into segments, we created a set of patterns using Tregex. In our context, a Tregex pattern

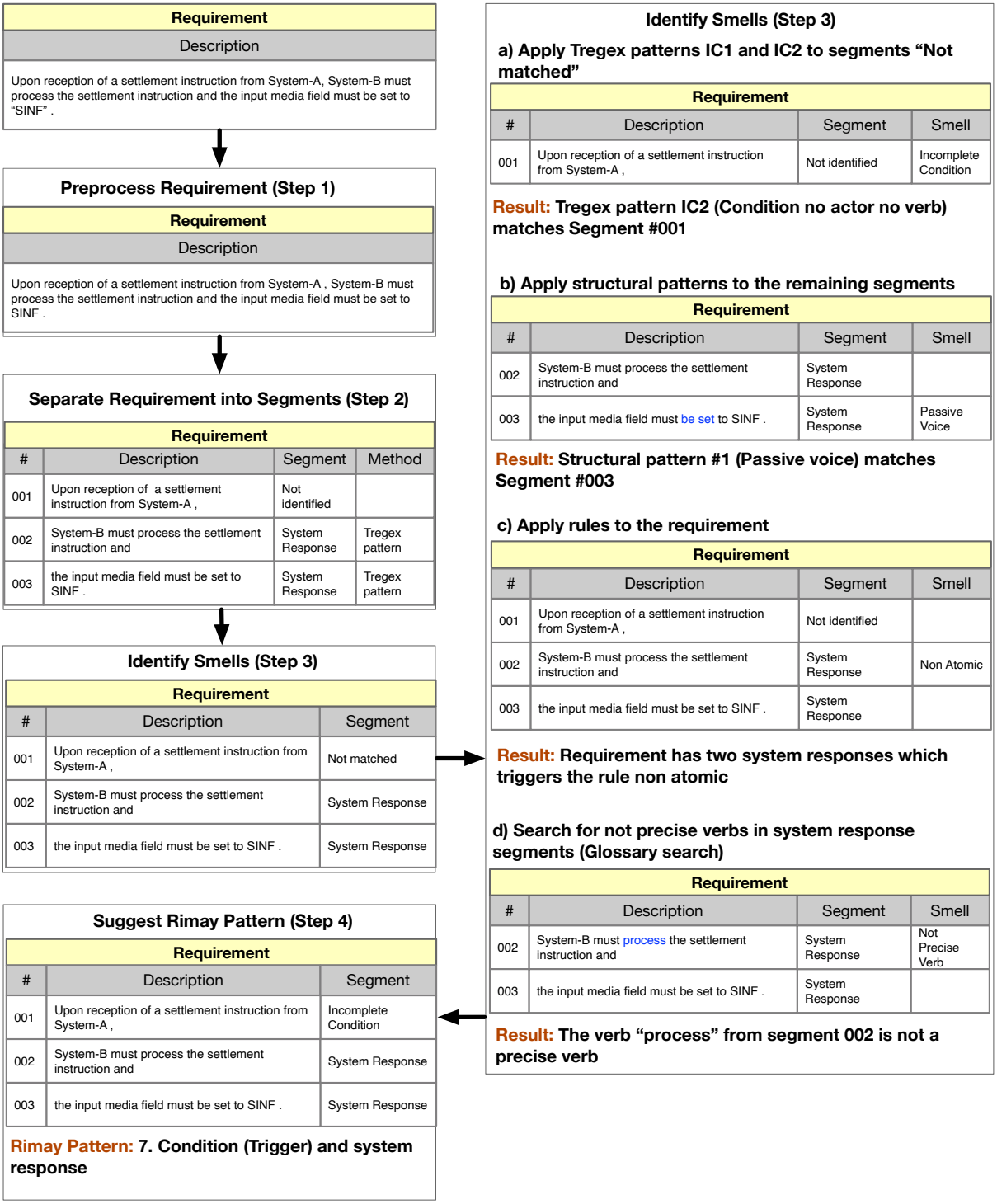


Figure 4.3: Detection smells and suggested Rimay patterns

ID	Segment	Tregex Pattern
SC1	Scope	(PP < ((IN < For) $+ NP)) > (S < (/(RB | , | ADVP)/ $+ (NP $+ VP)))
SC2	Scope	((PP < ((IN < For) $+ NP)) [>− ((VP < MD) $- NP)] & ! >> PP)
C1	Condition	WHADVP $+ (S <, (S < (NP $+ VP)))) > SBAR &!» VP
C2	Condition	(SBAR < (WHADVP $+ (S < (NP $+ VP)))) > (S > SBAR) & > !VP
C3	Condition	((IN < once) $+ (S < (NP $+ VP))) [> SBAR | > S] & > !VP
C4	Condition	SBAR < (WHADVP $+ S < (NP $++ VP))
C5	Condition	(WHADVP $+ (S < (NP $+ VP)) !» /(VP|SBAR)/)
C6	Condition	(WHADVP !< /(of | to)/) $+ (NP $+ VP) !» /(VP | SBAR)/
C7	Condition	(SBAR < ((WHADVP !« that) $+ (S &«, (NP $++ VP | $++ VB)))) !» VP
C8	Condition	(SBAR < ((WHADVP !« that) $+ (S !« SBAR &«, (NP $++ VP | $++ VB)))) !» VP
C9	Condition Time	(PP < (IN < (/^(after | before)$/) $+ (NP !< VB < NN))) > S
SR1	System	NP $+ (VP < (MD ?$+ ADVP $++ (VP «, (/VB.?/ $+ (S < (NP $++ VP)))))) > S

S: Clause, SBAR: Subordinate clause, WHADVP: Wh-adverb phrase, VP: Verb phrase, VBG: Verb gerund, NP: Noun phrase, PP: Prepositional phrase, IN: Preposition, NN: Noun, RB: Adverb, ADVP:Adverb phrase, MD: Modal, and VB: Verb

Table 4.3: Tregex patterns to identify segments in requirements

matches the specific structure of the constituency structure of a requirement. The constituency structures of the requirements were obtained in Step 1 (Section 4.3.1).

To create Tregex patterns, we analyzed the syntax of 384 requirements. The set of requirements was similar to the set used in Section 4.2.1. The process to derive the patterns was as follows: (1) we grouped the requirements that had the same segments (a segment could be a scope, condition, and system response); (2) we analyzed the constituency structures of the segments of each group; (3) we derived patterns that matched the constituency structure of the segments of each group; (4) we gathered all the patterns that matched the same segment to refine them and merge them, if possible. If not, a segment may have more than one pattern.

Table 4.3 outlines the 12 Tregex patterns that we derived after following the process described above. In total, we derived two patterns that detect scope (SC1, SC2), nine patterns that detect conditions (C1–C9), and one pattern that detects system response (SR1).

Figure 4.4 shows an example of the usage of the pattern C4 to extract the condition of a requirement. Figure 4.4 shows the constituency parsing tree of requirement R. The requirement has a scope (For all depositories), a condition (when System-A receives an email alert from System-B), and a system response (System-A must create an MT530_transaction). Some concepts of requirement R were anonymized to comply to the confidentiality agreement with our industrial partner. The condition is composed by a subordinated conjunction (WHADVP: when), a noun phrase (NP: System-A), and a verb phrase (VP: receives an email alert from System-B). Figure 4.4 shows the pattern that matches the condition of requirement R. This pattern identifies a subordinated clause (SBAR) that immediately dominates (<) a Wh-adverb phrase (WHADVP), which is the immediate left sister ($+) of a clause (S). Clause (S) immediately dominates (<) a noun phrase (NP), which is the immediate left sister ($+) of a verb phrase (VP).

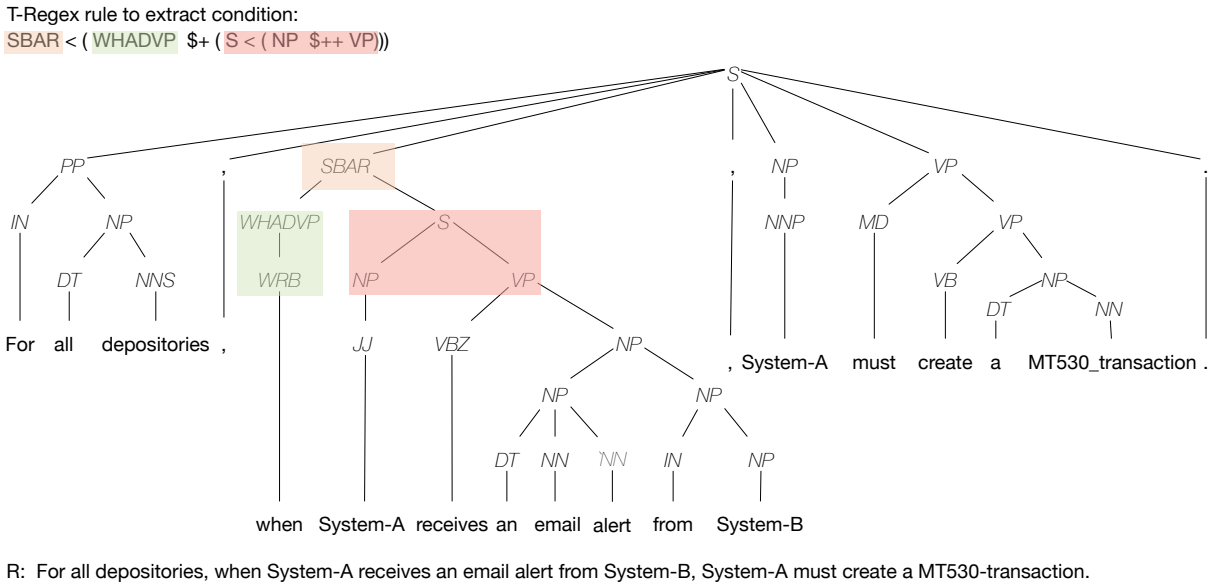


Figure 4.4: Example Tregex pattern to match conditions after system response

Segment	Information Content
Scope	`For [each | all | none] noun`
Condition	`[When | if | where | while | until] noun verb`
System response	`[then | <lb> | ; | else | otherwise] noun modal-verb verb`

Table 4.4: Information content characterizing the requirement segments

Segment splitter. We propose a segment splitter that attempts to separate requirements into segments (i.e., scope, condition, and system response). The segment splitter is only used when the Tregex pattern fails to identify segments in requirements. This failure may be caused by a mismatch between the syntax of the segment and the syntax described in the Tregex patterns. The new syntax is related to structures not observed during the creation of the Tregex patterns.

To create our segment splitter, we first collected the keywords that characterized the beginning of each segment of the requirement. These keywords include, for the condition, "when", "if", "where", "while", "until", for the scope, "for", and for the system response "then", (line break),";","else", and "otherwise".

Our segment splitter then detects the above keywords in the requirements to split them into segments. Then, our approach validates each segment. A segment is considered valid if it has the mandatory content information. Table 4.4 shows the mandatory information content that each segment should have to be considered valid.

Finally, in the scenarios where segments miss mandatory concepts, our approach labels them as "Not Matched". These segments will be further analyzed to detect smells in Step 4.

In the example depicted in Figure 4.3, we first applied all the Tregex patterns (Table 4.3) to the requirement. The pattern SR1 was matched and segments 002 and 003 were identified as a system response. Furthermore, our approach applied our segment splitter. The segment splitter could not identify segment 001, since the word "Upon" is not present in our keyword list.

4.3.3 Step 3: Identify Smells

This section aims to answer **RQ2: How can smells be detected?** For this purpose, in this step, we describe how we analyze the segments of requirements resulting from Step 2 with the purpose of detecting any of the smells introduced in Section 4.2.1.

We created an automated procedure to detect the smells using the following techniques: (1) structural patterns (these patterns analyze the presence and sequence of the words that describe the concepts of segments of a requirement), (2) rules (our rules check which segments the requirement has, how the segments of the requirements are connected, and the sequence of the segments in a requirement), (3) TRegex patterns (this method checks for the specific syntax used to describe incomplete information content in the segments of requirements), and (4) glossary search (this method extracts the verbs from conditions and system responses to check if the verbs are not precise). Our approach detects the smell "Not a precise verb" using the glossary search method. Our approach finds the smells, "Non-atomic", "Incomplete requirement", "Incorrect order requirement", "Coordination ambiguity", and "Not a requirement" using the rules. It detects the smell "Incomplete condition" using structural patterns and Tregex patterns. It uses structural patterns to detect the smells "Incomplete system response", "Incomplete scope", and "Passive voice". In the following section, we describe in detail the aforementioned methods and the smells that each method detects.

Structural patterns. This method analyzes the segments of the requirement using structural patterns to detect the smells "Incomplete condition", "Incomplete SR", "Incomplete Scope", and "Passive voice". A structural pattern checks the presence of certain words that describe the concepts of segments in a specific sequence.

We defined 15 structural patterns, shown in Table 4.5, that were derived to detect the above smells. There are eight patterns that check for the smell "Passive voice" in the following tenses: present simple, present perfect, past simple, and past perfect. Moreover, we derived three patterns that check for the smell "Incomplete condition", three patterns to detect the smell "Incomplete system response", and one pattern to detect the smell "Incomplete scope".

Structural patterns were matched to the segments resulting from Step 2. For example, the structural pattern "Passive voice #7" matched the condition of the following requirement "When/`WRB` a/`DT` System-A/`NNS` has/`VBZ` been/`VBN` assigned/`VBN` via/`IN` propagation/`NNP` ..". The condition contains v_1 verb "has" in the present tense, followed by v_3 verb "been" in the past participle, followed by v_4 verb "assigned" in the past participle.

In the example depicted in Figure 4.3 (Step 3b), our approach applied our structural patterns to the segments of the requirement. We matched the structural pattern "Passive Voice 1" with segment 003. This segment has the v_1 verb "be" in its base form and the verb v_4 verb "set" in its past participle form.

Rules. We proposed a set of rules that analyze the segments of the requirements identified in Step 4.3.2 with the purpose of detecting the smells "Non-atomic", "Incomplete requirement", "Incorrect order requirement", "Coordination ambiguity", and "Not a requirement". Our rules aimed to (1) analyze how the segments are connected to each other and (2) determine the sequence of the segments of the requirement. In the following, we describe these rules:

Smell	#	Structural Pattern	Example
Passive voice	1	v_1 v_4	"...is taken..."
	2	v_1 adv v_4	"...has not taken..."
	3	v_2 v_4	"...was taken..."
	4	v_2 adv v_4	"...was not taken..."
	5	v_2 v_3 v_4	"...had been taken..."
	6	v_2 adv v_3 v_4	"...had not been taken..."
	7	v_1 v_3 v_4	"...has been taken..."
	8	v_1 adv v_3 v_4	"...has not been taken..."
Incomplete condition	9	sc o_1	"When for each subscriptions..."
	10	sc v_1	"When receives the subscription order ..."
	11	sc n_1 o_1	"When the System-A seennd the subscription order ..."
Incomplete system response	12	md v_1	"then must send the settlement request..."
	13	n_1 v_1	"System-A closes the Filter screen..."
	14	n_1 md o_2	"System-B must sed the settlement request..."
Incomplete scope	15	p q o_3	"For each using the T-model..."

v_1: Verb base form (be | have), v_2: Verb past (be | have), v_3: Verb past participle (be), v_4: Verb past participle, adv: Adverb (not) sc: Subordinated conjunction, n_1: Noun, md: Modal verb, o_1: Other word than noun and verb, o_2: Other word than verb, p: Preposition (for), q: Quantifier, o_3: Other word than noun

Table 4.5: Structural patterns for smell detection

1. **Non-atomic:** The requirement has more than one segment of the type system response.
2. **Incomplete requirement:** The requirement misses the system response segment, but it has other segments, such as scope and condition.
3. **Incorrect order requirement:** The requirement has one or more segments of the type condition that are after the system response.
4. **Coordination ambiguity:** The requirement has two or more subsequent conditions. Our approach extracts the word(s) or character(s) that separate the conditions. If the separator(s) is the conjunction *"or"*, then our approach triggers the smell "Coordination ambiguity".
5. **Not a requirement:** The segments of the requirements are not scope, condition, or system response.

In the example depicted in Figure 4.3 (Step 3c), we applied our rules to the requirement. The rule "Non-atomic" identified two system responses in the requirement that triggered the smell "Non-atomic".

Tregex patterns. Recall from Section 4.3.2 that the Tregex patterns match the specific structures of the constituency parsing tree resulting from the NL requirement. While analyzing the set of requirements in Section 4.3.2, we observed two groups of requirements that contained incomplete conditions.

The following are two examples of these requirements: EIC1 and EIC2. *"EIC1: Upon reception from System-A the status Pending of an Instruction, then ..."*. The condition of EIC1 misses the actor and the verb. Instead of a verb, the condition has the noun "reception". *"EIC2: When creating a new participant, System-A must..."*. The condition of EIC2 misses the actor, and the verb is described using a gerund. To detect these conditions, we derived two Tregex patterns. Therefore, we gathered a set of 55 requirements

ID	Tregex Pattern
IC1	((SBAR < (WHADVP $+ (S < ((VP < (VBG $+ NP | $+ PP)) !$++ NP !$– NP)))) !> > VP)
IC2	((PP < ((IN < Upon) $+ (NP < ((NP < < NN) $++ PP)))) !> > /(VP | SBAR)/)

S: Clause, SBAR: Subordinate clause, WHADVP: Wh-adverb phrase, VP: Verb phrase, VBG: Verb gerund, NP: Noun phrase, PP: Prepositional phrase, IN: Preposition, and NN: Noun

Table 4.6: Tregex pattern to detect incomplete conditions

that contained similar examples, such as EIC1 and EIC2. Next, we grouped the requirements into two sets. Each set shared the same information content. Then, for each set, we derived a Tregex pattern. Table 4.6 shows the two derived patterns, IC1 and IC2.

In the example shown in Figure 4.3 (Step 3a), our approach matched the pattern IC2 with segment 001 "Upon reception of a settlement instruction from System-A". The condition misses the actor and the verb; therefore, the smell "Incomplete condition"was triggered.

Glossary search. This method aims to identify the smell: "Not a precise verb". For this purpose, we created a glossary of verbs that do not describe a precise action. For example, according to the English dictionary, the verb "process" means "operate on (data) by means of a program". This verb does not provide a precise action, which makes it difficult for an analyst to test the requirements that contain such a verb. To create our glossary of verbs, we gathered all the verbs of the requirements used in Section 4.2.1. We searched for verbs that did not have a precise action and were difficult to test. Our glossary includes the following verbs: accomplish, account, base, come, consider, default, define, do, get, make, perform, process, propose, make, raise, read, support, and want. Our list includes also verbs that have several meanings but only one etymology (polysemy). These verbs in our glossary are: come and get.

We have elaborated on our glossary in collaboration with two experts working for our industrial partner. The experts agreed that they would prefer to avoid using these verbs when specifying requirements as they are not precise enough and are indeed difficult to test.

To detect these verbs in the requirements, our approach automatically extracts verbs from the requirement segments condition and system response. Next, our approach obtains the lemmas of these verbs. A lemma is the base form of a verb. Finally, our approach searches for the lemma in our glossary. If there is a match, our approach triggers the smell "Not a precise verb".

In the example shown in Figure 4.3, our approach only analyzes the segment 002 and 003 because they are segments that are system responses; then, we extracted the verb "process" because it belonged to our glossary. The smell "Not a precise verb" was triggered.

4.3.4 Step 4. Suggesting Rimay Patterns

This section aims to answer **RQ3: How can we suggest templates to improve requirement quality?** For this purpose, our approach analyzes the segments of the requirements identified in the previous steps to match one of the 10 Rimay patterns (Section 4.2.2). The suggested pattern will guide analysts to fix any smell detected in requirements and convert them into Rimay requirements.

To identify a suitable Rimay pattern, our approach first computes the frequencies of the segments of the requirements. More concretely, our approach counts the number of segments scopes, conditions, and system responses that appear in a requirement. Furthermore, for the segments that are conditions, our approach further classifies them into trigger, time, and precondition (Section 3.4) and computes the

frequency for each type of condition. To identify a "time condition", our approach checks if the segment type was matched by the pattern "C9" (Table 4.3) in Step 2. Furthermore, to identify "precondition condition", we extract from the condition segments the verb phrase (VP). If the VP contains one of the operators "is equal to", "less or equal to", "contain", and "have" (Chapter 3), then the type is a "precondition condition". Moreover, to identify trigger conditions, we extract the VP from the condition. If the VP contains verbs other than the ones used by the "precondition condition", then it is a "trigger condition". We also consider the frequencies of incomplete segments. The incomplete segments are the results of detecting the smells "incomplete scope", "incomplete condition", and "incomplete system response". Once the frequencies are calculated, we map the frequencies to any of the 10 Rimay patterns. Table 4.7 indicates the frequency of segments for each of the 10 Rimay patterns. The first column indicates the name of the pattern. The second through sixth columns show the frequency of each segment contained in each of the 10 Rimay patterns.

In the example shown in Figure 4.3, Step 4, we analyzed the segments in the requirement. The requirement had segment 001 as an "Incomplete condition", segment 002 as a "System response", and segment 003 as a "System response". Segment 001 is a condition. Our approach could not detect any verb in segment 001; however, the Tregex pattern that identified the smell "Incomplete condition" suggests that this type of condition uses a noun "reception" instead of the verb, which suggests that the verb is an action verb, suggesting that it is a "condition trigger" To summarize, the requirement in Figure 4.3 (Step 4) has a "condition trigger" and two system responses. Since Rimay discourages analysts to writing non-atomic requirements, then our approach suggests Rimay pattern "7. Condition(Trigger) and system response".

Table 4.7: Rimay patterns by segment frequency

Pattern	**Scope**	**Condition**			**System Response**
		Pre-condition	**Trigger**	**Time**	
1. Scope and system response	1				1
2. Scope, condition (precondition), and system response	1	1			1
3. Scope, condition (trigger), and system response	1		1		1
4. Scope, condition (time) and system response	1			1	1
5. System response					1
6. Condition(precondition) and system response		1			1
7. Condition (trigger) and system response			1		1
8. Condition (time) and system response				1	1
9. Scope, multiple conditions, and system response	1	2 or more			1
10. Multiple conditions and system response		2 or more			1

4.4 Evaluation

In this section, we describe the case study conducted to address RQ4 and RQ5. We follow best practices for reporting case study research in software engineering [56].

4.4.1 Case Study Design

Our evaluation aims to answer the following RQs:

RQ4: Can our approach correctly indicate the occurrence of smells?

RQ5: How accurate is our approach in recommending requirement templates to fix smells?

To answer RQ4 and RQ5, we measured the performance of our approach in detecting smells and suggesting Rimay patterns against a human-annotated ground truth. We constructed our ground truth (GT) as follows: (1) we collected five new SRSs from our industrial partner (we refer to this set as S), (2) an external annotator analyzed the syntax and semantics of each requirement of the set S to detect smells and assign a Rimay pattern, and (3) we monitored the annotation results each time the annotator completed an SRS via in-person sessions with the annotator. In each session, we discussed the difficulties encountered while annotating the requirements and propose solutions to correct such difficulties. In addition, we validated the overall annotations by reviewing a random set of 10% of the annotations. In case of anomalies, we asked the annotator to revise the annotations.

Once the ground truth was completed and validated, we conducted our evaluation using an iterative and sequential approach. We divided set S into two batches. Table 4.8 shows the distribution of the batches of set S. First, we applied our approach to the first batch of S to detect smells and suggest patterns. Second, we compared our results against the ground truth by computing precision and recall. Third, if the approach performed less than 80% (overall precision and recall), we analyzed cases showing disagreements with the ground truth. Fourth, we improved our approach to correcting disagreements with the ground truth. Finally, we applied the enhanced version of our approach to the second batch.

Table 4.8: S batch distribution

Batch #	SRS ID	# Requirements
1	SRS1	294
	SRS2	162
2	SRS3	196
	SRS4	340
	SRS5	150
Total		1142

4.4.2 Data Collection and Preparation

To build our ground truth GT, we first collected data from our industrial partner, Clearstream. Financial analysts from Clearstream provided us with a set of five representative SRSs. Each SRS contained a different number of requirements. These SRSs described different types of projects, including updates to existing applications, compliance of the applications with new regulations, creation of new applications, and description of the migration of existing applications to new platforms. These SRSs contained requirements that were not observed during the creation of our approach (Section 4.2). We name the SRSs SRS1-SRS5 and refer to them as set S in this section. They contain 1142 requirements in total. Second, we applied domain-specific preprocessing steps to the requirements of set S. We discussed these preprocessing steps in Section 4.3.1. Third, an external annotator, with a background in requirements

engineering and more than three years of experience, conducted a 180-hour annotation process. The annotator manually analyzed the syntax and semantics of each requirement in SRSs of set S to detect smells and assign a Rimay pattern. After the annotator completed annotating each SRS, we monitored the annotation results by having a monitoring session (30-60 minutes). In each session, the annotator pointed us to the requirements that were difficult to annotate. We then discussed them to reach an agreement on the correct annotations. Once the annotation process concluded, we randomly selected 10% of the requirements annotated in S for inspection. From the analysis results, we found that most of the annotations (more than 80%) were satisfactory. As for the errors (less than 20%), we identified their causes and asked the annotator to correct them throughout all SRSs in S. We then accepted the annotations.

4.4.3 Collecting Evidence and Results

This section describes the raw data collected in the case study. Table 4.9 provides the annotation results of the smells detected in set S. The first column indicates the smell name. The second through sixth columns present the number of requirements containing the smell listed in each row for each SRS in S. The last column displays the total number of occurrences of each of the 10 smells in set S. From Table 4.9, we can see that "Passive Voice" is the smell with the highest frequency in S (accounting for 30.8% of the requirements). In contrast, we have smells such as "Not a requirement" and "Incomplete scope", which are absent from set S. However, these smells were observed during their derivation (Section 4.2.1).

Table 4.9: Smells - Annotation results for set S

Smell	SRS1	SRS2	SRS3	SRS4	SRS5	Total
1. Non-atomic	80	24	26	69	11	210
2. Incomplete requirement	0	2	0	0	0	2
3. Incorrect order requirement	13	7	6	9	7	42
4. Coordination ambiguity	11	5	16	3	4	39
5. Not a requirement	0	0	0	0	0	0
6. Incomplete condition	56	17	68	87	19	247
7. Incomplete SR	4	2	0	0	4	10
8. Incomplete scope	0	0	0	0	0	0
9. Passive voice	54	78	29	59	82	302
10. Not precise verb	4	30	55	1	38	128

Table 4.10 shows the frequencies of Rimay patterns assigned by the annotator to the set of requirements S. The first column shows the Rimay pattern. The second through sixth columns present the number of requirements assigned to each Rimay pattern for each SRS in S. The last column displays the total number of occurrences of each of the 10 Rimay patterns in set S. It is apparent from the data in Table 4.10 that "5. System response" is the most frequently assigned Rimay pattern. This pattern was suggested 314 times in set S. Least frequently assigned Rimay patterns (less than ten times) are "1. Scope and system response", "3. Scope, condition(Trigger), and system response" and "8. Condition (Time) and system response". In Table 4.10, we can also see Rimay patterns that were not suggested to any requirements in set S, i.e., "2. Scope, condition (precondition), and system response " and "4. Scope, condition (Time) and system response".

Table 4.10: Rimay patterns - Annotation results for set S

Rimay Pattern	SRS1	SRS2	SRS3	SRS4	SRS5	Total
1. Scope and system response	1	0	0	0	3	4
2. Scope, condition (precondition), and system response	0	0	0	0	0	0
3. Scope, condition (trigger), and system response	1	1	0	0	0	2
4. Scope, condition (time), and system response	0	0	0	0	0	0
5. System response	2	218	14	52	28	314
6. Condition (precondition) and system response	16	36	1	50	3	106
7. Condition (trigger) and system response	31	142	63	33	13	282
8. Condition (time) and system response	0	3	0	0	0	3
9. Scope, multiple conditions, and system response	0	0	0	0	0	0
10. Multiple conditions and system response	32	28	13	32	13	118

4.4.4 Analysis of Collected Data

This section assesses the accuracy of our approach to detecting smells in NL requirements (RQ4) and suggesting Rimay patterns to analysts (RQ5). We applied our approach to the set of requirements S to detect smells and suggest Rimay patterns. We compared these results against GT (Section 4.4.1) by computing precision and recall metrics. For this purpose, we first classified the predictions of our approach into the following categories:

True positives (TP) are the correct predictions. In smell identification, a TP occurs if we detect the same smell as the ground truth. In pattern suggestion, a TP occurs if a requirement is assigned to the same Rimay pattern as GT.

False negatives (FN) are missed annotations. In smell detection, a missed annotation occurs when we incorrectly indicate that the requirement does not have a smell in contrast to GT. In pattern suggestion, a missed annotation occurs when we do not suggest any pattern in contrast to GT.

False positives (FP) are misclassified annotations. In smell identification, an FP occurs when our approach incorrectly indicates the presence of a smell. In pattern suggestion, this occurs when we incorrectly suggest a Rimay pattern.

Next, for each smell and Rimay pattern, we calculated the precision (P) as $P = \frac{TP}{TP+FP}$ and the recall (R) as $R = \frac{TP}{TP+FN}$. Furthermore, we calculated the overall precision as `Overall-P` $= \frac{\sum_{i=1}^{l} TP_i}{\sum_{i=1}^{l}(TP_i+FP_i)}$ where for smell detection i takes the values from Smell 1 to Smell 10 and for pattern suggestion, i takes the values from Rimay pattern 1 to Rimay pattern 10. The overall recall was calculated as `Overall-R`$= \frac{\sum_{i=1}^{l} TP_i}{\sum_{i=1}^{l}(TP_i+FN_i)}$ where, for smell detection, i takes the values from Smell 1 to Smell 10 and for pattern suggestion, i takes the values from Rimay pattern 1 to Rimay pattern 10.

Table 4.11 shows the P and R results that we obtained to detect smells (RQ4). These results were calculated for the evaluation set S and compared against the ground truth GT. The first column of Table 4.11 shows the smell name. The second and third columns show the precision (P) and recall (R) values for each of the 10 smells found in the first batch. The fourth and fifth columns show the precision (P) and recall (R) values for each of the 10 smells found in the second batch. The sixth and seventh columns show the overall precision (P) and recall (R) values for each of the 10 smells found in set S. The last row of Table 4.11 shows the overall P and R values for batches one, two, and overall.

To follow our evaluation design (Section 4.4.1), we first applied our approach to the first batch of S to detect smells and compare our results with GT. Our approach obtained P and R values below 80% and

were deemed unsatisfactory. To address this, we analyzed the misclassified (FP) and missed annotations (FN). Most of the FPs and FNs cases resulted from new scenarios that were not considered during the creation of our approach. We improved it to support these new scenarios and detected smells in batch one with a precision of 88% and a recall of 84% (Table 4.11). As shown in Table 4.11, a decrease of 19% in the recall value in the second batch affected the overall recall value of our approach. In Section 4.5, we provide information on the reasons for the decrease in recall.

Table 4.11: Smell detection - Performance on set S

Smell	**Batch 1**		**Batch 2**		**Overall**	
	P	R	P	R	P	R
1. Non-atomic	0.86	0.93	0.85	0.88	0.85	0.91
2. Incomplete requirement	0.67	1.00	0.33	1.00	0.44	1.00
3. Incorrect order requirement	0.70	1.00	0.75	0.95	0.73	0.97
4. Coordination ambiguity	1.00	1.00	0.92	0.50	0.92	0.52
5. Not a requirement	N/A	N/A	N/A	N/A	N/A	N/A
6. Incomplete condition	0.83	0.76	0.82	0.45	0.83	0.54
7. Incomplete SR	1.00	0.83	0.62	0.62	0.77	0.71
8. Incomplete scope	N/A	N/A	N/A	N/A	N/A	N/A
9. Passive voice	0.96	0.76	0.95	0.81	0.95	0.78
10. Not a precise verb	1.00	0.91	1.00	0.61	1.00	0.68
Overall	0.88	0.84	0.88	0.68	0.88	0.74

Table 4.12 shows the P and R scores that our approach obtained for suggesting precise Rimay patterns to analysts (RQ5). The results were calculated by applying our approach to the evaluation set S and compare the results to the ground truth GT. Table 4.12 shows for all Rimay patterns (column 1) the P and R values for the patterns suggested in the first batch (columns 2-3), second batch (columns 4-5) and overall (columns 6-7). The last row of Table 4.12 shows the overall scores for P and R for batches one, two, and the overall set. We applied our approach to the first batch of S to find suitable Rimay patterns and compared our results with GT. The results were satisfactory. We obtained a P of 90% and an R of 85% in batch 1. Given that these values were above 80%, we did not further enhance our approach. In terms of the overall accuracy, the suggestions for Rimay patterns have an overall P of 89% and an R of 82% (Table 4.12).

From the data in Table 4.12, we can see that the P and R values in batch two decreased compared to batch one, which affected the overall performance. We discuss the reasons for such a decrease in P and R in Section 4.5.

From our experience, we believe that, in practice the acceptance of a new tool is highly dependent on obtaining correct predictions. Therefore, precision plays a particularly important role in enhancing user acceptance. If a tool provides incorrect findings too often, users will tend to lose confidence in the approach and will eventually stop using it. In our evaluation results, we observed that our approach obtained a high overall precision score (smell detection 88%, and pattern suggestion 89%), suggesting that our results are promising and likely to foster acceptance by our industrial partner.

Table 4.12: Performance pattern suggestion dataset S

Rimay Pattern	Batch 1		Batch 2		Overall	
	P	**R**	**P**	**R**	**P**	**R**
1. Scope and system response	1.00	1.00	1.00	0.44	1.00	0.57
2. Scope, condition (precondition), and system response	N/A	N/A	N/A	N/A	N/A	N/A
3. Scope, condition (trigger), and system response	0.98	0.97	0.95	0.95	0.97	0.96
4. Scope, condition (time) and system response	N/A	N/A	N/A	N/A	N/A	N/A
5. System response	1.00	1.00	0.92	1.00	0.93	1.00
6. Condition (precondition) and system response	0.00	0.00	0.67	0.67	0.50	0.57
7. Condition (trigger) and system response	0.75	0.76	0.72	0.76	0.73	0.76
8. Condition (time) and system response	N/A	N/A	0.83	0.83	0.83	0.83
9. Scope, multiple conditions, and system response	0.90	0.92	0.85	0.72	0.88	0.85
10. Multiple conditions and system response	0.94	0.77	0.96	0.76	0.95	0.76
Overall	0.90	0.85	0.88	0.80	0.89	0.82

4.5 Discussion

4.5.1 Approach Performance

The results presented in Section 4.4 show that our approach to answering RQ4 is accurate in terms of detecting smells in NL requirements (P = 88%). However, we observed that our approach achieved a low precision score for the detection of a particular smell, "Incomplete Requirement" (22%), and a low recall score for detecting the smells "Coordination ambiguity" (52%)" and "Incomplete condition" (52%). To determine the root causes of the low precision and recall obtained by our approach, we analyzed the misclassified and missed annotations for each of the smells mentioned above.

Incomplete Requirement. Recall from Section 4.2.1 that this smell occurs when the requirement misses a system response. We observed that misclassified annotations were related to inaccurate POS tags assigned to the verb of the system response. Table 4.13 shows requirement R1. The verb of the system response of R1 (i.e., route) was incorrectly identified as a noun. Since no verb was found in the requirement, our approach triggered the smell "Incomplete requirement."

Coordination Ambiguity. Recall from Section 4.2.1 that this smell occurs when a requirement has more than two conditions and the conditions have at least one connector ("or") as a separator. We observed that missed annotations were related to new scenarios that were not observed during the creation of our approach or during updates made after Batch 1. An example of a new scenario is shown in Table 4.13 (requirement R2). This requirement is composed of three conditions that are connected by an "or" connector. Two out of the three conditions for R2 have the symbols "< >" instead of a verb. The methods applied in Step 4.3.2 of our approach do not have a scenario that identifies this syntax to express conditions. Therefore, our approach did not recognize this condition and thus did not apply the method that checks for the smell "Coordination ambiguity."

Incomplete Condition. Recall from Section 4.2.1 that this smell occurs when the condition of the requirement misses the verb or the actor. We observed that the main cause of the misclassifications was the assignment of incorrect POS tags to the verbs in the condition. The absence of the verb in the condition triggers the smell "Incomplete Condition". Regarding missed annotations, we observed that the main cause was related to scenarios not observed during the creation of our approach or during the updates made after Batch 1. Table 10 shows an example (R3) of a missed annotation. R3 misses a verb and

instead has the symbol "=," which denotes "equals to." Furthermore, R3 is made up of a compound noun, "System-A Order Issuer Ordering data." The word "Order" of the compound noun is identified as a verb by the Post Tagger, which suggests that the condition is a complete condition. Therefore, our approach did not trigger any smells. However, in reality, R3 misses a verb.

Similarly, the results of RQ5 show that our approach accurately suggests requirement patterns in most cases. However, we noted that our approach obtained low recall when suggesting requirement pattern "1. Scope and system response" (57%) and obtained low precision and recall when suggesting pattern "6. Condition (precondition) and system response" (P = 50% and R = 57%). To determine the root causes of the low precision and recall obtained by our approach, we analyzed cases in which our approach had misclassified and missed annotations.

Pattern: 1. Scope and system response. This pattern is suggested when a requirement has the following segments: scope and system response. The obtention of missed annotations was related to new scenarios that were not observed during the creation of our approach. Requirement R5 in Table 10 is an example of a missed annotation. R5 has a scope phrase followed by additional information that should be in a condition. Having this additional information in the phrase scope prevented our approach from properly distinguishing the phrase scope; therefore, it did not provide a precise Rimay pattern.

Pattern: 6. Condition (precondition) and system response. This pattern is suggested when a requirement has the following segments: a condition of type precondition and a system response. The misclassifications were related to new scenarios that were not observed during the creation of our approach. Requirement R6, in Table 4.13, is an example of a missed annotation (R6). The condition of R6 lacks a verb; instead, R6 has the operator "=". The missing verb prevented our approach from recognizing the condition and providing an accurate suggestion.

In summary, we identified two main reasons why our approach obtained lower precision and recall in the cases mentioned above. The first is POS Tagger limitations: POS Tagger incorrectly assigns POS tags to words, which causes our approach to fail to correctly identify the smells and syntax of the requirement. Our approach does not have control over the accuracy of the POS Tagger, since it is a third-party component. Concerning new scenarios, we found several that were not previously observed. These scenarios include different structures of the requirement segments and the usage of symbols instead of verbs. If analysts agree, we can enhance our tool to support these new scenarios. However, some of these scenarios are examples of bad practices in specifying requirements. For example, in requirement R2 of Table 4.13, the analyst has used symbols instead of verbs.

Table 4.13: Example of missed and misclassified annotations.

ID	Requirement Description
R1	On receipt of a valid C01 cancellation from System-A Participant, then the System-B must route the cancellation to the same destination
R2	If the Property-A < > 0 or the Property-B = 0 and the Property-C 1 < > 0 , then...
R3	if the System-A Order Issuer Ordering data = Value-A
R4	When the value of Property-1 changes in System-A, System-A must ...
R5	- For each portfolio ID associated to an Entitiy-A Participant with a Service Provider of System-A, then ...
R6	If the Report type = Statement of Orders Aggregated , then ...

4.5.2 Lack of Testing Data

We observed that the results of the evaluation of RQ4 contained smells that were not evaluated (i.e., "Not Requirement" and "Incomplete Scope"). Similarly, the results of RQ5 show that our approach was not tested to detect the Rimay patterns "2. Scope, condition (precondition), and system response" and "4. Scope, condition (Time), and system response." We were unable to test our approach in the above cases because the evaluation set lacks requirements that contain the smells "Not Requirement" and "Incomplete Scope". Moreover, the requirements did not have the syntax to suggest Rimay patterns: "2. Scope, condition (precondition), and system response" and "4. Scope, condition (Time), and system response". As described in Section 4.4.2, the SRSs used to evaluate our approach were collected from our industrial partners. Our industrial partner provided us with a set of five SRSs over which we had no control. However, the aforementioned cases were tested during the creation of the approach.

4.6 Threats to Validity

Internal validity focuses on confounding factors. A threat to internal validity is related to the potential biases introduced during the empirical evaluation. To reduce bias, we delegated the annotation task of ground truth (Section 4.4) to trained third-party annotator. The annotator did not have access to our approach; therefore, the annotator was not influenced by the results of our approach.
External validity refers to the generalizability of the results of the case study. Despite the fact that our evaluation used SRSs that contain NL requirements from the financial domain, the requirements are representative of a broader class of information systems, such as those in the banking and securities industries. Nonetheless, future investigations are necessary to determine whether and how our approach can be applied to other domains and information systems.

4.7 Related Work

In this section, we present existing work related to improving the quality of NL requirements. The literature groups existing work into three categories of research [37].
The *first category* describes tools that aim to help business analysts reduce common semantic and syntactic problems [61] found in NL requirements. These tools identify problems in NL requirements and provide guidance on how the quality of requirements can be improved.
The *second category* includes approaches that transform NL requirements into formal models (e.g., object-oriented analysis models) and logic specification languages. Transformations are typically performed by a linguistic analysis of NL requirements.
The *third category* discusses tools that automatically classify NL requirements according to their good or bad overall quality, for example, ambiguity detection. These tools use methods based on machine learning or rule-based learning.

This chapter discusses how we built a tool that detects 10 smells that are commonly present in requirements for financial applications. In addition, we guide analysts through the transition from unrestricted requirements to Rimay (Section 3.4) requirements as a way to fix requirements that contain smells and improve requirement quality. Given our objective, our tool falls into the first category of research mentioned above. In the following section, we consider recent studies that have discussed

approaches to identifying smells in requirement specifications and provide directions for improving the quality of the requirements. We considered four studies relevant to our work: two were concerned with identifying errors on NL requirement specifications, while the other two identified errors on feature requests and use-case descriptions.

The work proposed by Osama et al. [50] identified three smells to detect attached ambiguity, coordination ambiguity, and analytical ambiguity in NL requirements. To correct the ambiguity identified in the requirements, the authors provide a tool that assists the user by providing unambiguous interpretations. Seki et al. [60] proposed a set of 54 smells aimed at detecting quality problems in use-case descriptions. These smells help locate problems related to ambiguity, incorrectness, redundancy, lack, misplacement, and inconsistency. The authors provide a tool that checks the aforementioned problems in NL requirements written in Japanese. Mu et al. [48] identified 10 smells to detect problems in feature requests. These smells are related to problems of ambiguity and incompleteness and incomprehensibility (unable to be understood or comprehended). The authors provide a tool that highlights the text in the event of containing any of the 10 smells. Femmer et al. [21] proposed an approach that detects nine smells in NL requirements. The smells are related to ambiguity and incompleteness problems.

In summary, Osama et al. and Femmer et al. [50, 21] detected smells in NL requirements while Seki et al. and Mu et al. [60, 48] detected smells in feature requests and use-case descriptions. The smells proposed by the studies above tackle problems in NL specifications that are related to ambiguity, incorrectness, inconsistency, misplacement, redundancy, incompleteness, and incomprehensibility. Furthermore, all the above studies provide a tool that helps business analysts automatically detect smells. Only the study proposed by Osama et al. [50] provided users with possible solutions to fix the smell found in the NL requirements. In contrast, apart from detecting smells, our approach suggests appropriate Rimay patterns to fix any detected smells and converts the requirement into a Rimay requirement. Moreover, 7 out of 10 smells detected by our approach are not proposed by any of the existing works. These smells are “Incomplete Scope,” “Incomplete System Response,” “Incomplete Condition,” “Not Requirement,” “Incorrect Order Requirement,” “Incomplete Requirement,” and “Non-atomic.”

4.8 Conclusions

The goal of this chapter was to better support business analysts in the specification of NL requirements by detecting smells in NL requirements and to guide them in fixing detected smells. To achieve these objectives, we propose a set of 10 smells that represent the most common syntactic and semantic errors found in NL requirements from financial applications. Furthermore, we derived 10 Rimay patterns. These patterns aim to fix the smells present in NL requirements and convert NL requirements into Rimay requirements. We then proposed an automated approach that automatically detects our proposed smells in NL requirements and suggests Rimay patterns to improve the overall quality of NL requirements.

After developing this approach, we tested it in an industrial case study. This evaluation measured the performance of our approach in detecting smells and suggesting accurate Rimay patterns. We evaluated our approach using a set of 1142 human-annotated NL requirements that contained smells. Over this set, our approach detected smells with a precision of 88% and a recall of 74%. Furthermore, our approach suggested a Rimay patterns with a precision of 89% and a recall of 82%.

In future work, we intend to expand our list of smells to provide broader coverage of smell detection. Our proposed smells tackle common smells found in the NL requirements of financial applications, but they do not represent all syntactic and semantic errors present across all NL requirements. Furthermore, it will be important to conduct a user study of the usefulness of our approach. This proposed study would assess in a more conclusive manner whether business analysts benefit from our approach to improving the quality of NL requirements. Finally, we intend to integrate our approach into an existing and widely known modeling and code-generation tool, Sparx Systems Enterprise Architect, because Enterprise Architect was already being used by our industrial partner.

Chapter 5

Leveraging Natural-language Requirements for Deriving Better Acceptance Criteria from Models

5.1 Motivations and Contributions

Acceptance testing is aimed at determining whether a system under test (SUT) meets its specified requirements [3]. A key step in acceptance testing is defining the Acceptance Criteria (AC) for the SUT. AC are conditions that the SUT must satisfy in order for the SUT to be accepted by its users or customers. Naturally, AC are derived from the requirements. It is desirable to make the AC derivation process as automated as possible, noting that, without automated support, it would be very tedious for the analysts to define the AC in a systematic and complete manner. This is specially true for complex systems with large numbers of requirements and for systems whose requirements evolve frequently.

An important complexity in automating the derivation of AC has to do with the fact that the requirements of an SUT may have been expressed using heterogeneous representations. Notably, our experience with several industry domains, including automotive, telecommunications and finance, indicates that analysts tend to specify their requirements using a *combination* of models and natural-language (NL) statements. These two modes of representation tend to provide complementary and yet overlapping information. When models (e.g., UML models) and NL statements are used simultaneously for specifying the requirements, the derivation of AC necessarily has to account for the requirements expressed in both representations.

To illustrate, we present a (highly simplified) example from the financial domain, involving one model and one NL requirement.

Model: Figure 5.1 presents a UML Activity Diagram related to setting up an order for purchasing bonds. If the (order) data is correct, an order is created; otherwise, an alerting process kicks in to notify the bond trader about the data anomaly.

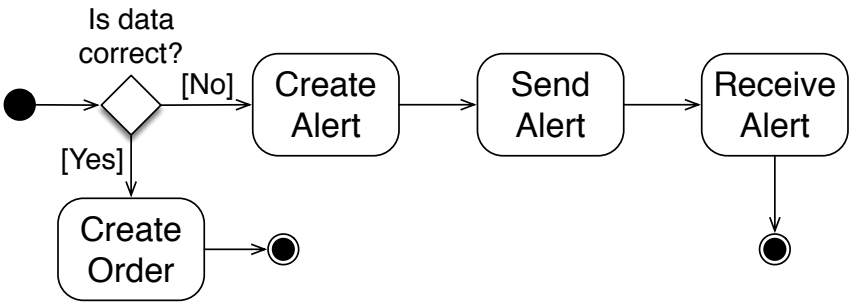


Figure 5.1: Example of a (requirements) model

NL Requirement: The NL requirement is as follows:

```
R: When System_A creates an alert, then System_A must set the priority of
the alert to "high".
```

This NL requirement is concerned with the behavior that is expected when an alert is created. As we are going to discuss in Section 5.2, in this chapter, we use an existing controlled language, named Rimay (Chapter 3), for writing NL requirements. The above-stated requirement, R, complies with Rimay's grammar.

AC Derivation: To generate AC, we employ an existing technique, named AGAC (Automated Generation of Acceptance Criteria) and its associated tool [2]. The AC produced by AGAC are represented in the Gherkin scenario language [76]. Gherkin scenarios follow a predefined (textual) template: *Given* [initial context], *When* [event or action], *Then* [expected result]. In AGAC, each acceptance criterion is captured by means of a sequence of Gherkin scenarios. Each such sequence exercises an end-to-end system behavior, starting from the initial node of an Activity Diagram and going all the way to a final node. Figure 5.2 shows the AC, AC 1 and AC 2, that one would intuitively expect for exercising the two alternative flows of the model in Figure 5.1[1]. To save space, we have truncated AC 2 by hiding the Gherkin scenarios induced by the *Send Alert* and *Receive Alert* actions in the model of Figure 5.1. In Figure 5.2, Gherkin's keywords are in bold. The fixed, predefined text coming from AGAC's AC templates is in regular black font. The text obtained from the model of Figure 5.1 or the NL requirement R is in blue.

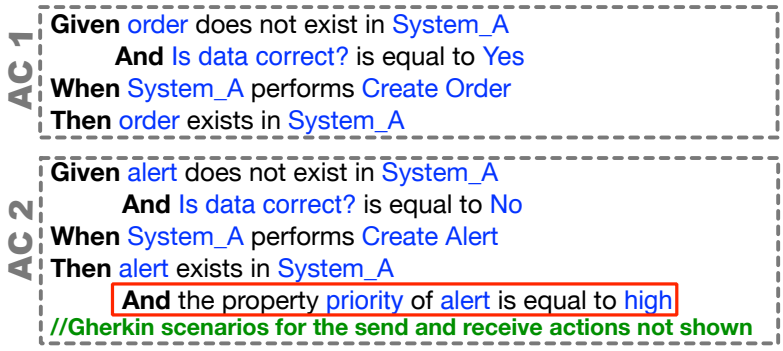


Figure 5.2: The generated AC

The generated AC; without considering the NL requirement R, the segment (post-condition) in AC 2 shown by a red box will not be generated

[1]AGAC supports AC generation from a model with (1) parallel flows via standard depth- and breadth-first search [14] to traverse non-concurrent and concurrent nodes, and (2) loops via a bounded unwinding of the loops. These are important considerations for deriving end-to-end AC, but are orthogonal to our illustrating example.

The example in Figure 5.2 highlights the fact that neither models nor NL requirements provide a complete picture of what is relevant to AC generation. Specifically, one cannot expect to find in the NL requirements alone all the information that is pertinent to AC. Notably, control-flow behaviors, e.g., the ordering of the actions in our example model of Figure 5.1, are often entirely absent from the NL requirements. The result is that, based on NL requirements alone, one typically cannot synthesize end-to-end system behaviors; exercising these behaviors is, however, critical to acceptance testing.

On the other hand, NL requirements often provide fine-grained details that analysts would not normally include in the model. In our example, the analysts, for instance, found it more convenient to use NL requirements to express the data properties of objects, e.g., the NL requirement `R` for the alert object. Without considering `R`, the post-condition marked in Figure 5.2 with a red box (i.e., *"the property priority of alert is equal to high"*) cannot be inferred, thus leaving AC 2 incomplete.

Our work in this chapter is prompted by the observation that a reconciliation of the information content in models and NL requirements is necessary for deriving precise and complete AC. We propose such an automated reconciliation approach and tool. The main idea behind our work is to *make models the central repository of information for the generation of AC*. To be able to do so, we need to devise a technique that can *enrich* models with information that is otherwise exclusively available in NL requirements.

The chapter investigates three Research Questions (RQs):

RQ1: How can we extract AC-related information from NL requirements? We answer RQ1 by defining a rule set composed of 13 information extraction rules that automatically extract AC-related information from NL requirements (first contribution). We identified these rules by analyzing the conceptual overlaps and distinctions between the element types in models and the element types in NL requirements, with a focus on information-system domains such as finance.

RQ2: How can we systematically enrich models with the (AC-related) information from NL requirements? We answer RQ2 by proposing a systematic method that generates recommendations for model enrichment, based on the information extracted by the rules developed in response to RQ1 (second contribution). The method identifies the model elements that can be enriched with the extracted information and provides guidance to the analysts as to how they can incorporate this additional information into the model. Subsequently, an existing model-based AC derivation technique, AGAC [2], is applied to the enriched model. This way, we make it possible to account for the information in *both* the model and the NL requirements during AC derivation.

RQ3: Are our recommendations for model enrichment useful in practice? We answer RQ3 through an industrial case study conducted in collaboration with a leading financial-services provider (third contribution) hereafter, referred to as our industrial partner. We applied our model enrichment method to this case study; this resulted in 27 recommendations for model enrichment. The study involved a group of five domain experts. The experts were asked if the recommendations were relevant to AC. Out of the 27 recommendations, 24 were deemed relevant by the experts (precision of 89%). The experts did not identify any additional AC-relevant information in the NL requirements which had not already been brought to their attention by the recommendations (recall of 100%).

The rest of this chapter is structured as follows: Section 5.2 provides background. Section 5.3 presents an overview of our model enrichment approach. Section 5.4 explains and illustrates the details of the approach. Section 5.5 reports on the evaluation of the approach in an industrial setting. Section 5.6 discusses threats to validity. Section 5.7 compares with related work. Section 5.8 concludes the chapter.

5.2 Background

5.2.1 Writing NL Requirements in Rimay.

Rimay, introduced in Chapter 3, is a CNL for writing requirements in the domain of information systems, initially validated in the financial domain. We use Rimay because: (1) it is a suitable match for our case study context. Indeed, our industrial partner is already using Rimay to write NL requirements for some of their banking and securities applications; (2) as a CNL, Rimay comes with precise syntax and semantics. The first characteristic ensures that one has enough expressive power to capture the NL requirements in our case study. The second characteristic enables us to devise structured and highly accurate rules for extracting AC-related information from NL requirements, thereby alleviating the need for heuristics based on natural language processing and machine learning, which are typically less accurate.

Rimay's main grammar rules are inspired by the Easy Approach to Requirements Syntax (EARS) templates [44]. EARS is considered by many practitioners to be a good trade-off between flexibility and precision, due to EARS' relatively low training overhead and the quality and readability of the resultant requirements [43].

The rule `REQUIREMENT` shown in Listing 5.1 provides the overall syntax for a requirement in Rimay. The rule shows that the presence of the `SCOPE` and `CONDITION_STRUCTURES` is optional, but the presence of an `ACTOR`, `MODAL_VERB` and a `SYSTEM_RESPONSE` is mandatory in all requirements. Refer to Chapter 3 for a complete reference to the concepts and constructs of the Rimay language.

```
REQUIREMENT: SCOPE? CONDITION_STRUCTURES? ARTICLE? ACTOR MODAL_VERB not?  SYSTEM_RESPONSE.
```

Listing 5.1: Overall syntax of a requirement in Rimay

5.2.2 Automated Generation of AC

We use the AGAC approach and its associated tool [2] for deriving AC from models. AGAC is an Activity Diagram-centered approach and consists of two tasks: (1) *Create Specifications* and (2) *Derive AC*. The first task, which is performed manually, is concerned with the creation of a requirements and analysis model by following an existing modeling methodology [2]. The resulting model include Activity Diagrams (ADs), Class Diagrams (CDs) and Use Case Diagrams (UCDs). *Actors* are defined in UCDs and execute *actions* that are part of the *activities* represented in ADs. *Domain entities* and their properties are characterized by a *domain model*, represented using CDs and referenced within the ADs. To enable automated AC generation, AGAC requires that analysts specify the intent of the actions in the ADs using 11 predefined stereotypes (Create, Read, Update, Delete, Send, Receive, Enable, Disable, Display, Not Display, and Validate). The intent type that is ascribed to a given action captures the nature of the observable behavior of the action, thus allowing the derivation of suitable criteria to exercise the behavior. The intent of a given action does not always need to be explicitly declared; AGAC can automatically infer the intent for certain actions. For instance, the Create stereotype is assigned automatically to an action when (1) the output edge of that action is connected to a domain entity with an identifier that has not been already encountered when processing previous actions, or (2) the name of that action starts by "Create" or one of its synonyms. For example, the *Create Order* action in Figure 5.1 (discussed in Section 5.1) will automatically receive the Create stereotype because its name starts by "Create".

The second task, *Derive AC*, is automated. This task matches the model created in the first task to a set of predefined AC templates, based on the intents of the actions in the model. The appropriate templates are then instantiated (potentially multiple times), producing AC represented in the Gherkin language [76]. More specifically, the templates define the fixed parts of the text in the Given-When-Then structure of a Gherkin scenario as well as the variable parts (placeholders) that need to be filled with content from the model. What enables the identification of the appropriate template for a given action is the tag `@Intent`, which is specified in every AC template. Gherkin's popularity can be attributed in large part to its capability to enforce the use of high-level, domain-specific terms, as well as to support traceability from AC to executable test cases [76].

5.3 Approach Overview

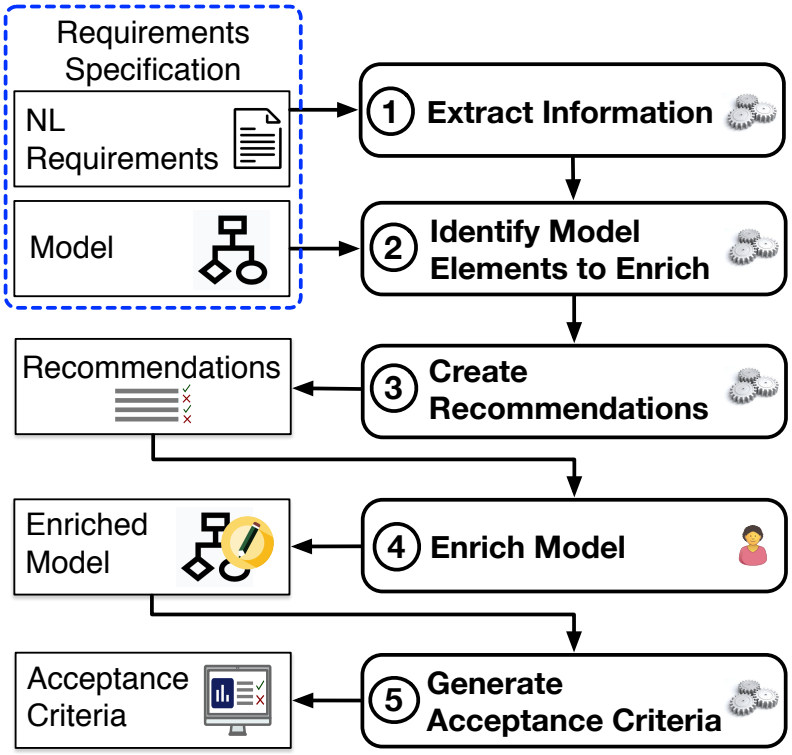


Figure 5.3: Approach overview

In this section, we introduce the inputs to our approach, followed by an overview of the different steps of the approach (Figure 5.3).

The input to the approach is a *Requirements Specification*. This specification is composed of a set of NL requirements and a requirements model (hereafter referred to as a model). In our context, a model is composed of two types of model artifacts: 1) UML diagrams (specifically, CDs, ADs, and UCDs) and 2) a traceability matrix.

NL Requirements. Natural languages (NL), such as English, are commonly used for expressing systems and software requirements [51]. Table 5.1 shows an example of five NL requirements from the financial domain. The requirements in our example are uniquely identified with Ids composed of the letter `R` followed by a digit. Each requirement follows the requirement syntax defined by the Rimay language introduced in Chapter 3. In requirement `R1` shown in Ta-

ble 5.1, there is a triggering condition (`When the Order_Issuer...creates an Order of type Subscription_Order`), an actor (`Order_Issuer`) that responds to the trigger, and one system response (`set the settlement_method of the Order to "FOP"`). Free of payment (`FOP`) is a securities industry settlement method that is not linked to a corresponding transfer of funds. In this case, only the securities are moved. An example involving a `FOP` transaction may be gifts or donations. In requirements `R2`, `R3` and `R4` (Table 5.1) , we use the generic names `System`, `Transfer_System`, and `Data_Provider` to anonymize the names of the settlement platform, the secure file transfer connectivity system, and the funds data provider of our industrial partner. Moreover, we use the name `File` to refer to a set of specific information about fund documents. For example, each line of the `File` may include the document type identifier, the codes of the countries in which the document can be published, the `ISIN` (International Securities Identification Number) code defining the share class for the document, a flag indicating if the document is written for a specific group of investors only, or the document URL. Requirement `R5` (Table 5.1) includes the terms `ISIN` and `Share_Class_Identifier`. `ISIN` is a code that uniquely identifies a specific securities issue. `Share_Class_Identifier` is a designation applied to a type of security, such as a mutual fund unit in the settlement platform of our industrial partner.

Table 5.1: Example NL requirements

ID	Requirement Description
R1	When the Order_Issuer (hereafter known as OI)creates an Order of type Subscription_Order, then OI must set the settlement_method of the Order to "FOP"
R2	When Transfer_System receives a File, Transfer_System must forward the File to System
R3	Every "calendar day", Data_Provider must send a File
R4	Before "8:00 am", every "calendar day", if System does not receive the File, then System must create an "Alert"
R5	For each "line of the File", System must check that Share_Class_Identifier.Value contains "line.ISIN"

Model. We assume that the input requirement model has been created by following the AGAC methodology. AGAC uses three types of UML diagrams to represent requirements: UCDs, CDs and ADs. Figure 5.4 shows *excerpts* of diagrams that can be enriched with information extracted from the NL requirements shown in Table 5.1. *Actors* are defined in UCDs (e.g., the actor `Order_Issuer`) and execute *actions* that are part of the *activities* represented in ADs (e.g., `Create Order`). *Domain entities* and their properties are characterized by a *domain model*, represented using CDs and referenced within the ADs. For example, the type `Subscription_Order`, shown in the CD, has one property named `settlement_date`. In the AD `Create subscription order` at the top of Figure 5.4, the action `Create Order` creates the object of type `Subscription_Order`, which is specified in the domain model.

Models typically include trace relationships (traces) between model elements that are mainly used in UML for tracking requirements and changes across models [49]. Those traces are usually represented in a traceability matrix. In our context, traces between NL requirements and AD actions are sufficient for our purpose as the extraction rules are driven by the control flow captured by actions (as we explain later in Step 1 of Section 5.4.1). Table 5.2 shows an example traceability matrix where the columns represent actions from Table 5.3 (discussed in Section 5.4.1), and the rows represent NL requirements

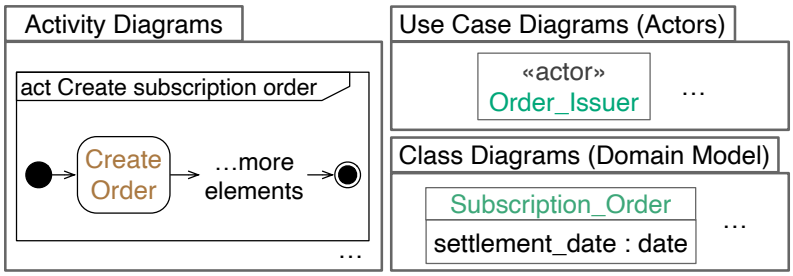


Figure 5.4: Model excerpts

from Table 5.1. An "X" in Table 5.2 means that the action in the column is *traced to* the NL requirement in the row, and vice versa. We also say that the action in the column is a *traced action* of the NL requirement. The Rimay tool is tightly integrated with the modeling environment and ensures that trace relationships are established correctly during requirements writing.

Table 5.2: Traceability matrix

		Model Elements				
		Create Order	Forward File	Send File	Create Alert	Check ISIN
Requirements	R1	X				
	R2		X			
	R3			X		
	R4				X	
	R5					X

The Approach. Our approach is composed of five steps, as depicted in Figure 5.3. All the steps, except "Enrich Model" (Step 4), are performed automatically.

Step 1 is concerned with extracting information (e.g., actors, classes, properties, conditions) from NL requirements expressed using the Rimay language (see Section 3.4). Only the NL requirements that are traced to model elements are analyzed during Step 1.

Step 2 is concerned with identifying the model elements that can be enriched by using the information extracted in Step 1. The inputs of this step are the model elements. The output of this step is the information about what and where the model elements can be enriched.

Step 3 is concerned with creating recommendations that suggest how to enrich the model elements in order to produce better AC. Each recommendation explains (a) what model element to enrich, e.g., activity partition, object, property value, and (b) where is the model element to enrich, i.e., the exact location of the model element traced to the NL requirement.

In Step 4, the user (in our case, an analyst) manually reviews the recommendations produced in Step 3 and decides whether to enrich the model according to the recommendations, or to discard the recommendations.

Step 5 is concerned with automatically generating AC from the enriched model. This step is performed via AGAC (see Section 5.2.2).

5.4 Information Extraction Approach for Deriving better AC

In this section, we describe in detail and illustrate over a small example the five steps of the approach of Figure 5.3.

5.4.1 Step 1. Extract Information

This step aims to answer RQ1. To do so, we propose 13 rules to extract information content from NL requirements that is relevant for generating AC. Our extraction rules were borne out of an analysis of the AC-related conceptual overlaps between the element types in models and the element types in Rimay, discussed in Section 5.2. To derive the rules, we systematically analyzed the Rimay grammar, identified correspondences between the grammar and the element types in the models, and finally defined the extraction rules.

Table 5.3 shows our extraction rules. The rules are organized into four categories that correspond to the main grammar rules of Rimay from which the information is extracted. These categories are: (a) `SCOPE`, (b) `CONDITION_STRUCTURE`, (c) `ACTOR`, and (d) `SYSTEM_RESPONSE`.

Each extraction rule in Table 5.3 has three columns "ID", "Extraction Rule / Recommendation" and "Example". The "ID" column uniquely identifies an extraction rule and is composed of the first letter(s) of the rule's category name and a number. For instance, `S1` and `SR3` are the first and third extraction rules related to the categories `SCOPE` and `SYSTEM_RESPONSE`, respectively.

Due to space constraints, we include the extraction rules and recommendations in the "Extraction Rule / Recommendation" column. In Step 3, we will discuss the recommendations. The structure of a rule is: *Rule: If* [conditions to be checked in a requirements specification], *then* [information to extract from the NL requirements]. To illustrate the rules in Table 5.3, consider rule *S1* as an example: "If a prepositional phrase starts by "for each" and mentions the type ***A*** of the collection that will be iterated over and the item ***B*** in the collection, then extract ***A*** and ***B***". Given the prepositional rule in `R5`: `For each "line of the File"`, `S1` will extract `"File"` (i.e., the type of the collection) and `"line"` (i.e., the item in the collection).

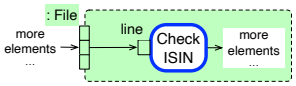
The "Example" column shows the model elements extracted from a single NL requirement identified by the letter `R` followed by a number (e.g., `R1`). We highlight in blue some of the model elements in the "Example" column; this is to show that the elements are traced to the NL requirement shown in the same table cell as the respective element. Moreover, we (1) shade an element green when there are updated or new elements in the model, and (2) enclose the text in the NL requirements in a red rectangle when the text mentions updated or new elements in the model.

Step 1 (Figure 5.3) is composed of two sub-steps: *Sub-Step 1.1* identifies the NL requirements that are traced to model elements (i.e., AD actions), based on a traceability matrix. If an NL requirement is not traced to any model element, our tool will show a warning message to the analysts. For instance, NL requirements `R1` to `R5` (Table 5.1) will be selected in Sub-Step 1.1 because they are traced to at least one model element according to the traceability matrix shown in Table 5.2. *Sub-Step 1.2* extracts information (e.g., conditions, actors, triggers, verb phrases, etc.) from the NL requirements selected in Sub-Step 1.1. The information to be extracted is determined by the rules shown in Table 5.3. Each rule extracts information from a specific part of an NL requirement. For instance, extraction rule `C1` (Table 5.3) extracts verb phrases from the `When` structure of NL requirements. In the case of `R2` (Table 5.1), `C1`

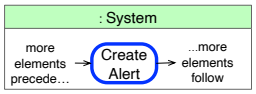
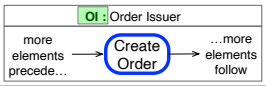
The table is organized according to the different grammar rules of Rimay: (a) SCOPE, (b) CONDITION_STRUCTURE, (c) ACTOR, and (d) SYSTEM_RESPONSE.

Table 5.3: Information extraction rules for NL requirements written in Rimay and the associated model-enrichment recommendations

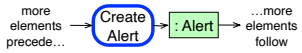
a)

ID	Extraction Rule / Recommendation	Example
S1	**Rule:** If a prepositional phrase starts by "for each", and further mentions: the type ***A*** of the collection that will be iterated over and an item ***B*** in the collection, then extract ***A*** and ***B***. **Recommendation:** (1) create an expansion region to include the traced action, (2) add an expansion node to the expansion region, (3) set the type of the expansion node to ***A***, (4) create an object node to represent ***B***, and (5) connect the expansion node to the object node.	R5: For each "line of the File", System must check that Share_Class_Identifier.Value contains "line.ISIN". 

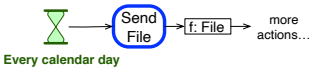
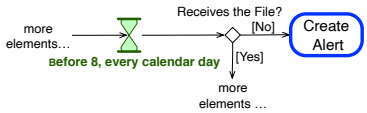
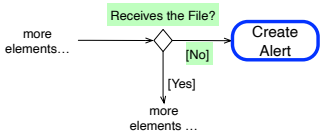
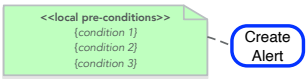
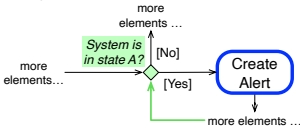
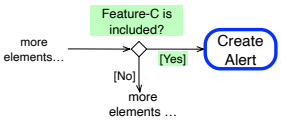
c)

ID	Extraction Rule / Recommendation	Example
A1	**Rule:** If an actor ***A*** in an NL requirement does not match the name of any UML actor linked to the activity partition of the traced action, then extract ***A***. **Recommendation:** (1) Create a UML actor and named it ***A***, (2) create an activity partition to include the traced action, and (3) link the activity partition to the created UML actor.	R4: Before "8:00 am", every "calendar day", if System does not receive the File, then System must create an "Alert". 
A2	**Rule:** If an actor has an alias ***A*** in an NL requirement, then extract ***A***. **Recommendation:** Name ***A*** the traced action's activity partition.	R1: When the Order_Issuer (hereafter known as OI) creates an Order of type Subscription_Order, then OI must set the settlement_method of the Order to "FOP". 

d)

ID	Extraction Rule / Recommendation	Example
SR 1	**Rule:** If a system response creates data ***A*** (e.g., Report, Instruction, Alarm), then extract ***A***. **Recommendation:** (1) Create an object node with the same name and type as ***A***, and (2) connect the traced action to the object node.	R4: Before "8:00 am", every "calendar day". if System does not receive the File, then System must create an "Alert". 
SR 2	**Rule:** If a system response sets a value to a property of a traced action's output object, then extract the property and the value. **Recommendation:** (1) Add the property to the object node, (2) set the property's value.	R1: When the Order_Issuer (hereafter known as OI) creates an Order of type Subscription_Order, then OI must set the settlement_method of the Order to "FOP". 
SR 3	**Rule:** If a system response refers to a specific data ***A*** by name, then extract ***A***. **Recommendation:** Name the object node with the same name as ***A***.	R1: When the Order_Issuer (hereafter known as OI) creates an Order of type Subscription_Order, then OI must set the settlement_method of the Order to "FOP". 

b)

ID	Extraction Rule / Recommendation	Example
C1	**Rule:** If the verb phrase ***A*** in a When structure does not match the name of any of the actions preceding the traced action, then extract ***A***. **Recommendation:** Create an action named ***A***.	R2: When Transfer_System receives a File, Transfer_System must forward the File to System. 
C2	**Rule:** If a prepositional phrase ***A*** expresses a timed event, and the timed event does not match any of the events or actions preceding the traced action, then extract ***A***. **Recommendation:** Create a time-triggered event named ***A***.	R3: Every "calendar day", Data_Provider must send a File. 
C3	**Rule:** If two prepositional phrases ***A*** and ***B*** express a timed event and do not match any of the events or actions preceding the traced action, then extract ***A*** and ***B***. **Recommendation:** Create a time-triggered event that combines the information of ***A*** and ***B***.	R4: Before "8:00 am", every "calendar day", if System does not receive the File, then System must create an "Alert". 
C4	**Rule:** If a condition ***A*** in an If structure does not match any decision node preceding the traced action, then extract ***A***. **Recommendation:** (1) create a decision node, (2) change negative conditions to positive ones (e.g., change from "does not receive" to "receives"), and (3) name the decision node ***A***.	R4: Before "8:00 am", every "calendar day", if System does not receive the File, then System must create an "Alert". 
C5	**Rule:** If the condition ***A*** in an If structure does not match any decision node preceding the traced action, then extract ***A***. **Recommendation:** Create a local pre-condition (constraint) in the traced action for each sub-condition in ***A***.	R4 (modified): If "condition 1", If "condition 2", and If "condition 3", then System must create an "Alert". 
C6	**Rule:** If a condition ***A*** in a While structure does not match any decision node in a pre-test loop preceding the traced action, then extract ***A***. **Recommendation:** Name ***A*** the decision node in the pre-test loop.	R4 (modified): While "System is in state A", System must create an "Alert". 
C7	**Rule:** If a condition ***A*** in a Where structure does not match any decision node preceding the traced action, then extract ***A***. **Recommendation:** Create a decision node, and name it ***A***	R4 (modified): Where "Feature-C is included", System must create an "Alert". 

extracts the verb phrase "receives a File". To give a more complete example of Sub-Step 1.2, Figure 5.5 shows requirement R1 alongside the AC-related information that can be extracted using the extraction rules of Table 5.3. According to extraction rules C1, A1, A2, and SR1 to SR3, the type of information extracted from R1 is actor, actor alias, action, object type, property name, object name, and property value. The yellow marks in Figure 5.5 indicate the information content in R1 extracted by the extraction rules. For instance, "property value" (e.g., "FOP") is a type of information extracted by the rule SR2.

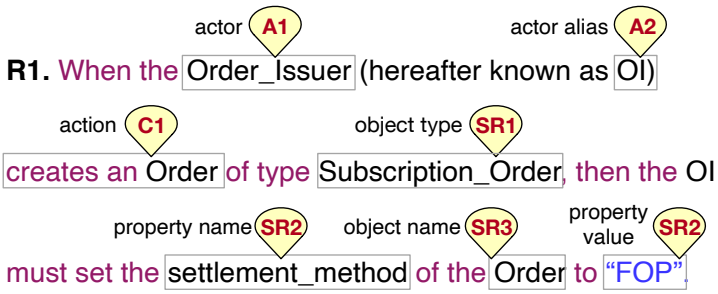


Figure 5.5: Illustration of information extraction applied to requirement R1 from Table 5.1

The extracted information is shown in rectangular boxes and the corresponding extraction rules in pin-shaped pointers

5.4.2 Step 2. Identify Model Elements to Enrich

This step uses the information extracted in Step 1 (Figure 5.3) in order to identify the model elements (e.g., activity partition, action, object, property, etc) that can be enriched. A model element is enriched when we add to it new AC-related information extracted from NL requirements using the extraction rules in Step 1.

In order to identify what model elements need to be enriched, our approach automatically compares the text sequences extracted from NL requirements (i.e., the information extracted in Step 1) to the names of the elements in the model. Specifically, our approach (1) runs a pre-processing step that includes lowercase text conversion, stemming (a process of reducing inflected words to their word stem), and stop-word removal (words such as articles are removed), and (2) deems two text sequences as matching when they are syntactically identical. To illustrate how the matching works, consider rule C1 and its example shown in Table 5.3 (b). According to C1, our approach needs to determine if there is an action named "receives a File" that precedes the action "Forward File". If such an action does not exist, our approach classifies the model element as *enrichable*.

For example, given the information extracted from Step 1 (actor: Order_Issuer, actor alias: OI, object name: Order, object type: Subscription_Order, property name: settlement_method, and property value: FOP), our approach identifies six model elements in Figure 5.4 that are enrichable for the following reasons:

1. The AD does not have an activity partition linked to the actor "Order_Issuer" defined in the UCD (rule A1).

2. There is no activity partition named "OI" in the AD (rule A2).

3. The action "Create Order" in the AD does not have any output object node of type "Subscription_Order" (rule SR1).

4. The output object node of type "Subscription_Order" does not have the property "settlement_method" (rule SR2).

5. The property "settlement_method" is not set to "FOP" in the output object node of type "Subscription_Order" (rule SR2).

6. There is no object node named "Order" in the AD (rule SR3).

5.4.3 Step 3. Create Recommendations

This step creates recommendations for the analysts regarding how to enrich the model elements classified as enrichable in Step 2 (Figure 5.3). All the recommendations were shown already in Table 5.3. Each recommendation describes the tasks that analysts should perform to enrich a model using AC-related information extracted by a rule shown in the same cell of Table 5.3.

Step 3 does not recommend tasks that cause the duplication of model elements. For example, a recommendation for the extraction rule `S1` (Table 5.3) will not include the task "create an expansion region to include the traced action" if the model already has an expansion region that includes the traced action. Recall from Section 5.3 that a traced action is one that is related to an NL requirement through a trace relationship.

The recommendations use mostly terminology from UML e.g., local-precondition, event, condition, expansion region. In addition, other recommendations use terminology derived from programming languages. For instance, the recommendation of the extraction rule `C6` (Table 5.3) uses the term *pre-test loop*. A pre-test loop is one in which a set of actions is to be repeated until a specified condition is no longer true, and the condition is tested before the set of actions is executed. In modern programming languages, a pre-test loop is implemented using the *while* statement.

Step 3 uses the general recommendations shown in Table 5.3 to produce specific recommendations for the model being created by the analysts. To illustrate the type of specific recommendations produced by Step 3, Table 5.4 shows five recommendations to enrich the model shown in Figure 5.4. The recommendations described in the "Description" column (Table 5.4) contain predefined text and text extracted from the NL requirements. The predefined text is in normal black and the text obtained from the NL requirements is in bold. For instance, in recommendation `Rec.1` (Table 5.4) the `Order_Issuer` is a model element extracted from the NL requirement that is used to enrich the model.

Table 5.4: Recommendations to enrich the model of Figure 5.4

ID	Description	Rule
`Rec.1`	Create an actor and name it "**Order_Issuer**", create an activity partition to include the "**Create Order**" action, and link the activity partition to the "**Order_Issuer**" actor	A1
`Rec.2`	Name "**OI**" the activity partition of the "**Create Order**" action	A2
`Rec.3`	Create an object node of type "**Subscription_Order**", and connect the "**Create Order**" action to the object node	SR1
`Rec.4`	Add the property "**settlement_method**" to the object node of type "**Subscription_Order**"	SR2
`Rec.5`	Set the "**settlement_method**" property's value to "**FOP**"	SR2
`Rec.6`	Name the object node as "**Order**"	SR3

5.4.4 Step 4. Enrich Model

In this step, the analysts consider the recommendations generated by our approach. The recommendations contain the steps that the analysts have to perform to enrich the model. This step is carried out manually because the analysts should have the final say as to whether to follow or discard the recommendations. Figure 5.6 exemplifies the output of Step 4 for the model shown in Figure 5.4 and requirement `R1` in

Figure 5.5, according to the six recommendations shown in Table 5.4. Note that, in Figure 5.6, we have assumed that the analysts would accept all the recommendations. We use comments (identified with the ids of the extraction rules that produced each recommendation) to mark the places in the model that have been enriched by the analysts.

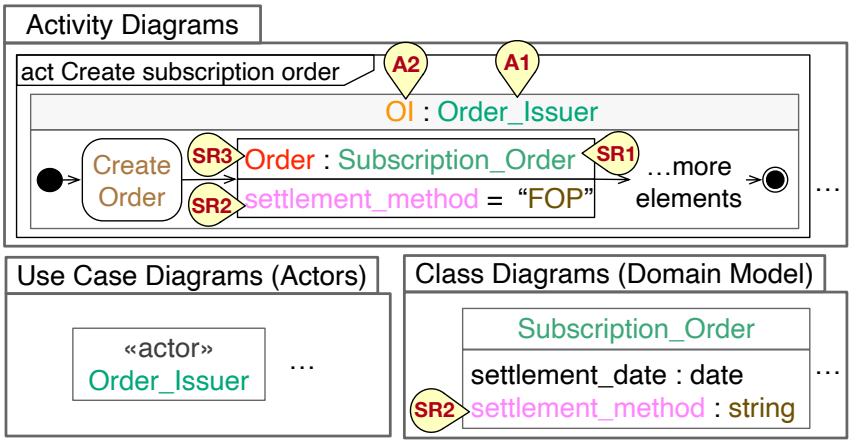


Figure 5.6: Enriched model and the mapping of new elements to the extraction rules of Table 5.3

5.4.5 Step 5. Generate Acceptance Criteria

This step automatically generates AC based on the intents of the AD actions in the model enriched by the analysts (Step 4). To generate AC, we use a set of predefined AC templates provided by AGAC (see Section 5.2.2). For example, the acceptance criterion shown in Figure 5.7 is generated from the action named "Create Subscription Order".

This acceptance criterion exercises the following behavior: when the *Order_Issuer* (*OI*) executes *Create a Subscription_Order*, if the *Order* does not exist, then the *Order* is created and its *settlement_method* is set to *FOP*.

```
@Intent Create
@Requirement_Id: R1
Scenario: Create an Order
Given an Order of type Subscription_Order does not
        exist in OI of type Order_Issuer
When OI Create Order,
Then Order exists in OI
  And the property settlement_method of Order is equal to FOP
```

Figure 5.7: Example acceptance criterion related to the model of Figure 5.6

5.5 Empirical Evaluation

In this section, we describe the case study we carried out to address RQ3. Throughout the section, we follow best practices for reporting on case study research in software engineering [56].

5.5.1 Objectives and Design

We evaluated our approach using a real requirements specification developed by our industrial partner. The specification includes three ADs, three CDs, one UCD, and 23 NL requirements. We reflect on the representativeness of this case study in Section 5.6.

We applied the 13 extraction rules shown in Table 5.3 to the 23 NL requirements and obtained 27 recommendations for the analysts to enrich the model. We carried out this evaluation in close collaboration with five analysts at our industrial partner. All the analysts were domain experts, with significant experience writing financial system requirements ranging from 7 to 28 years. We asked these analysts to answer the following two questions:

(Q1) Are the recommendations to enrich the model useful to generate better AC? *Yes/No*

(Q2) Should the refinements introduced into the model be made visible to the analysts to facilitate its interpretation? *Yes/No*

Q1 addresses relevance as the main topic of investigation in our evaluation. Relevance refers to whether analysts deem a recommendation useful to enrich a model and generate more precise and complete AC. For the purpose of this evaluation, we are interested in Q1 only. Nevertheless, we also included Q2 to gather feedback on the level of detail that analysts deem useful to show in models in order to facilitate their understanding. Naturally, for a given recommendation, Q2 was asked only when the answer to Q1 was positive (Yes).

5.5.2 Preparation for Data Collection

In this section, we define the procedures and protocols for data collection. The following data was provided to the analysts for evaluation: (a) the original requirements specification (including NL requirements and a model), (b) the set of 27 model-enrichment recommendations produced by our approach, and (c) the AC generated from the model in (a). For each recommendation, as a group and through discussions, the analysts were asked to: (1) if needed, enrich the model following the recommendation, (2) generate AC from the enriched model, (3) compare the AC in (2) to the AC generated from the original model in (a), and (4) answer Q1 and Q2. Note that the five analysts were already trained to use AGAC and to read AC specified using Gherkin scenarios.

5.5.3 Collecting Evidence and Results

We report on the analysis of the data collected from the questionnaire's answers. Table 5.5 shows the number of Yes and No answers to Q1 and Q2, given by the group of analysts after they reached consensus.

Table 5.5: Questionnaire answers

Question	Yes	No
Q1	24	3
Q2	7	17

With regard to Q1, the analysts agreed to follow 24 of the 27 recommendations to enrich the model. Table 5.6 compares the original model and the enriched one in terms of numbers of elements of different

types. The increase in instances across all element types in the *enriched* model confirms that our approach is effective in creating recommendations that the analysts deem necessary for the purpose of AC generation. After enriching the model, the analysts generated AC. Table 5.7 provides a comparison of the AC that were generated from the original and the enriched model. In AGAC, one AC is described using a sequence of one or more Gherkin scenarios. Therefore, we compared AC in terms of the total number of Gherkin scenarios (`Given-When-Then` structures), and the pre- and post-conditions in their `Given` and `Then` parts, respectively. Though there is a general, significant increase, the most striking value in Table 5.7 is the 596% increase in the number of post-conditions. This result was mainly due to the addition of 75 values for the properties of action's output objects (Table 5.7) that AGAC transforms into post-conditions in the Gherkin scenarios. As a result, along with a more modest increase in pre-conditions, this contributes to making AC more precise. Another noteworthy result is the 22% increase in Gherkin scenarios, thus showing that model refinement leads to more complete AC.

Table 5.6: Comparison of the original and enriched models

Model Element	Original	Enriched	% Increase
Actions	22	24	9.1%
Events	1	3	200%
Objects	11	15	36.4%
Decision Nodes	8	9	12.5%
Fork and Join Nodes	2	3	50%
Propriety Values	0	75	N/A

Table 5.7: Comparison between the original AC (Original) and the AC derived from the enriched model (Augmented)

AC Details	Original	Augmented	% Increase
Pre-conditions	438	535	22,1%
Post-conditions	325	2262	596%
Gherkin scenarios	156	191	22,4%

5.5.4 Analysis of Collected Data

In this section, we assess the relevance of our approach in producing better AC by answering Q1 and Q2.

To answer Q1, we define the following metrics, based on the answers provided by the five analysts: (1) true positives (TPs) as the cases in which the recommendation is deemed correct and relevant to the generation of AC, (2) false positives (FPs) as the cases in which the recommendation is deemed irrelevant and has no bearing on the generation of AC, and (3) false negatives (FNs) as the cases in which the analysts identify recommendations that are missed by our approach, but are useful for the generation of AC.

We calculate precision as $TP/(TP + FP)$ and recall as $TP/(TP + FN)$. Table 5.8 shows our accuracy results: over our case study, the recommendations have a precision of 89% and recall of 100%. There are three FPs, resulting from three recommendations that suggested missing elements in the model, but these elements were identified to be already present by the analysts. Our approach did not yield any FNs due to the systematic derivation of our extraction rules (Section 5.4.1). Nevertheless, the analysts

Table 5.8: Accuracy metrics for our recommendations

TPs	FPs	FNs	P%	R%
24	3	0	89	100

were asked to try to identify any recommendations that our approach might have missed; they could not identify any.

To answer Q2, we considered only the 24 recommendations that were deemed correct and relevant by analysts (Q2 in Table 5.5). The analysts found the additional information resulting from 17 of these recommendations not worth visualizing in the model. For example, the analysts agreed that details such as the expected run-time property values extracted by extraction rule `SR2` (Table 5.3) did not need to be visualized. The analysts agreed that the additional information resulting from the seven remaining recommendations should be visualized as the information was deemed helpful for improving model comprehension. For example, the analysts found the conditions extracted from NL requirements by rule `C4` (Table 5.3) to be useful information to visualize in decision nodes. To conclude, most of the information extracted from NL requirements, despite being relevant to the generation of AC, was deemed not useful for visualization purposes. This observation provides evidence about the largely complementary nature of the information content captured by models versus NL requirements.

5.6 Threats to Validity

Internal validity focuses on confounding factors. The main confounding factor to mitigate in our case study is related to the fact that analysts may have influenced one another when responding to our questionnaire during the evaluation. In particular, the answers obtained from the analysts could be about what each person feels and thinks, but it could also be influenced by a phenomenon such as 'groupthink', through which people conform to what others believe. To mitigate this bias, for each recommendation, we asked analysts to (1) first answer Q1-Q2 offline, and (2) then report their answers during the group discussion. In addition, we focused the group discussion on using consensus-seeking dialog as a method to converge (on answers to Q1-Q2) through debate.

External validity concerns the generalizability of our case study results. Although our evaluation is based on a single case study, the NL requirements and the model in the case study are representative of a broader class of information systems, such as the ones in the banking and securities industry. However, future investigations are necessary to determine whether and how our approach can be applied to other domains and information systems.

In our case study, the `While` and `Where` structures – two out of Rimay's eight main grammar rules (see Section 3.4) – were left unused by the NL requirements. Nevertheless, these structures would be treated similarly to the `If` structure (which does get used in our case study). As for the AC, the Gherkin scenarios derived from the enriched model in our case study cover six out of the 11 intent types supported by AGAC. The covered intent types are: "delete", "send", "receive", "update", and "validate". The intent types not covered are "read", "enable", "disable", "display" and "not display". The template for "read" is conceptually similar to that for "send". For example, the post-condition for "read" ("Then [actor] read [object]") is similar to that for "send" ("Then [actor] sent [object]"). The templates for the other four intent types ("enable", "disable", "display" and "not display") are similar to that for "delete". For example, the pre-condition for all these four intent types is "Given [object] exists in [actor]", just like for

"delete". We thus anticipate that our results will generalize to other systems of the same type. This being said, further case studies involving other types of systems that are commonly modeled using ADs, CDs and UCDs, e.g., public administration services, remain necessary for improving external validity.

5.7 Related Work

This section presents existing work that is related, to various degrees, to the extraction of information from NL requirements with the objective of creating or enhancing UML models.

We consider eight studies relevant to our work; seven aim at extracting information from NL requirements to create UML models and one – to enhance existing UML models for the purpose of test-case generation. Table 5.9 outlines these eight studies. The first column cites the study. The second column indicates whether an empirical evaluation was conducted as part of the study. The third column shows the number of rules, patterns, or heuristics that are included in the study. Finally, the fourth column indicates whether the study uses CNLs, patterns, or templates.

Table 5.9: Summary of related work

Study Reference	Empirical Evaluation	Number of Rules	Restricted NL?
Ilieva et al. [29]	No	14	No
Moreno et al. [47]	No	9	No
Fliedl et al. [23]	No	5	No
Smialek et al. [62]	No	6	Yes
Arora et al. [6]	Yes	21	No
Thakur et al. [68]	Yes	54	No
Yue et al. [80]	Yes	65	Yes
Wang et al. [73]	Yes	N/A	Yes

The approaches described in [29, 62, 23, 47] were identified from the systematic review of transformation approaches between user requirements and analysis models conducted by Yue et al. [78]. We further included subsequent studies to this systematic review [6, 68, 80, 73].

We start with the studies that extract information from NL requirements for the creation of UML models. Ilieva et al. [29] propose a methodology to extract information from unrestricted NL requirements to build UCDs, ADs, and domain models. To this end, they describe 14 heuristics that define the correspondence between UML model elements and pieces of information in the NL requirements. Moreno et al. [47] discuss nine patterns for structuring the information representing the data with which the system works (static information) and the data that describes the system behavior (dynamic information) as extracted from unrestricted NL requirements. In this study, static and dynamic information is used, respectively, for building conceptual and behavioral models.

Fliedl et al. [23] propose a semi-automated approach that considers the user's feedback to linguistically analyze unrestricted NL requirements and translate these requirements into a conceptual pre-design schema. This schema contains static and dynamic information. The study maps this information to UML model diagrams, e.g, ADs, using a set of five extraction rules.

Smialek et al. [62] restrict the representation of UCDs through a concrete syntax. They describe the meta-model of their concrete syntax and define mapping patterns that link the concrete syntax meta-model

components to the UML elements of Sequence Diagrams and ADs. They describe six mapping patterns in the study. None of the above approaches report empirical evaluations.

Arora et al. [6] collect from previous work a set of 18 extraction rules and propose three new extraction rules for unrestricted NL requirements. Their rules extract elements for building domain models. The authors evaluate the usefulness of their approach via a case study and expert surveys.

Thakur et al. [68] report an automated approach to extract domain elements from unrestricted UCD specifications. They propose 54 extraction rules and empirically compare their approach with two similar approaches.

Yue et al. [80] propose an approach that, by implementing 65 transformation rules (28 rules for CDs, 18 rules for sequence diagrams, and 19 rules for ADs), automatically generates UML model diagrams from NL requirements. They consider NL requirements that conform to a restricted natural language, named RUCM [79], for writing UCD specifications. They empirically evaluate their approach in terms of consistency, completeness, applicability, and performance through a series of case studies.

All the above-mentioned studies target the construction of UML models from scratch. In contrast, the solution proposed by Wang et al. [73] updates UML models using NL requirements to generate test cases. Their approach enables users, based on UCD specifications written in RUCM [79], to extract model elements for checking completeness and add new information to models, e.g., conditional statements to be inserted as pre- and post-conditions. Additionally, their approach creates a test model for the generation of test cases. An empirical evaluation reported that their approach generates 25% more test case scenarios compared to manually written test case scenarios developed by experts.

To summarize, seven studies [29, 47, 23, 6, 68, 80, 62] discussed in this section suggest rules to fully transform textual requirements to models. One of our rules ($C4$), presented in Table 5.3, shares the same rationale as a rule introduced by Ilieva and Ormandjieva [29]. The rest of our rules differ from the rules, heuristics, and patterns introduced in the above-cited studies. One approach [73] includes information from NL requirements to enhance conceptual models and generate test cases. In contrast, our approach identifies AC-related information from NL requirements which, if included in the model, can significantly improve the precision and completeness of generated AC in terms of Gherkin scenarios. Moreover, our approach differs from that of Wang et al. in the nature of the NL requirements. Whereas Wang et al. orient their work around textual use-case specifications resulting from object-oriented analysis, the CNL that underlies our work, Rimay, has been designed around more traditional requirements engineering practices where the requirements are expressed as independent statements with modal verbs (e.g., shall, must, will).

5.8 Conclusions

The goal of this chapter is to better support the model-based derivation of system test Acceptance Criteria (AC) by enriching requirements models with additional information from natural language (NL) requirements. Though this chapter targets a specific modeling methodology (AGAC) and controlled natural language (Rimay) for requirements specifications, many of the principles we describe are general. Through a comparative analysis of the conceptual overlap of AGAC and Rimay, we first systematically derived 13 rules to extract AC-relevant information from NL requirements. Then, we proposed a semi-automatic approach that identifies the model elements that can be enriched with this extracted information, creates recommendations for the analysts, augments the model according to the selected recommendations, and automatically generates AC from the enriched model using AGAC.

Our empirical evaluation, which was conducted through a case study in the financial domain, involved collecting feedback and decisions from five domain experts and provided initial but strong evidence of the feasibility and benefits of our approach. Indeed, most recommendations were followed by the experts (24 out 27) and led to a significantly augmented model and AC, thus providing stronger support for acceptance testing.

Chapter 6

Conclusions & Future Work

6.1 Summary

In this dissertation, we proposed solutions to improve the quality of NL requirements and requirement models. These solutions respond to the practical needs observed in the financial sector. For companies in this sector, it is desirable to reconcile the information content in NL requirements and models to easily communicate the requirements among project stakeholders and develop successful software applications. It is also important to note that leveraging requirement information content can enable the automation of certain tasks (e.g., generating AC). The work presented in this dissertation was conducted in collaboration with Clearstream Services SA Luxembourg, a security services company owned by Deutsche Borse AG. All contributions have been empirically evaluated using industrial case studies. Below, we summarize the contributions of this dissertation..

This dissertation introduced a new methodology for defining controlled natural languages (CNLs) to specify requirements. We used this methodology to create a CNL (Rimay) for specifying functional requirements in the financial domain. The grammar of Rimay was derived from an extensive quality study using 2755 requirements. The purpose of this quality study was to identify the information content that financial analysts should account for in the requirements of financial applications. We empirically evaluated Rimay in a realistic setting. In this evaluation, we measured Rimay's expressiveness to represent the requirements. The results showed that 405 of the 460 (88%) requirements evaluated in our case study could be expressed using Rimay.

We identified a set of 10 smells commonly observed in requirements from the financial domain. These smells describe the semantic and syntactic errors found in the requirements. Furthermore, we derived 10 Rimay patterns from the Rimay language, which were then provided as suggestions for users to fix smells. This resulted in the development of a tool that detects smells in requirements and suggests Rimay patterns to fix the smells. We evaluated this tool through an industrial case study, during which we measured the performance of our approach. Our results were then compared against a human-annotated ground truth. The results showed a precision of 88% and a recall of 74% in smell detection. Furthermore, Rimay patterns were suggested with a precision of 89% and a recall of 82%.

This dissertation also introduces a methodology that supports the model-based derivation of acceptance criteria (AC) by enriching requirements models with AC-related information in NL requirements. Our approach, along with Rimay, leverages a specific modeling methodology, the Automated Generation of Acceptance Criteria (AGAC), to specify requirements. First, we systematically derived 13 rules to extract AC-relevant information from NL requirements. These rules were identified by analyzing the conceptual overlap between AGAC and Rimay. Then, we proposed a semi-automatic approach that (1) identifies the model elements that can be enriched with this extracted information, (2) creates recommendations for analysts to augment the model according to the selected recommendations, and (3) automatically generates AC from the enriched model using AGAC. We evaluated our approach by applying our model enrichment method to an industrial case study. Our method suggested 27 recommendations for model enrichment; these results were validated by a group of experts who provided feedback and decisions about which recommendations were relevant to AC. The results showed that 24 of the 27 recommendations proved relevant (precision of 89%). This led to a significantly augmented model and AC.

6.2 Future Work

The findings of this dissertation lay the groundwork for future research to further improve current practices related to requirements specifications and acceptance testing. Further studies evaluating the usefulness of our solutions presented in Chapters 3 and 4 could broaden the knowledge of the benefits we can expect to improve the quality of NL requirements. Furthermore, the proposed smells outlined in Chapter 4 describe common syntactic and semantic problems encountered in NL financial requirements. There remains a broader spectrum of syntactic and semantic problems to consider. Therefore, we intend to augment our catalog of smells to provide better coverage for smell detection (Chapter 4).

Our method for generating recommendations for model enrichment presented in Chapter 5 uses syntactic comparisons between the information content of NL requirements and requirement models. Enhancing our approach to include semantic analysis would provide a more accurate match and would significantly improve the performance of our approach.

Appendix A

Action Phrases in Rimay

Table A.1 and Table A.2 show the name, summary, and examples of the Rimay grammar rules related to action phrases. Table A.1 displays the rules built during the qualitative study and Table A.2 depicts the rules created in the empirical evaluation.

Table A.1: Types of action phrase rules in Rimay (from Qualitative Study).

Grammar Rule Name	Grammar Rule Summary	Examples
ADMIT_65	exclude|excludes MODIFIER? PROPERTY| INSTANCE|TEXT (in ELEMENTS)? (using|based on TEXT)? (in compliance with TEXT (described in TEXT)?)?	exclude the "Gregorian dates that are not business days" in the System based on "the relevant calendar".
ADVISE_37_9_1	instruct|instructs (MODIFIER? ACTOR (to|in))? MODIFIER? TEXT (using|based on TEXT)?	instruct CAIN in "Deliveries" using the "P format only".
ALLOW-64.1	allow|allows| authorize|authorizes (MODIFIER? ACTOR to)? MODIFIER? TEXT (in MODIFIER? (CLASS|PLACE| UI_COMPONENT))?	Example 1: allow the "use of the new input media SIGMA" Example 2: allow the "use of wild card *" in the "criteria" field
BEG_58_2	request|requests MODIFIER? ACTOR (for|to TEXT)? (by using MODIFIER? TEXT)?	Example 1: request the System to "provide the following position types: AWAS, BLOK, BLCA, RSTR, DRAW, PLED" Example 2: request the System to "cancel the settlement" by using the "Order Reference".

Table A.1: (continued) Types of action phrase rules in Rimay (from Qualitative Study).

BEGIN-55.1-1	start|starts|begin|begins MODIFIER? TEXT	start the "calculation of the next NAV date on daily basis".
CANCEL	cancel|cancels MODIFIER? TEXT	cancel the "request of Validation".
CONCEALMENT-16-1	hide|hides MODIFIER? (UI_COMPONENT) |TEXT) (from ACTOR)?	hide the "PSC parties" section displayed on "Parties" screen.
CONTRIBUTE-13.2	restore|restores MODIFIER? PROPERTIES (to TEXT)? (for a period of TEXT starting from TEXT)?	restore "FundsHandler archived data" for a period of "10 years" starting from "Nov-2017".
CREATE-26.4	compute|computes| publish|publishes MODIFIER? PROPERTY (as TEXT)? (for MODIFIER? TEXT)? (using|based on TEXT)? ((in compliance with) ARTICLE? TEXT)? (described in ARTICLE? TEXT)?)?	Example 1: calculate the "Record Date Balance" using "Calculation Rule". Example 2: compute the "Trade Dated balance (TDB)" in compliance with "Trade Dated balance". Example 3: publish the "end of life statuses" for each "instruction types linked to an investment fund instrument" on SIGMA.
ENABLE_DISABLE	enable|disable MODIFIER? (ACTOR to)? TEXT (in MODIFIER? (CLASS|PLACE| UI_COMPONENT))?	enable the User to "select a 5, 6, 8 or 9-digit account number" in the "Client Account number" field.
ENFORCE_63	enforce|enforces MODIFIER? (ACTOR to)? TEXT (in MODIFIER? (CLASS|PLACE| UI_COMPONENT_INSTANCE))?	enforce the "upper case for the criteria entry" in the "Search" screen.
ENGENDER-27	create|creates| generate|generates ADVERB_PHRASE? MODIFIER? INSTANCES (in ELEMENTS)? (for MODIFIER? TEXT)? (in compliance with ARTICLE? TEXT)? (described in ARTICLE? TEXT)?)?	Example 1: create an "entry" in the "Market Calendar table". Example 2: create "5 different values of <num> <day>" in compliance with "converting rule". Example 3: create a "Transaction" in "Settlement Request" for "OI".

Table A.1: (continued) Types of action phrase rules in Rimay (from Qualitative Study).

EXCHANGE-13.6	replace|replaces MODIFIER? PROPERTIES for MODIFIER? TEXT (in compliance with| by applying the rule TEXT (described in ARTICLE? TEXT)?)?	replace the "4 last characters" of Allegements_Clearstream_ Identifier for "D001".
FORBID-67	prevent|prevents ARTICLE? (ACTOR? from TEXT) |TEXT	prevent the User from "deleting an element"
GET_FROM	download|downloads MODIFIER? INSTANCE|CLASS (through ACTOR) (and|or ACTOR)* (in compliance with TEXT (described in TEXT)?)?	download "the confirmation messages from "FDEP".
HERD-47.5.2	aggregate|aggregates MODIFIER? PROPERTY together?	aggregate all "fee Types"
INTERRUPT	interrupt|interrupts MODIFIER? (ACTOR|(TEXT process))(with TEXT)?	interrupt the "Settlement Request".
INVOLVE-107	include|includes MODIFIER? PROPERTY (in INSTANCE|CLASS)?	include the "5-digit creation account" in Settlement_Instruction.
KEEP-15.2	store|stores MODIFIER? (property|properties PROPERTIES value|values TEXT) (in MODIFIER? CLASS| INSTANCE)? (for a period of TEXT starting from TEXT)?	Example 1: store all "deleted parameters" in the "List A". Example 2: store the values "Validate", "Authorize",... in the Life_Cycle. Example 3: store the "FundsHandler data" in "location A" for a period of "at least 10 years" starting from "DD/MM/YYYY".
LIMIT-76	limit|limits|restrict| restricts|reduce|reduces MODIFIER? PROPERTY| INSTANCE (to TEXT)?	restrict the "DATA-ENTRY Profile" and "AUTHORIZATION Profile" to "have messages Setr.004, Setr.005".
MIGRATE	migrate|migrates MODIFIER? (NON_UI_COMPONENT_INSTANCE |CLASS)+ (from ACTOR|CLASS |INSTANCE|PROPERTY)? (to (ACTOR|INSTANCE| PROPERTY)+)?	migrate "All the data that have been decommissioned as listed in the KD01" to "Oxygen".

Table A.1: (continued) Types of action phrase rules in Rimay (from Qualitative Study).

Grammar Rule Name	Grammar Rule Summary	Examples
MIX-22.1-2	add|adds MODIFIER? PROPERTY|INSTANCE|ACTOR (about TEXT)? to MODIFIER? PROPERTY|INSTANCE|ACTOR	Example 1: add "SIGMA" to XY_System. Example 2: add a "record" about "missing Market Calendar" to the "exception log".
MIX_22_1_2_1	link|links (INSTANCE_WITH_INSTANCE| PROPERTY_WITH_PROPERTY) (, INSTANCE_WITH_INSTANCE| PROPERTY_WITH_PROPERTY)*	Example 1: link "BIC" to "Matching BIC". Example 2: link "allegement message MT578" to "outgoing CIF message - RTS".
NEGLECT-75-1-1	neglect|neglects| ignore|ignores MODIFIER? PROPERTY (from ELEMENTS)? (using|based on TEXT)? (in compliance with TEXT (described in TEXT)?)?	ignore the "ex/cum transaction condition indicator" from "Instruction".
OBTAIN-13.5.2	accept|accepts|receive| receives|retrieve| retrieves MODIFIER? INSTANCE|CLASS (from ELEMENTS_NO_UI)? (through ACTORS)? (in compliance with TEXT (described in TEXT)?)?	Example 1: receive a DA_file from CFCL_IT. Example 2: reject the "Message" in compliance with "current validation rules".
OTHER_COS-45.4	close|closes|reverse |reverses MODIFIER? UI_COMPONENT_INSTANCES	close the "Confirmation" message.
PUT-9.1	insert|inserts MODIFIER? PROPERTY|TEXT on|in MODIFIER? INSTANCE|PROPERTY|TEXT	append "XXX" on the "depository LI, LJ, LK, LL, LM, LO and YN".
REFLEXIVE_APPEARANCE-48.1.2	display|displays| show|shows MODIFIER? INSTANCES|CLASSES|TEXT (to ACTOR)? (as TEXT UI_COMPONENT?)? (on|in MODIFIER? (ACTOR| UI_COMPONENT_INSTANCE))? (until STRING)? (with the default (values |value) TEXT)?	Example 1: display "5, 6, 8 or 9-digit account number" in the "exported report account" field. Example 2: display the "relevant Jurisdiction Calendar" as "selected".

Table A.1: (continued) Types of action phrase rules in Rimay (from Qualitative Study).

Grammar Rule Name	Grammar Rule Summary	Examples
REMOVE_10_1	extract|extracts|remove| removes|delete|deletes| deduct|deducts MODIFIER? PROPERTIES (from ELEMENTS)? (using|based on TEXT)? (in compliance with TEXT (described in TEXT)?)?	Example 1: extract the "description" of Fund_frequency from the "reference data". Example 2: delete the "DECU field" from the "Settlement Parties block".
SAY-37.7-1	report|reports|propose| proposes MODIFIER? TEXT to ELEMENTS_NON_UI (using TEXT)?	report all "allegements received without a customer account" to Report_service using "defaulted Allegement Main Account".
SEE-30.1-1	detect|detects MODIFIER? NON_UI_COMPONENT_INSTANCE | CLASS (and|or (NON_UI_COMPONENT_INSTANCE |CLASS)? (on NON_UI_COMPONENT_INSTANCE |CLASS) (and|or NON_UI_COMPONENT_INSTANCE |CLASS)?)?	detect the "corresponding settlement request".
SELECT_ UNSELECT	select|selects|unselect| unselects MODIFIER? UI_COMPONENT_INSTANCE |CLASS|PROPERTY (from MODIFIER? NON_UI_COMPONENT_INSTANCE |CLASS|LABEL)? (using|based on TEXT)?	select the "last price date" from Vestima_ref_data.
SEND-11.1	return|returns|send|sends| forward|forward|pass| passes|export|exports ADVERB_PHRASE? (MODIFIER? CLASS| INSTANCE)+ (from ACTOR)? to MODIFIER? ACTOR (and|or MODIFIER? ACTOR)* (through ACTOR)?	send a "Settlement Request" to SIGMA.
SHAKE-22. 3-2-1	concatenate|concatenates MODIFIER? CLASS|INSTANCE with|into MODIFIER? LABEL? ((in compliance with| by applying the rule) TEXT (described in TEXT)?)?	concatenate "Accrued interest" with "Narrative". PROPERTY

Table A.1: (continued) Types of action phrase rules in Rimay (from Qualitative Study).

Grammar Rule Name	Grammar Rule Summary	Examples
SYNCHRONIZE	synchronize|synchronizes (INSTANCE_WITH_INSTANCE| PROPERTY_WITH_PROPERTY)+	synchronize "participants with the status Valid and the data corresponding to that status" with "participants and data of Vestima+".
THROW-17.1	discard|discards MODIFIER? TEXT (from TEXT)? (to TEXT)?	discard the "changes made by the user".
TRANSCRIBE-25.4	copy|copies MODIFIER? PROPERTY|LABEL into MODIFIER? INSTANCE| PROPERTY|TEXT	copy the "PSC" of FNCBL_Custodian_ SIP_participant into "Custodian SIP PSC" screen.
TURN-26.6.1	convert|converts|change| changes|transform| transforms PROPERTY_INSTANCE_ OR_VALUE_AND_ITS_CHANGE + (in compliance with TEXT (described in TEXT)?)?	Example 1: convert "<day> value" of Fund_frequency into "5 different values of <num> <day>" in compliance with "converting rule" described in "Rule_location". Example 2: The HUB must transform "cancellation" of Message to "TNP XML".
UPDATE	update|updates|set|sets PROPERTY_INSTANCE_OR_ VALUE_AND_ITS_CHANGE + (in compliance with TEXT (described in TEXT)?)?	set the "83a: Instr. Party" field into "EDA (account 10999)".
USE-105	use|uses|apply|applies MODIFIER? NON_UI_ COMPONENT_INSTANCE|CLASS| LABEL|PROPERTY|TEXT (as NON_UI_COMPONENT_INSTANCE| CLASS|LABEL|PROPERTY| TEXT)? (for|to TEXT)? (in ACTOR|CLASS|LABEL| PROPERTY)? (during TEXT)?	use the "wild card *" in the "criteria".
VALIDATE	validate|validates| check|checks (MODIFIER? NON_UI_COMPONENT_INSTANCE _OR_CLASS_TO_BE_VALIDATED by checking that)|that| TEXT (EXPRESSION)? (following MODIFIER? TEXT)?	validate Settlement_Request by checking that "Transaction Type" contains "SWIT".

Table A.2: Types of action phrase rules in Rimay (from Empirical Evaluation).

Grammar Rule Name	Grammar Rule Summary	Examples
CALCULATE	calculate|calculates| recalculate|recalculates MODIFIER? TEXT (for MODIFIER? TEXT)? (using|based on TEXT)? ((in compliance with) ARTICLE? TEXT)? (described in ARTICLE? TEXT)?)?	calculate the "Record Date Balance" using "Calculation Rule".
ESTABLISH_ 55_5_1	establish|establishes MODIFIER? TEXT (with TEXT)?	establish a "mechanism of Ack and Nack to ensure that the Settlement request has been received by Vestima register".
SEARCH_35_2	search|searches for MODIFIER? (PROPERTY|INSTANCE)+ on|in ACTOR	search for the Account_Trades in Vestima_Prime_GUI.
SPLIT	split|splits MODIFIER? PROPERTY (into TEXT)? using TEXT	split the "frequency description" into "individual values" using "+ sign".
STOP_55_4	stop|stops| finish|finishes (TEXT|(MODIFIER? TEXT|CLASS))	stop "processing the Settlement Request".
SUBSCRIBE	subscribe|subscribes (to|for) MODIFIER? TEXT	subscribe to the "PM publisher flow related to the instruction status update".
UPLOAD	upload|uploads ADVERB_PHRASE? MODIFIER? (NON_UI_COMPONENT_INSTANCE| INSTANCE)+ to MODIFIER? (ACTOR|CLASS|INSTANCE| PROPERTY)+ (through ACTORS)?	upload the "excel file" to the System.

Bibliography

[1] AHONEN, J. J., AND SAVOLAINEN, P. Software engineering projects may fail before they are started: Post-mortem analysis of five cancelled projects. *J. Syst. Softw. 83*, 11 (2010), 2175–2187.

[2] ALFÉREZ, M., PASTORE, F., SABETZADEH, M., BRIAND, L. C., AND RICCARDI, J. Bridging the gap between requirements modeling and behavior-driven development. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019* (2019), IEEE, pp. 239–249.

[3] AMMANN, P., AND OFFUTT, J. *Introduction to software testing*. Cambridge University Press, 2016.

[4] APEL, S., AND KÄSTNER, C. An overview of feature-oriented software development. *J. Object Technol. 8*, 5 (2009), 49–84.

[5] ARORA, C., SABETZADEH, M., BRIAND, L. C., AND ZIMMER, F. Automated checking of conformance to requirements templates using natural language processing. *IEEE Trans. Software Eng. 41*, 10 (2015), 944–968.

[6] ARORA, C., SABETZADEH, M., BRIAND, L. C., AND ZIMMER, F. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016* (2016), ACM, pp. 250–260.

[7] ARORA, C., SABETZADEH, M., BRIAND, L. C., AND ZIMMER, F. Automated extraction and clustering of requirements glossary terms. *IEEE Trans. Software Eng. 43*, 10 (2017), 918–945.

[8] AUTILI, M., GRUNSKE, L., LUMPE, M., PELLICCIONE, P., AND TANG, A. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Trans. Software Eng. 41*, 7 (2015), 620–638.

[9] BADAMPUDI, D., WOHLIN, C., AND PETERSEN, K. Software component decision-making: In-house, OSS, COTS or outsourcing - A systematic literature review. *J. Syst. Softw. 121* (2016), 105–124.

[10] BETTINI, L. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing, Birmingham, 2013.

[11] BOEHM, B., AND BASILI, V. Top 10 list [software development]. *Computer 34*, 1 (2001), 135–137.

[12] BOLANDER, P., AND SANDBERG, J. How employee selection decisions are made in practice. *Organization Studies 34*, 3 (2013), 285–311.

[13] CARVALHO, G., FALCÃO, D., DE ALMEIDA BARROS, F., SAMPAIO, A., MOTA, A., MOTTA, L., AND BLACKBURN, M. R. Nat2test$_{\text{scr}}$: Test case generation from natural language requirements based on SCR specifications. *Sci. Comput. Program. 95* (2014), 275–297.

[14] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[15] CRAPO, A. W., MOITRA, A., MCMILLAN, C., AND RUSSELL, D. Requirements capture and analysis in ASSERT(TM). In *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017* (2017), IEEE Computer Society, pp. 283–291.

[16] DENGER, C., BERRY, D. M., AND KAMSTIES, E. Higher quality requirements specifications through natural language patterns. In *2003 IEEE International Conference on Software - Science, Technology and Engineering (SwSTE 2003), 4-5 November 2003, Herzelia, Israel* (2003), IEEE Computer Society, p. 80.

[17] DICK, J., HULL, M. E. C., AND JACKSON, K. *Requirements Engineering, 4th Edition*. Springer, 2017.

[18] DIETTERICH, T. G. Approximate statistical test for comparing supervised classification learning algorithms. *Neural Computation 10*, 7 (1998), 1895–1923.

[19] ECKHARDT, J., VOGELSANG, A., FEMMER, H., AND MAGER, P. Challenging incompleteness of performance requirements by sentence patterns. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016* (2016), IEEE Computer Society, pp. 46–55.

[20] FEMMER, H., FERNÁNDEZ, D. M., JÜRGENS, E., KLOSE, M., ZIMMER, I., AND ZIMMER, J. Rapid requirements checks with requirements smells: two case studies. In *1st International Workshop on Rapid Continuous Software Engineering, RCoSE 2014, Hyderabad, India, June 3, 2014* (2014), ACM, pp. 10–19.

[21] FEMMER, H., FERNÁNDEZ, D. M., WAGNER, S., AND EDER, S. Rapid quality assurance with requirements smells. *J. Syst. Softw. 123* (2017), 190–213.

[22] FERNÁNDEZ, D. M., WAGNER, S., KALINOWSKI, M., FELDERER, M., MAFRA, P., VETRO, A., CONTE, T., CHRISTIANSSON, M., GREER, D., LASSENIUS, C., MÄNNISTÖ, T., NAYABI, M., OIVO, M., PENZENSTADLER, B., PFAHL, D., PRIKLADNICKI, R., RUHE, G., SCHEKELMANN, A., SEN, S., SPÍNOLA, R. O., TUZCU, A., DE LA VARA, J. L., AND WIERINGA, R. J. Naming the pain in requirements engineering - contemporary problems, causes, and effects in practice. *Empir. Softw. Eng. 22*, 5 (2017), 2298–2338.

[23] FLIEDL, G., KOP, C., MAYR, H. C., SALBRECHTER, A., VÖHRINGER, J., WEBER, G., AND WINKLER, C. Deriving static and dynamic concepts from software requirements using sophisticated tagging. *Data Knowl. Eng. 61*, 3 (2007), 433–448.

[24] FOWLER, M. *UML Distilled : a brief guide to the standard object modeling language*, 3rd edition. ed. Addison-Wesley Longman Publishing Co., Inc., 2003.

[25] FUCHS, N. E., KALJURAND, K., AND KUHN, T. Attempto controlled english for knowledge representation. In *Reasoning Web*. Springer, 2008, pp. 104–124.

[26] GÉNOVA, G., FUENTES, J. M., MORILLO, J. L., HURTADO, O., AND MORENO, V. A framework to measure and improve the quality of textual requirements. *Requir. Eng. 18*, 1 (2013), 25–41.

[27] GLASER, B. G. *The discovery of grounded theory : strategies for qualitative research*, [reprinted]. ed. Aldine Transaction, New Brunswick London, 2006.

[28] HULL, E. C., JACKSON, K., AND DICK, J. *Requirements Engineering, Third Edition*. Springer, 2011.

[29] ILIEVA, M. G., AND ORMANDJIEVA, O. Models derived from automatically analyzed textual user requirements. In *Fourth International Conference on Software Engineering, Research, Management and Applications (SERA 2006), 9-11 August 2006, Seattle, Washington, USA* (2006), IEEE Computer Society, pp. 13–21.

[30] INDURKHYA, N., AND DAMERAU, F. *Handbook of natural language processing*, second edition ed. Chapman & Hall/CRC machine learning & pattern recognition series. Boca Raton, Florida : Chapman & Hall/CRC, 2010.

[31] JURETA, I., MYLOPOULOS, J., AND FAULKNER, S. A core ontology for requirements. *Appl. Ontology 4*, 3-4 (2009), 169–244.

[32] KASSAB, M., NEILL, C. J., AND LAPLANTE, P. A. State of practice in requirements engineering: contemporary data. *Innov. Syst. Softw. Eng. 10*, 4 (2014), 235–241.

[33] KIPPER, K., DANG, H. T., AND PALMER, M. S. Class-based construction of a verb lexicon. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA* (2000), AAAI Press / The MIT Press, pp. 691–696.

[34] KONRAD, S., AND CHENG, B. H. C. Facilitating the construction of specification pattern-based properties. In *13th IEEE International Conference on Requirements Engineering (RE 2005), 29 August - 2 September 2005, Paris, France* (2005), IEEE Computer Society, pp. 329–338.

[35] KONRAD, S., AND CHENG, B. H. C. Real-time specification patterns. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA* (2005), ACM, pp. 372–381.

[36] KUHN, T. A survey and classification of controlled natural languages. *Comput. Linguistics 40*, 1 (2014), 121–170.

[37] KUMMLER, P. S., VERNISSE, L., AND FROMM, H. How good are my requirements?: A new perspective on the quality measurement of textual requirements. In *11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018, Coimbra, Portugal, September 4-7, 2018* (2018), IEEE Computer Society, pp. 156–159.

[38] LEVY, R., AND ANDREW, G. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC 2006, Genoa, Italy, May 22-28, 2006* (2006), European Language Resources Association (ELRA), pp. 2231–2234.

[39] LÚCIO, L., RAHMAN, S., CHENG, C., AND MAVIN, A. Just formal enough? automated analysis of EARS requirements. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings* (2017), vol. 10227 of *Lecture Notes in Computer Science*, pp. 427–434.

[40] MANNING, C. D., AND SCHÜTZE, H. *Foundations of statistical natural language processing*. The MIT Press, Cambridge MA London, 1999.

[41] MARCUS, M. P., SANTORINI, B., AND MARCINKIEWICZ, M. A. Building a large annotated corpus of english: The penn treebank. *Comput. Linguistics 19*, 2 (1993), 313–330.

[42] MAVIN, A., AND WILKINSON, P. Big ears (the return of "easy approach to requirements engineering"). In *RE 2010, 18th IEEE International Requirements Engineering Conference, Sydney, New South Wales, Australia, September 27 - October 1, 2010* (2010), IEEE Computer Society, pp. 277–282.

[43] MAVIN, A., WILKINSON, P., GREGORY, S., AND UUSITALO, E. Listens learned (8 lessons learned applying EARS). In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016* (2016), IEEE Computer Society, pp. 276–282.

[44] MAVIN, A., WILKINSON, P., HARWOOD, A., AND NOVAK, M. Easy approach to requirements syntax (EARS). In *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009* (2009), IEEE Computer Society, pp. 317–322.

[45] MÉNDEZ, D., WAGNER, S., KALINOWSKI, M., AND FELDERER, M. Napire: Naming the pain in requirements engineering. http://napire.org.

[46] MILLER, G. A. Wordnet: A lexical database for english. *Commun. ACM 38*, 11 (1995), 39–41.

[47] MORENO-CAPUCHINO, A. M., JUZGADO, N. J., AND VAN DE RIET, R. P. Formal justification in object-oriented modelling: A linguistic approach. *Data Knowl. Eng. 33*, 1 (2000), 25–47.

[48] MU, F., SHI, L., ZHOU, W., ZHANG, Y., AND ZHAO, H. NERO: a text-based tool for content annotation and detection of smells in feature requests. In *28th IEEE International Requirements Engineering Conference, RE 2020, Zurich, Switzerland, August 31 - September 4* (2020), IEEE, pp. 400–403.

[49] OMG. Unified modeling language. version 2.5.1, 2017. Accessed 20 September 2022.

[50] OSAMA, M., ZAKI-ISMAIL, A., ABDELRAZEK, M. A., GRUNDY, J. C., AND IBRAHIM, A. S. Score-based automatic detection and resolution of syntactic ambiguity in natural language requirements. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020* (2020), IEEE, pp. 651–661.

[51] POHL, K. *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, 2010.

[52] POHL, K., AND RUPP, C. *Requirements Engineering Fundamentals - A Study Guide for the Certified Professional for Requirements Engineering Exam: Foundation Level - IREB compliant*. Rocky Nook, 2011.

[53] POST, A., AND HOENICKE, J. Formalization and analysis of real-time requirements: A feasibility study at BOSCH. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings* (2012), vol. 7152 of *Lecture Notes in Computer Science*, Springer, pp. 225–240.

[54] POST, A., MENZEL, I., AND PODELSKI, A. Applying restricted english grammar on automotive requirements - does it work? A case study. In *Requirements Engineering: Foundation for Software Quality - 17th International Working Conference, REFSQ 2011, Essen, Germany, March 28-30, 2011. Proceedings* (2011), vol. 6606 of *Lecture Notes in Computer Science*, Springer, pp. 166–180.

[55] RIAZ, M., KING, J. T., SLANKAS, J., AND WILLIAMS, L. A. Hidden in plain sight: Automatically identifying security requirements from natural language artifacts. In *IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014* (2014), IEEE Computer Society, pp. 183–192.

[56] RUNESON, HÖST, M., RAINER, A., AND REGNELL, B. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012.

[57] SADRAEI, E., AURUM, A., BEYDOUN, G., AND PAECH, B. A field study of the requirements engineering practice in australian software industry. *Requirements Engineering 12*, 3 (Jul 2007), 145–162.

[58] SALDAÑA, J. *The coding manual for qualitative researchers.* Sage, 2015.

[59] SCHIENMANN, B. *Kontinuierliches Anforderungsmanagement : Prozesse - Techniken - Werkzeuge*. Programmers's choice. Addison-Wesley, München, 2002.

[60] SEKI, Y., HAYASHI, S., AND SAEKI, M. Detecting bad smells in use case descriptions. In *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019* (2019), IEEE, pp. 98–108.

[61] SELVYANTI, D., AND BANDUNG, Y. The requirements engineering framework based on iso 29148: 2011 and multi-view modeling framework. In *2017 International Conference on Information Technology Systems and Innovation (ICITSI)* (2017), IEEE, pp. 128–133.

[62] SMIALEK, M., BOJARSKI, J., NOWAKOWSKI, W., AMBROZIEWICZ, A., AND STRASZAK, T. Complementary use case scenario representations based on domain vocabularies. In *Model Driven*

Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings (2007), vol. 4735 of *Lecture Notes in Computer Science*, Springer, pp. 544–558.

[63] SOLEMON, B., SAHIBUDDIN, S., AND GHANI, A. Requirements engineering problems and practices in software companies: An industrial survey. In *Advances in Software Engineering - International Conference on Advanced Software Engineering and Its Applications, ASEA 2009* (2009), vol. 59, Springer, pp. 70–77.

[64] SOMMERVILLE, I. *Software engineering*, 9th ed. ed. Pearson, Boston, 2011.

[65] SOMMERVILLE, I., AND SAWYER, P. Requirements engineering: a good practice guide. *John Wiley and Sons 113* (1997), 114.

[66] STEVENSON, A., AND CORDY, J. R. A survey of grammatical inference in software engineering. *Sci. Comput. Program. 96* (2014), 444–459.

[67] STOL, K., RALPH, P., AND FITZGERALD, B. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (2016), ACM, pp. 120–131.

[68] THAKUR, J. S., AND GUPTA, A. Identifying domain elements from textual specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016* (2016), ACM, pp. 566–577.

[69] THE STANDISH GROUP. The chaos report, 1995-2019. Accessed 20 September 2022.

[70] TOUTANOVA, K., KLEIN, D., MANNING, C. D., AND SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2003, Edmonton, Canada, May 27 - June 1, 2003* (2003), The Association for Computational Linguistics.

[71] VEIZAGA, A., ALFÉREZ, M., TORRE, D., SABETZADEH, M., AND BRIAND, L. C. On systematically building a controlled natural language for functional requirements. *Empir. Softw. Eng. 26*, 4 (2021), 79.

[72] VEIZAGA, A., ALFÉREZ, M., TORRE, D., SABETZADEH, M., BRIAND, L. C., AND PITSKHELAURI, E. Leveraging natural-language requirements for deriving better acceptance criteria from models. In *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020* (2020), ACM, pp. 218–228.

[73] WANG, C., PASTORE, F., GOKNIL, A., BRIAND, L. C., AND IQBAL, M. Z. Z. UMTG: a toolset to automatically generate system test cases from use case specifications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015* (2015), ACM, pp. 942–945.

[74] WITHALL, S. *Software requirement patterns*. Pearson Education, 2007.

[75] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., AND REGNELL, B. *Experimentation in Software Engineering*. Springer, 2012.

[76] WYNNE, M., AND HELLESOY, A. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017.

[77] YOUNG, R. The main thing is keeping the main thing the main thing. *Requirements Engineering Magazine 1* (2015).

[78] YUE, T., BRIAND, L. C., AND LABICHE, Y. A systematic review of transformation approaches between user requirements and analysis models. *Requir. Eng. 16*, 2 (2011), 75–99.

[79] YUE, T., BRIAND, L. C., AND LABICHE, Y. Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Trans. Softw. Eng. Methodol. 22*, 1 (2013), 5:1–5:38.

[80] YUE, T., BRIAND, L. C., AND LABICHE, Y. aToucan: An automated framework to derive UML analysis models from use case models. *ACM Trans. Softw. Eng. Methodol. 24*, 3 (2015), 13:1–13:52.

[81] ZAVE, P., AND JACKSON, M. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol. 6*, 1 (1997), 1–30.

[82] ZHAO, L., ALHOSHAN, W., FERRARI, A., LETSHOLO, K. J., AJAGBE, M. A., CHIOASCA, E., AND BATISTA-NAVARRO, R. T. Natural language processing for requirements engineering: A systematic mapping study. *ACM Comput. Surv. 54*, 3 (2021), 55:1–55:41.

[83] ZIKMUND, W., BABIN, B., CARR, J., AND GRIFFIN, M. *Business Research Methods*. Cengage Learning, 2013.