Automated Smell Detection and Recommendation in Natural Language Requirements

Alvaro Veizaga, Seung Yeob Shin, *Member, IEEE,* Lionel C. Briand, *Fellow, IEEE*

Abstract—Requirement specifications are typically written in natural language (NL) due to its usability across multiple domains and understandability by all stakeholders. However, unstructured NL is prone to quality problems (e.g., ambiguity) when writing requirements, which can result in project failures. To address this issue, we present a tool, named Paska, that takes as input any NL requirements, automatically detects quality problems as smells in the requirements, and offers recommendations to improve their quality. Our approach relies on natural language processing (NLP) techniques and a state-of-the-art controlled natural language (CNL) for requirements (Rimay), to detect smells and suggest recommendations using patterns defined in Rimay to improve requirement quality. We evaluated Paska through an industrial case study in the financial domain involving 13 systems and 2725 annotated requirements. The results show that our tool is accurate in detecting smells (89% precision and recall) and suggesting appropriate Rimay pattern recommendations (96% precision and 94% recall).

Index Terms—requirement smells, requirement quality, smell detection and recommendation, natural language processing, and controlled natural language

✦

1 INTRODUCTION

Requirements typically drive software development and are generally expressed using natural language (NL), which is widely used in many industrial contexts. The central role of NL in software requirement specifications (SRSs) stems from its usability in all application domains and its ease of understanding by all stakeholders in software development projects [1]. A study [2] reports that 61% of users prefer to express requirements using NL. However, despite its popularity, NL is highly prone to quality problems, such as vagueness, ambiguity, complexity, and incompleteness [3, 4].

One important cause of project failures in industry is quality problems in NL requirements [5, 6]. When such problems are not fixed early during development, they carry over to subsequent development phases, and fixing them becomes a costly and time-consuming process. Improving the quality of NL requirements by identifying quality problems at early development stages is therefore a pivotal need for successful software development.

In collaboration with an industrial partner, which provides post-trade services for various types of financial securities and develops information systems to support these

- *Alvaro Veizaga is with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) of the University of Luxembourg, Luxembourg (e-mail: alvaro.veizaga@uni.lu).*
- *Seung Yeob Shin is with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) of the University of Luxembourg, Luxembourg (e-mail: seungyeob.shin@uni.lu).*
- *Lionel C. Briand holds shared appointments with the Lero SFI Centre for Software Research, University of Limerick, Ireland and the school of EECS, University of Ottawa, Ottawa, Canada (e-mail: lionel.briand@lero.ie). Part of this work was done when he was affiliated with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) of the University of Luxembourg.*

services, we noticed elevated costs associated with in-house processes to improve the quality of NL requirements, which typically involve several manual inspections and are thus prone to errors. A tool that automatically detects quality problems in NL requirements and guides analysts to improve the quality of NL requirements is thus highly desirable.

Various approaches have been proposed to improve the quality of NL requirements by detecting semantic and syntactic problems, that are often referred as "smells" [7, 8, 9, 10, 11]. For example, Femmer et al. [9] introduced Smella, an automated tool for detecting requirement smells such as ambiguous adverbs and vague pronouns. Smella relies on part-of-speech (POS) tagging, dictionaries, and lemmatization. However, these approaches do not provide recommendations to analysts on how to rewrite requirements in a disciplined manner to improve their quality. Furthermore, existing work [9, 10], which detects a set of quality problems in a requirement, still requires further research to account for many of the recurrent problems faced by analysts. For example, analysts sometimes describe multiple functions in a single requirement (i.e., non-atomic requirement), miss essential words (e.g., actors and verbs) or even phrases (e.g., system responses), or write a requirement following an ambiguous structure (e.g., a system response between conditions). We note that some of these quality problems have been studied individually in many prior works, such as checking the completeness [12, 13, 14, 15, 16] or ambiguity [17, 18, 19, 11, 20] of requirements. However, compared to these research strands, there has been relatively less focus on developing an automated solution that can both detect and resolve multiple quality problems in a requirement. Such a solution is needed to provide a complete picture of the overall requirement quality and thus enable

the proposal of a solution that properly fixes all relevant problems together.

To address the challenges stated above, we developed an approach and tool that addresses common quality problems in NL requirements. In particular, our work aims at assisting analysts with automatically detecting quality problems and suggesting recommendations to fix them in functional requirements. Given our focus on information systems, a functional requirement specifies what system response an actor is expected to receive when providing certain inputs, if certain conditions are met [21]. We have named our tool Paska, which means "solution" in Quechua. In this article, the term "smells" refers to these quality problems in functional requirements that can lead to defects at different levels of severity. For example, while the use of passive voice alone in an NL requirement is not necessarily a direct cause of defects, it can contribute to communication issues and misunderstandings, which can in turn increase the risk of defects. To illustrate this point, we present a real-world example of an NL requirement provided by our industrial partner in the financial domain, anonymized for confidentiality: *"When an order cancellation message is received then the System_A GUI must display the field Reason_of_Cancellation"*. The condition of this NL requirement *"When an order cancellation message is received"* is in passive voice. In this requirement, it is difficult for stakeholders to identify the actor receiving the *"order cancellation message"*, which can result in defects in the final product. In a functional requirement, a smell indicates a problem located in a specific word or segment (i.e., set of words) that quality analysts should inspect [9]. Paska detects nine smells that are commonly present in functional requirements we analyzed in the financial domain, though they are not in any way specific to this domain. Most importantly, Paska also provides suggestions for fixing smells and thus improving requirement quality.

Contributions. To automatically detect quality problems in functional requirements and provide concrete improvement recommendations, our work relies on a state-of-the-art controlled natural language (CNL) called Rimay [21], which is used to specify functional requirements for information systems. Rimay requires analysts to write requirements in a disciplined manner using controlled grammar and vocabulary, ensuring they follow best practices for writing requirements. This results in writing precise, unambiguous, complete, and atomic requirements, which in turn enables automated analysis such as traceability analysis [22] and acceptance criteria generation [22]. However, defining a language like Rimay is a different problem than applying it to detect quality problems in non-compliant requirements and propose recommendations for their rectification. We first define requirement smells corresponding to violations of quality attributes, such as completeness, clarity, atomicity, and correctness, which Rimay enforces when writing requirements. Based on the Rimay grammar, we define a set of Rimay patterns that capture recommended structures for expressing requirements. In order to help analysts address smells in a requirement, Paska then provides Rimay patterns as suggestions. Analysts can then rewrite these requirements based on the recommended patterns. Indeed, such patterns indicate what requirement segments are missing, warrant change, or must be re-ordered. To automate the detection of requirement smells and the recommendation of Rimay patterns, Paska combines natural language processing (NLP) techniques, including tokenization, lemmatization, POS tagging, constituency parsing, glossary search, and Tregex [23]. We note that this article not only presents an automated method for identifying a large spectrum of smells but, different from existing work, also provides automated and accurate guidance on how to address them.

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

RQ1) What are the NL requirement smells commonly found in the financial domain? We answer RQ1 by identifying nine smells after analyzing a set of 1404 actual requirements in the financial domain. The smells indicate quality problems in requirements regarding completeness, clarity, atomicity, and correctness. Paska is able to detect quality problems in both 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. It combines Tregex [23] patterns, structural patterns, rules, and glossary search.

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

RQ4) Can Paska accurately indicate the occurrence of smells? To answer RQ4, we evaluate Paska by measuring the accuracy of identifying smells. To accomplish this, we conducted a case study using 13 SRSs from financial applications. We compared our results against a ground truth established manually by four annotators. Our evaluation results suggest that Paska accurately detects smells with a precision and a recall of 89%.

RQ5) How accurate is Paska in recommending requirement patterns to fix smells? RQ5 assesses how well Paska suggests appropriate Rimay patterns to fix smells in NL requirements. We compared the performance of Paska against a ground truth matching requirements with smells and Rimay patterns, which were manually annotated by four annotators. The results show that Paska accurately suggests appropriate Rimay patterns with a precision of 96% and a recall of 94%.

To summarize, the main contributions of this work are (1) an automated approach that detects smells in SRSs and suggests Rimay patterns to fix these smells and (2) an extensive industrial case study on smell detection, involving a large set of 2725 information system requirements from 13 projects in the financial domain, to assess the accuracy of Paska when detecting smells and suggesting patterns to fix them. Our industrial case study is the largest to date regarding requirements quality assurance.

Organization. The article is structured as follows: Section 2 provides an overview of Rimay. Section 4 describes

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

Listing 1: Overall syntax of a requirement in Rimay.

```
1  SCOPE: For MODIFIER? TEXT (and MODIFIER? TEXT)?,
2  CONDITION_STRUCTURE: WHILE_STRUCTURE|WHEN_STRUCTURE|↩
       WHERE_STRUCTURE|IF_STRUCTURE|TEMPORAL_STRUCTURE
3  WHILE_STRUCTURE: While PRECONDITION_STRUCTURE
4  WHEN_STRUCTURE: When TRIGGER
5  WHERE_STRUCTURE: Where TEXT
6  IF_STRUCTURE: If PRECONDITION_STRUCTURE|TRIGGER
7  TEMPORAL_STRUCTURE: (Before|After|Every) TEXT
8  CONDITION_STRUCTURES: CONDITION_STRUCTURE ( (,|and|or↩
       |,or|,and) CONDITION_STRUCTURE)*, then?
```

Listing 2: Syntax of scope and condition structures in Rimay.

how we derive smells and Rimay patterns. In Section 5, we describe Paska in detail. Section 6 evaluates Paska through an industrial case study. Section 7 examines some issues that impact the performance of Paska. Sections 8 and 9 discuss threats to the validity of our results and related work. Finally, Section 10 concludes this article.

2 CONTROLLED NATURAL LANGUAGE: RIMAY

In this section, we introduce Rimay as our work relies on it to define smells and patterns (Section 4), as well as to identify smells and recommend patterns (Section 5). Rimay is a controlled natural language (CNL) for writing functional requirements in the domain of information systems [21]. Rimay applies restrictions on vocabulary, grammar, and semantics of NL to allow analysts to write complete, unambiguous, and precise requirements. Rimay's main grammar rules are inspired by the Easy Approach to Requirements Syntax (EARS) templates [24]. Many practitioners consider EARS as a good trade-off between flexibility and precision, due to EARS's relatively low training overhead and the quality and readability of the resultant requirements [25]. However, EARS templates, which consist of predefined sentence structures with general, coarse-grained concepts and constructs, are not amenable to the type of analyses enabling task automation because they allow the introduction of unstructured and ambiguous text. In contrast to EARS, Rimay provides sentence structures with more specialized, precise, fine-grained concepts and constructs, enabling automated analysis such as reconciliation support between requirement text and models [22] and automated acceptance criteria generation [22].

Listing 1 shows the rule REQUIREMENT specifying the overall syntax of a requirement in Rimay. The rule indicates that the presence of SCOPE and CONDITION_STRUCTURES is optional, whereas the presence of ACTOR, MODAL_VERB and SYSTEM_RESPONSE is mandatory in all requirements. In a functional requirement, an actor is expected to achieve a system response under certain conditions. An actor is a role played by an entity that interacts with the system by exchanging signals, data, or information [26]. Furthermore, requirements written in Rimay may have a scope to delimit the effects of the system response. For simplicity, in this study, we use "system response" to refer to a phrase consisting of an actor, a modal verb, and a system response.

The Rimay grammar enables analysts to write a wide variety of functional requirements while ensuring that they follow recommended syntactic structures [21]. For example, Listing 2 depicts the grammar rules for scope and condition structures. As shown in the listing, Rimay's grammar has some common constructs such as MODIFIER (line 1). The construct MODIFIER includes articles (e.g., "a", "an", and "the") and quantifiers (e.g., "each", "all", "none", "only one", and "any"). The rule SCOPE (line 1) uses the keyword For, along with the rules MODIFIER and TEXT.

The rule CONDITION_STRUCTURE (line 2 of Listing 2) defines different ways to use system states, trigger events, and features to express conditions. In a functional requirement, such conditions must hold for the system response to be triggered. Furthermore, listing 2 (lines 3-7) shows the rules for the following condition structures: WHILE, WHEN, WHERE, IF, and TEMPORAL structures. 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 in the system. The WHERE_STRUCTURE is used for system responses that are triggered only when the system includes particular features. These features are described in free form using the rule TEXT. While Rimay provides fine-grained constructs, it still includes the rule TEXT to handle situations where the use of free text is necessary or desirable. The IF_STRUCTURE is used when a specific triggering event happens, or when the system is in a particular state before triggering any system responses. The TEMPORAL_STRUCTURE is used when the system responses are triggered before or after an event. Line 8 of Listing 2 shows the rule CONDITION_STRUCTURES that enables the creation of a condition composed of two or more of the conditions mentioned above (lines 3-7).

Table 1 lists Rimay examples of requirement conditions, i.e., WHEN_STRUCTURE, TEMPORAL_STRUCTURE, and IF_STRUCTURE, as well as SYSTEM_RESPONSE. Note that these examples are independent from one another. The WHEN_STRUCTURE and TEMPORAL_STRUCTURE examples capture the events that trigger system responses, whereas the IF_STRUCTURE example specifies a precondition required for system responses. The SYSTEM_RESPONSE example specifies the action a user must take.

Thanks to its restrictions on vocabulary, grammar, and semantics of NL, Rimay allows analysts to write functional requirements that satisfy the following quality attributes: *completeness*, *clarity*, *atomicity*, and *correctness*. *Completeness* refers to the inclusion of all the information required for the requirement to be complete. Rimay achieves completeness by having mandatory constructs that ensure the presence of certain contents. Note that any omission of these constructs will result in syntax errors in a Rimay requirement, making it syntactically incomplete. For example, Rimay does not allow analysts to write a requirement without a system response. Such a requirement is thus syntactically incomplete in Rimay. *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 restricted vocabulary. *Atomicity* refers to ensuring that an NL require-

TABLE 1: Examples of when, temporal, and if conditions, as well as a system response written in Rimay.

Rimay syntax	Rimay example
WHEN_STRUCTURE	When System-B receives an "email alert" from System-A
TEMPORAL_STRUCTURE	Before System-A sends an "Instruction" to System-B
IF_STRUCTURE	If an "Instruction" contains a "Keyword"
SYSTEM_RESPONSE	The User must upload the "excel file" to System-A

ment describes a single system function. Rimay enforces that a requirement does not have more than one system response. *Correctness* refers to the proper use of Rimay's syntax, i.e., the correct arrangements of words and phrases. For example, Rimay does not allow the use of modal verbs in conditions.

We note that these quality attributes, particularly completeness and clarity, align with those prominently studied in the literature [27], indicating they correspond to commonly observed quality issues in requirements engineering. Specifically, the completeness of a functional requirement in Rimay (thereafter a Rimay requirement) is based on prior studies [9, 10] on the internal completeness [28] of a requirement, which concerns the self-containment of a requirement. The clarity of a Rimay requirement is related to research strands [19, 20] on analyzing requirement ambiguities. We further discuss these research contributions in Section 9. Regarding the atomicity of a Rimay requirement, it contributes to improving the other quality attributes, such as clarity and verifiability. For example, a Rimay requirement, precisely describing a single system function, allows for the automatic generation of acceptance criteria for the system function [21, 22]. This, in turn, enhances the traceability between a functional requirement in Rimay and its associated acceptance criteria. However, the correctness of a Rimay requirement, in this article, is different from the notion of correctness in existing works [29], which concerns whether a requirement accurately captures users' needs. In our context, as described earlier, the correctness of a Rimay requirement pertains to the correct arrangements of words and phrases defined in the Rimay rules. Such correct arrangements of words and phrases enable analysts to write requirements in a consistent and unambiguous manner.

Rimay is implemented as an add-on editor for Sparx Enterprise Architect [30] with the following features: (a) syntax highlighting to color requirements and format them with different visual styles according to the elements of Rimay, (b) error marking to automatically highlight the parts of the requirements indicating errors, and (c) content assisting to automatically, or on request, provide suggestions to analysts on how to complete the statement based on the grammar rules.

3 INDUSTRIAL PARTNER AND DATA COLLECTION

We conducted this study in collaboration with an industrial partner, who provided the actual requirements for this study and gave feedback while developing Paska. Below, we provide more details about the context of our study and the dataset obtained from our industrial partner.

3.1 Industrial Partner

Our industrial partner is a leading financial company that provides post-trade services for various types of financial securities and develops information systems to support these services, serving 2500 customers in 110 countries. Specifically, our collaboration was with their specialized financial services division. This division is responsible for several tasks, including the development of new financial applications, the maintenance of existing ones, and the enhancement of applications using advanced technologies. Their goal is to provide clients with cutting-edge solutions while ensuring compliance with current regulations. The teams within the division cover project management, service operations, development, testing, and requirements analysis.

The company employs a methodology rooted in best practices and years of experience for the tasks mentioned above. For example, financial analysts use natural language (NL) combined with Unified Modeling Language (UML) models to specify requirements. Their textual NL requirements, written in English, aim to adhere to the Rupp template [31]. The company follows a carefully planned software development process grounded in the V-Model [32], tailored for stringently regulated industries such as finance.

We note that the first author was already familiar with the development process, including requirements engineering, of the company, before starting this work, through participating in training sessions and attending numerous meetings. In addition, all the authors of this article interacted with the industrial partner during the project period through regular bi-weekly meetings and additional meetings as needed to obtain feedback on our progress in defining the problem and developing Paska. For the meetings, the number of industry participants varied between one and five, depending on their availability and interests. They collectively have over 50 years of industry experience with significant expertise in business analysis, functional design, project management, and requirements engineering.

The research team of this article collaborated with the company for approximately 1.5 years. Our collaboration covered defining the problem this work addresses, identifying smells in requirements, defining Rimay patterns, and developing an automated approach for smell detection and pattern recommendation. Hence, our work was motivated by practitioners' needs and was validated in a practical context. However, given its scale, the evaluation of the Paska prototype was completed after the project ended, including data annotation, implementation, and evaluation. Since Paska is an automated analysis tool, which does not require human intervention, we were able to successfully evaluate Paska without significant involvement from our industrial partner.

TABLE 2: Distribution of actual requirements received from our industrial partner.

Subset	SRS ID	# Requirements	# Words
S^D	SRS1	192	6363
	SRS2	188	7462
	SRS3	118	5699
	SRS4	161	7925
	SRS5	451	22057
	SRS6	294	14573
S^E	SRS7	367	9512
	SRS8	90	3573
	SRS9	167	6211
	SRS10	192	8892
	SRS11	19	331
	SRS12	340	15448
	SRS13	146	7539
Total		2725	115585

3.2 Data Collection

To conduct this study, we first collected actual NL requirements from our industrial partner. Financial analysts provided us with a set of 13 representative SRSs, written by different analysts and containing various numbers of requirements. These SRSs describe various types of projects, including functional updates to existing applications, making existing applications compliant with new regulations, creating new applications from scratch, and the migration of existing applications to new platforms. We refer to these SRSs as SRS1-SRS13, belonging to set S. They contain 2725 requirements in total, which makes this industrial case study the largest to date regarding requirements quality assurance.

Table 2 shows the distribution of requirements, along with the total word count, across SRSs. We randomly split the set S of 2725 requirements into two subsets S^D and S^E, containing 1404 and 1321 requirements, respectively. The subset S^D contains SRSs SRS1-SRS6 and was used for developing Paska (see Sections 4 and 5). The subset S^E contains SRSs SRS7-SRS13 and was used for evaluating Paska (see Section 6).

4 REQUIREMENTS SMELLS AND RIMAY PATTERNS

This section describes the process we conducted to derive requirements smells and patterns from NL requirements and the Rimay language. We define a catalog of smells that identify the syntactic and semantic errors commonly found in NL requirements. In addition, we introduce Rimay patterns that describe predefined structures for high-quality requirements expressed in Rimay. These patterns will be used as suggestions for analysts to rewrite requirements containing smells, thereby improving the quality of the requirements.

4.1 Requirements Smells

This section aims to answer RQ1: **What are the NL requirement smells commonly found in the financial domain?** For RQ1, we inspected a set S^D of 1404 NL requirements, which were written by different analysts across six financial systems, as described in Section 3.2. In order to review the

requirements in a consistent manner, we used the quality attributes enforced by Rimay as our review criteria. These criteria enabled us to identify requirements that have quality problems with respect to completeness, clarity, atomicity, and correctness. In this article, the term "requirement smell" refers to these quality problems in functional requirements that may lead to misunderstandings, which can, in turn, increase the risk of defects in the product. We note that there are existing collections of requirement smells reported in prior work [9, 10], which are further discussed in Section 9. Drawing upon these works, we cross-referenced our findings with these existing smell collection. In our context, for the reasons mentioned above, we apply Rimay to support a practical solution to detect a large variety of common smells and rectify them. Hence, we chose to inspect the 1404 NL requirements using the quality attributes enforced by Rimay.

In our inspection process, the first author of this article manually reviewed the requirements to uncover a list of those that violate the quality attributes and to define a catalog of smells. We note that this author has extensive expertise and practical experience in Rimay from defining the language to (re)writing requirements in it, as presented in the author's previous work [21]. Given this background, the author was ideally suited to review the requirements and pinpoint violations of the quality attributes that Rimay enforces. We randomly divided the set S^D of 1404 requirements into six distinct subsets for incremental inspection. Specifically, S_1^D contains 35% of S^D, and each S_i^D contains 13% of S^D, where $i \in \{2, \ldots, 6\}$. We then sequentially inspected each subset of requirements in the order S_1^D, S_2^D, ..., S_6^D. During each inspection, we identified requirements that violate the quality attributes and characterized these violations as requirement smells. For example, if an inspected requirement had multiple actions in its system response, we identified it as violating atomicity and characterized the violation as the non-atomic requirement smell. After inspecting each subset of requirements, we obtained feedback from our industrial partner to validate that the catalog of smells we defined aligns with common errors frequently made by analysts when writing NL requirements. In addition to such feedback, the other authors closely monitored progress through regular meetings. The catalog of smells was thus continuously refined and expanded throughout the inspection process.

We applied the notion of saturation [33] to determine when to stop the inspection process. Saturation occurs when no new information can be gained from the data being analyzed. We continued the inspection process as long as we detected any instances of new smells that were not already listed in our smell catalog. From this process, we defined a catalog of nine smells: non-atomic requirement, incomplete requirement, incorrect order requirement, coordination ambiguity, not requirement, incomplete condition, incomplete system response, passive voice, and not precise verb. Table 3 lists the nine smells. The first column shows the smell names, and the second column provides the smell descriptions and examples. We note that the examples listed in this table are derived from the actual requirements used in our case study. However, they are sanitized for confidentiality. The third column indicates the quality attribute that each smell violates. What is noteworthy from the nine smells

TABLE 3: Catalog of nine smells. The examples in this table are derived from the actual requirements used in our case study but are anonymized for confidentiality.

Smell name	(D) Description and (E) Example	Quality Attribute
Non-atomic requirement	(D) Non-atomic requirement refers to a requirement that has more than one action in the system response. (E) *"System-A must add System-B to their downstream systems and allow System-C to subscribe to the Reporting flow."*	Atomicity
Incomplete requirement	(D) Incomplete requirement refers to a requirement that does not have a system response but has other optional segments, i.e., condition and scope. (E) *"When System-A receives a message from Security Manager and if the message is part of the B-file, according to the mapping rules."*	Completeness
Incorrect order requirement	(D) Incorrect order requirement refers to a requirement that its condition segment is located after its system response. This arrangement of requirement segments may lead to a vague interpretation of the condition. (E) *"When the user is on the Utilities page and the user clicks on the button "Display on main page", System-A must open the Alert section when the user launches System-A."*	Correctness
Coordination ambiguity	(D) Coordination ambiguity refers to a requirement that has two or more conditions and these conditions are connected by a coordinated conjunction "or". (E) *"When System-A performs eligibility check for a participant or if the holding type is complex or if the holding type is simple and the F-value is Prime, then System-B must ..."*	Clarity
Not requirement	(D) Not requirement refers to a statement that does not contain any requirement segment, i.e., scope, condition, and system response. (E) *"The R6 instruction defines the original instruction."*	Correctness
Incomplete condition	(D) Incomplete condition indicates a condition that lacks either actor or verb. (E) *"Upon receipt of a message in the message Queue, System-A must set the state to unprocessed."*	Completeness
Incomplete system response	(D) Incomplete system response refers to a system response that lacks either an actor, a modal verb, or a verb. (E) *"When the user clicks on the Filter button, System-A opens the Filter screen."*	Completeness
Passive voice	(D) Passive voice indicates that the condition or system response in a requirement is described in the passive voice. Such requirements likely miss actors. (E) *"When a rejection order is received for a cancellation request, System-A must raise a web alert."*	Completeness
Not precise verb	(D) Not precise verb refers to a verb used in the condition or system response in a requirement that is not precise enough. Such a verb does not define a precise action. The list of our not precise verbs includes: "accomplish", "account", "come", "consider", "default", "define", "do", "get", "make", "perform", "process", "propose", "raise", "read", "support", and "want". This list is curated by the analysts at our industrial partner. (E) *"System-A must be able to process System-B's instructions with input media INPUT."*	Clarity

we identified is that they seem rather generic and probably applicable to information systems in other domains.

> *The answer to RQ1 is that*, based on our inspection, the following NL requirement smells are commonly found in the financial domain: non-atomic requirement, incomplete requirement, incorrect order requirement, coordination ambiguity, not requirement, incomplete condition, incomplete system response, passive voice, and not precise verb. These smells violate atomicity, completeness, correctness, and clarity of NL requirements. None of these smells appears to be specific to the financial domain.

4.2 Rimay Patterns

This section describes the process conducted to derive requirements patterns from the Rimay language. These patterns describe predefined structures of Rimay concepts. As we describe in Section 2, Rimay provides specialized concepts and constructs to specify functional requirements. However, Rimay does not provide patterns that guide analysts on how to rewrite a requirement with smells.

To derive the Rimay patterns, we first created a conceptual model for capturing the concepts and their relations underlying the Rimay language. Figure 1 shows the conceptual model of Rimay. The model captures five concepts to define "Requirement' as follows: "Scope", "Condition_Structure", "Actor", "Modal_Verb", and "Action_Phrase". "Condition_Structure" is further specialized into five concepts: "While_Structure", "If_Structure", "When_Structure", "Where_Structure",

and "Temporal_Structure". Rimay restricts the ways in which these conditions can be expressed. For example, "While_Structure" can be used for system responses that are triggered while the system is in a particular state (see the relation between "While_Structure" and "Precondition" in Figure 1). The concept "Action_Phrase" is specialized into 58 concepts (e.g., "ADMIT_65" and "BEG_58_2") that correspond to the grammar rule names defined in Rimay [21]. Note that the grammar rules of the "Action_Phrase" concept describe the syntactic structure and vocabulary allowed. Figure 1 shows only a few concepts for "Action_Phrase".

Table 4 presents examples of Rimay action phrases corresponding to the concepts that specialize the "Action_Phrase" concept in Figure 1. For example, "BEG_58_2" restricts the usage of the word "request" in expressing an action phrase. Specifically, the Rimay syntax of "BEG_58_2" restricts that "request" can be followed by an optional modifier, a mandatory actor, an optional "for" or "to" phrase, and an optional "by using" phrase, in this specific order. The example `request the System to "cancel the settlement" by using the "Order Reference"` follows the Rimay syntax. We note that the exact Rimay syntax for action phrases, along with examples of Rimay action phrases, is provided in the previous work on Rimay [21].

To identify Rimay patterns, we utilized both the Rimay conceptual model (Figure 1) and grammar. We traversed the syntax tree of the grammar to derive possible combinations of Rimay concepts. In Rimay, a functional requirement is composed of scope, condition, and
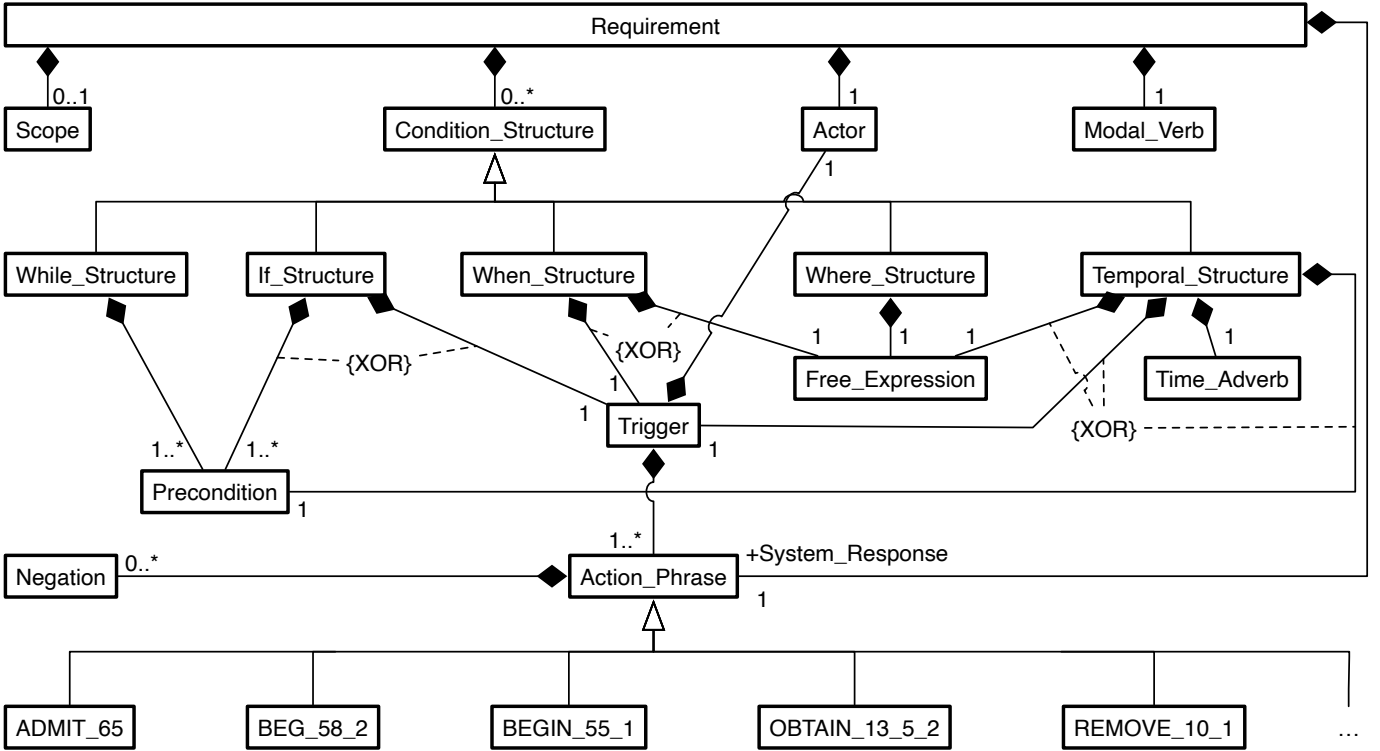
Fig. 1: Rimay conceptual model.

TABLE 4: Concepts of action phrases (shown in Figure 1) and their Rimay examples (excerpted from the previous work on Rimay [21]).

Concept	Example: Rimay Action Phrase
ADMIT_65	exclude the "Gregorian dates that are not business days" in the System based on "the ↩ relevant calendar"
BEG_58_2	request the System to "cancel the settlement" by using the "Order Reference"
BEGIN_55_1	start the "calculation of the next NAV date on daily basis"
OBTAIN_13_5_2	receive a DA_file from CFCL_IT
REMOVE_10_1	delete the "DECU field" from the "Settlement Parties block"

system response segments. These segments correspond to the "Scope", "Condition_Structure", and "Action_Phrase" concepts, respectively, as shown in Figure 1. According to the Rimay grammar (see Listing 1), the scope, condition, and system response segments of a functional requirement must appear in that specific order. In addition, Rimay specifies that the scope and condition segments are optional in a Rimay requirement, while the system response segment is mandatory. Regarding the condition segment in a Rimay requirement, analysts can express the following three condition concepts: "Precondition", "Trigger", and "Temporal_Structure". These concepts respectively correspond to a condition in which the requirement can be invoked ("Precondition"), an event that initiates the requirement ("Trigger"), and a temporal event that occurs either before or after the requirement's invocation ("Time_Adverb"). Furthermore, Rimay allows a condition segment to specify multiple conditions. Hence, in Rimay, analysts have the flexibility to specify a condition segment as either a precondition, a trigger condition, a time condition, or a combination of these, referred to as multiple conditions. Since Rimay provides analysts with 2 options for scope

(i.e., inclusion and exclusion), 5 options for condition (i.e., inclusion — precondition, trigger condition, time condition, and multiple conditions — and exclusion), and 1 option for system response (i.e., inclusion), a Rimay requirement can be classified into 10 distinct patterns, as shown in Table 5.

Table 5 outlines the 10 Rimay patterns derived by combining the concepts captured in the conceptual model of Rimay. The first column shows the pattern names. The second column specifies the pattern in the Rimay syntax. Finally, the third column indicates the combination of the Rimay concepts used to derive the pattern. Table 5 does not include all the keywords defined in Rimay and does not include the templates for Action_Phrases. We refer interested readers to our previous work on Rimay [21] for a complete reference to the concepts and constructs of the Rimay language.

The derived patterns will be used by Paska to provide suggestions to analysts when Paska detects any smells (see Table 3) in NL requirements. The patterns, as guidance, are intended to help analysts rewrite the NL requirements in Rimay to improve their quality. Indeed, such patterns indicate what requirement segments are missing, warrant change, or must be re-ordered. Recall that Rimay allows ana-

TABLE 5: 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

lysts to write requirements that satisfy the following quality attributes: completeness, clarity, correctness, and atomicity. Since Paska detects requirement smells that violate these quality attributes, proposing appropriate Rimay patterns will assist analysts in addressing the identified smells in NL requirements. We note, however, that Paska is not designed to automatically rewrite NL requirements in Rimay to rectify any smells. Such automatic translation is probably not possible in general as it would require automated additions of missing segments or changes, which we leave out of the scope of this study.

5 APPROACH

Figure 2 provides an overview of the four steps of Paska. The inputs are (1) software requirements specifications (SRSs) that contain a set of functional requirements, (2) Rimay patterns (Section 4.2), and (3) the catalog of nine smells commonly found in NL requirements (Section 4.1).

As shown in Figure 2, in Step 1, Paska applies preprocessing steps to the NL requirements extracted from the SRSs. In Step 2, requirements are separated into segments (e.g., scope, condition, and system response), relying on patterns and a segment splitter. The patterns are written using Tregex, which is a language for defining patterns in text syntax trees. In Step 3, Paska detects smells in NL requirements by applying several techniques, such as Tregex patterns, structural 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, Paska suggests a pattern for analysts to fix the NL requirements with smells and convert them into Rimay. Rimay helps reduce the risk of having quality problems in requirements since it has precise syntax and semantics. In

this section, we describe in detail all the steps of Paska using the running example shown in Figure 3.

5.1 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 but keep the metadata in requirements (i.e., the MS Word metadata in our case study). Such metadata includes line breaks and bullet points and are, in our context, useful for detecting multiple conditions and system responses. Figure 3 (Step 1) shows an example of a preprocessed requirement. Paska removes the double quotes since we observe that single and double quotation marks prevent us from correctly identifying the structural units of the sentences when using a constituency parsing algorithm (e.g., AllenNLP [34]).

5.2 Step 2: Separate Requirement into Segments

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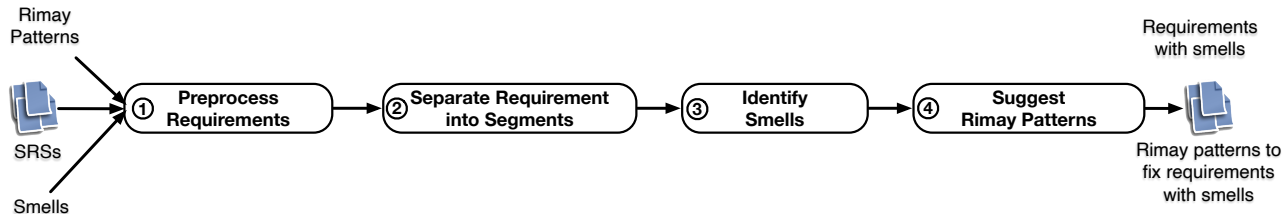
Fig. 2: Approach overview.

Tregex patterns. The Tregex query language allows users to define regular expression-like patterns in tree structures [23]. Tregex is designed to define patterns that involve the tree nodes and the hierarchical relations among the tree nodes in the syntax tree of a requirement. To separate a requirement into segments, we created a set of patterns using Tregex. In our context, a Tregex pattern matches a specific structure of the constituency structure of a requirement. The constituency structures of the requirements are obtained in Step 1 (Section 5.1).

To create Tregex patterns, we analyzed the syntax of 1404 requirements that are identical to the requirements used in Section 4.1. The process to derive the patterns is as follows: (1) We group the requirements that have the same segments. A segment could be a scope, condition, and system response. (2) We analyze the constituency structures of the segments of each group. (3) We derive Tregex patterns that match the constituency structure of the segments of each group. (4) We analyze all patterns matching the same segment across all groups, attempting to refine and merge them, if possible. If not, a segment may have more than one pattern.

Table 6 presents the 11 Tregex patterns that we derived after following the process described above. In total, we derived two patterns that detect scope (see SC1 and SC2 in Table 6), eight patterns that detect conditions (C1–C8), and one pattern that detects system response (SR1). In the table, for each Tregex pattern, there is an example requirement containing the segment matched by the pattern. The segment is emphasized in italic.

Figure 4 shows an example of the usage of the pattern C3 (see Table 6) to extract the condition of requirement R from the corresponding constituency parsing tree. 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"). We note that some concepts of requirement R are anonymized to comply with the confidentiality agreement with our industrial partner. The condition is composed of a subordinate conjunction (WHADVP: "when"), a noun phrase (NP: "System-A"), and a verb phrase (VP: "receives an email alert from System-B"). As shown in Figure 4 (highlighted in the colored boxes), the pattern C3 identifies a subordinate clause (SBAR) that is the parent (<) of a Wh-adverb phrase (WHADVP), which is the immediate left sister ($+) of a clause (S). Clause (S) is the parent (<) of a noun phrase (NP), which is the left sister ($++) of a verb phrase (VP).

Segment splitter. We propose a segment splitter that attempts to separate requirements into segments (e.g., scope,

condition, and system response). The segment splitter is only used when all the Tregex patterns fail to identify segments in a requirement. We note that such failures can be caused by a malformed constituency parsing tree that incorrectly identifies the structural units of a requirement or a requirement structure that was not observed during the creation of the Tregex patterns.

To create our segment splitter, we first collect the keywords that characterize the beginning of each segment in a requirement. These keywords include (1) for the condition: "when", "if", "where", "while", and "until", (2) for the scope: "for", and (3) for the system response: "then", (line break), ";", "else", and "otherwise". We identified these keywords based on both the Rimay grammar and the requirements inspected when deriving our catalog of requirements smells (see Section 4.1). In general, such an analysis method that relies on keywords has limitations in terms of exhaustiveness. When needed, however, extending our set of keywords is very straightforward.

Our segment splitter splits the requirement into segments by detecting the above keywords in the requirements and then validates each segment. A segment is considered valid if it has the mandatory information content (e.g., actor, modal verb, and verb in a system response). Table 7 shows the mandatory information content that each segment should have to be considered valid.

Finally, when segments do not comply with their mandatory information contents, Paska labels them as "Not Matched". These segments will be further analyzed to detect smells in Step 3 (Section 5.3).

In the example shown in Figure 3 (Step 2), we first apply all the Tregex patterns (Table 6) to the requirement. The pattern SR1 is matched, and segments 002 and 003 are identified as system responses. Paska then applies the segment splitter. However, the segment splitter fails to identify segment 001, since the word "Upon" is not present in our keyword list. Since this word describes a preposition, it can be anywhere in a requirement and not only at the beginning of a segment.

5.3 Step 3: Identify Smells

This section answers **RQ2: How can smells be detected?** Algorithm 1 presents our automated procedure for detecting any of the smells introduced in Section 4.1. As shown in line 1 of the algorithm, we analyze the requirement segments obtained from Step 2 in Figure 2. To detect smells in each segment, the procedure employs the following techniques: Tregex patterns, structural patterns, rules, and glossary search (see lines 3-9). Below, we describe in detail these techniques and the smells that each of them detects.

Requirement

Description

Upon reception of a settlement instruction from System-A, System-B must process the settlement instruction and the input media field must be set to "SINF" .

↓

Preprocess Requirement (Step 1)

Requirement

Description

Upon reception of a settlement instruction from System-A , System-B must process the settlement instruction and the input media field must be set to SINF .

↓

Separate Requirement into Segments (Step 2)

Requirement

#	Description	Segment	Method
001	Upon reception of a settlement instruction from System-A ,	Not identified	
002	System-B must process the settlement instruction and	System Response	Tregex pattern
003	the input media field must be set to SINF .	System Response	Tregex pattern

↓

Identify Smells (Step 3)

Requirement

#	Description	Segment
001	Upon reception of a settlement instruction from System-A ,	Not matched
002	System-B must process the settlement instruction and	System Response
003	the input media field must be set to SINF .	System Response

→

Suggest Rimay Pattern (Step 4)

Requirement

#	Description	Segment
001	Upon reception of a settlement instruction from System-A ,	Incomplete Condition
002	System-B must process the settlement instruction and	System Response
003	the input media field must be set to SINF .	System Response

Rimay Pattern: 7. Condition (Trigger) and system response

Identify Smells (Step 3)

a) Apply Tregex patterns IC1 and IC2 to segments "Not matched"

#	Description	Segment	Smell
001	Upon reception of a settlement instruction from System-A ,	Not identified	Incomplete Condition

Result: Tregex pattern IC2 (Condition no actor no verb) matches Segment #001

b) Apply structural patterns to the remaining segments

#	Description	Segment	Smell
002	System-B must process the settlement instruction and	System Response	
003	the input media field must be set to SINF .	System Response	Passive Voice

Result: Structural pattern #1 (Passive voice) matches Segment #003

c) Apply rules to the requirement

#	Description	Segment	Smell
001	Upon reception of a settlement instruction from System-A ,	Not identified	
002	System-B must process the settlement instruction and	System Response	Non Atomic
003	the input media field must be set to SINF .	System Response	

Result: Requirement has two system responses which triggers the rule non atomic

d) Search for not precise verbs in system response segments (Glossary search)

#	Description	Segment	Smell
002	System-B must process the settlement instruction and	System Response	Not Precise Verb
003	the input media field must be set to SINF .	System Response	

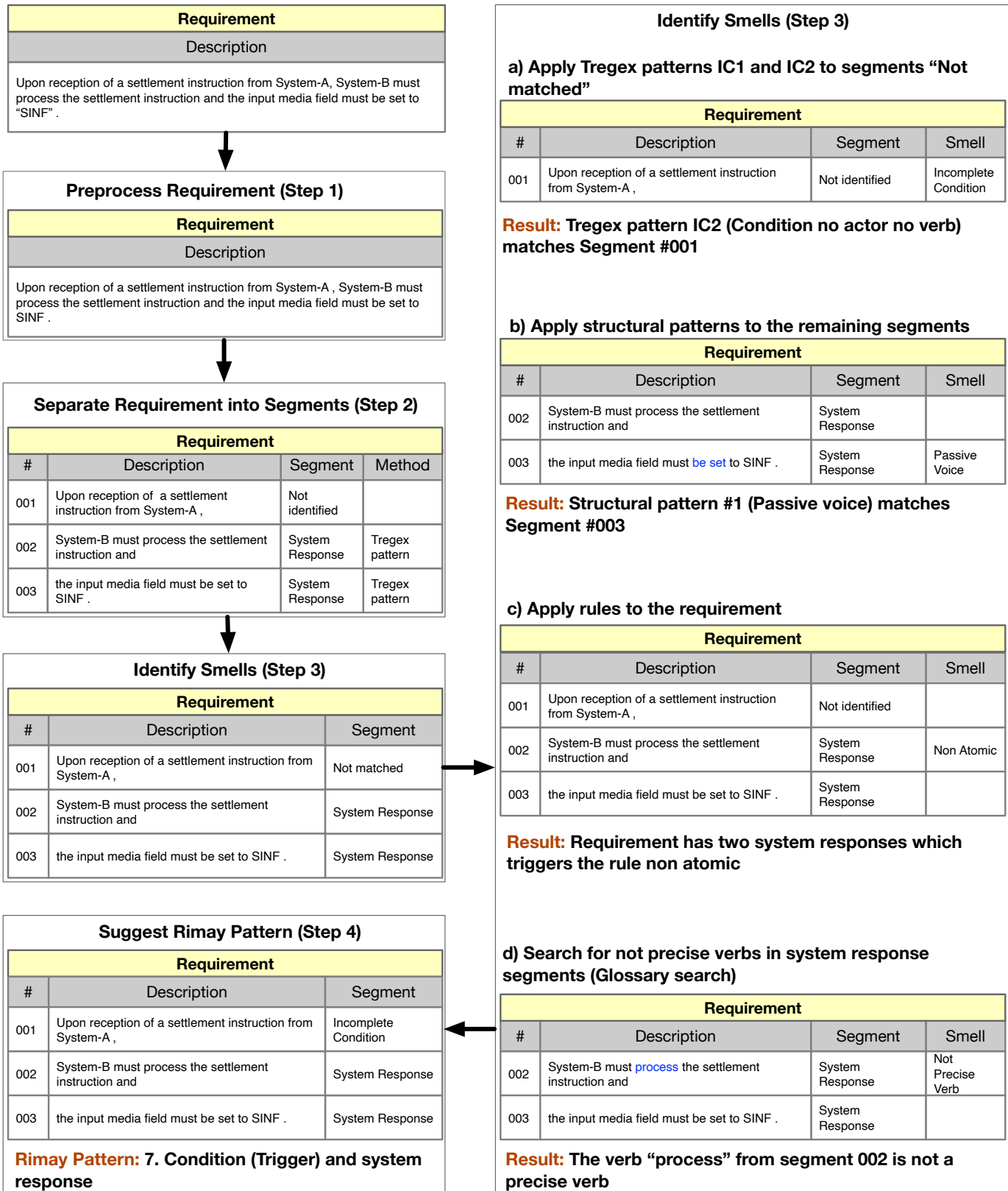**Result:** The verb "process" from segment 002 is not a precise verb

Fig. 3: Examples illustrating smell detection and Rimay pattern suggestions.

Tregex patterns. Recall from Section 5.2 that the Tregex patterns match the specific structures of the constituency parsing tree resulting from an NL requirement. While analyzing the set of requirements in Section 5.2, we observed two groups of requirements that contain incomplete conditions.

The following are two examples of these requirements: EIC1 and EIC2. *"EIC1: Upon reception from System-A the*

TABLE 6: Tregex patterns to identify segments in requirements. The examples in this table are derived from the actual requirements used in our case study but are anonymized for confidentiality.

ID	Segment	Tregex Pattern	Example
SC1	Scope	`(PP < ((IN < For) $+ NP)) > (S < (/(RB | , | ADVP)/ $+ (NP $+ VP) ))`	*For all the depositories, System-A must create a T30 transaction processing command.*
SC2	Scope	`( (PP < ((IN < For) $+ NP)) [ >- ((VP < MD) $- NP) ] & !>> PP)`	*System-A must create an instruction with the the remote code value "B" for each settlement request.*
C1	Condition	`WHADVP $+ (S <, (S < (NP $+ VP)))) > SBAR & !>> VP`	*When System-A creates one of the following reports: list, list with Beta, System-A must populate the field "A" in the report.*
C2	Condition	`((IN < once) $+ (S < (NP $+ VP))) [> SBAR	> S] & > !VP`	*Once System-A has successfully validated a settlement request, System-A must send an acknowledge message to System-B.*
C3	Condition	`SBAR < ( WHADVP $+ S < (NP $++ VP))`	*System-A must send a settlement request to System-B when the contract note has been received from System-C.*
C4	Condition	`(WHADVP $+ (S < (NP $+ VP)) !>> /(VP | SBAR)/)`	*When the fund frequency in reference data has an empty value, then System-A must set the fund frequency to "daily".*
C5	Condition	`(WHADVP !< /(of | to)/) $+ (NP $+ VP) !>> /(VP | SBAR)/`	*When the user clicks on the left side menu, portfolio section, System-A must display the portfolios.*
C6	Condition	`(SBAR < ((WHADVP !<< that) $+ (S << (SBAR <<, /(That | that)/) & <<, (NP $++ VP | $++ VB)))) !>> VP`	*When a user confirms that he wants to cancel the creation of an account record, the System-A must delete the related parameters recorded in account with status "draft".*
C7	Condition	`(SBAR < ((WHADVP !<< that) $+ (S !<< SBAR &<<, (NP $++ VP | $++ VB)))) !>> VP`	*If the settlement date is present in the instruction sent to System-A then System-A must store in data storage unit.*
C8	Condition Time	`(PP < (IN < (/(after | before)$/) $+ (NP !< VB < NN) )) > S`	*Before the System-A cutover, System-A must update the entity-A data model.*
SR1	System Response	`NP $+ (VP < ( MD ?$+ ADVP $++ (VP <<, (/VB.?/ $+ (S < (NP $++ VP)))))) > S`	*System-A must send the fund details report to local team daily.*

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 7: Information contents characterizing the requirement segments

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

Algorithm 1 Detect Smells in NL Requirement.

Input:
 r: NL requirement
Output:
    D: detected requirement smells

1: $SEG \leftarrow$ SEPARATESEGMENTS(r) // Step 2
2: $D \leftarrow \{\}$
3: **for each** $seg \in SEG$ **do**
4:     $D_t \leftarrow$ APPLYTREGEXPATTERN(seg)
5:     $D_s \leftarrow$ APPLYSTRUCTURALPATTERN(seg)
6:     $D_r \leftarrow$ APPLYRULES(seg)
7:     $D_g \leftarrow$ SEARCHGLOSSARY(seg)
8:     $D \leftarrow D \cup D_t \cup D_s \cup D_r \cup D_g$
9: **end for**
10: **return** D

status Pending of an Instruction, then ...". The condition of EIC1 misses the actor and the verb. Instead of a verb, the condition contains the noun "reception", making it unclear who or what is doing the receiving. *"EIC2: When creating a new participant, System-A must...".* The condition of EIC2 lacks the actor, and the verb is described using a gerund. Missing an actor or a verb in a condition statement can result

in ambiguity and incompleteness issues in a functional requirement. To detect such conditions, we derived two Tregex patterns. To do so, we found a set of 55 requirements from the 1404 requirements (i.e., S^D in Section 4.1) that are similar to EIC1 and EIC2. Next, we grouped the requirements into two sets. Each set shares the same information content. Then, for each set, we derived a Tregex pattern. Table 8 shows the two derived patterns, i.e., IC1 and IC2, to detect incomplete conditions.

In the example shown in Figure 3 (Step 3a), Paska matches the pattern IC2 with segment 001 "Upon reception of a settlement instruction from System-A". The condition lacks the actor and the verb; therefore, the smell "Incomplete condition" is detected.

Structural patterns. This method analyzes the segments of a requirement using structural patterns to detect the following smells: "Incomplete condition", "Incomplete system response (SR)", and "Passive voice". Regarding the smell "Incomplete condition", recall that our Tregex patterns aim at detecting it. However, due to various reasons, in practice, the constituency parsing tree of a requirement may be malformed. In such cases, our structural patterns are applied to detect the smell. A structural pattern checks the presence of certain words describing the concepts (see Figure 1) of segments in a specific sequence.

Tregex pattern to extract conditions:
SBAR < (WHADVP $+ (S < (NP $++ VP)))
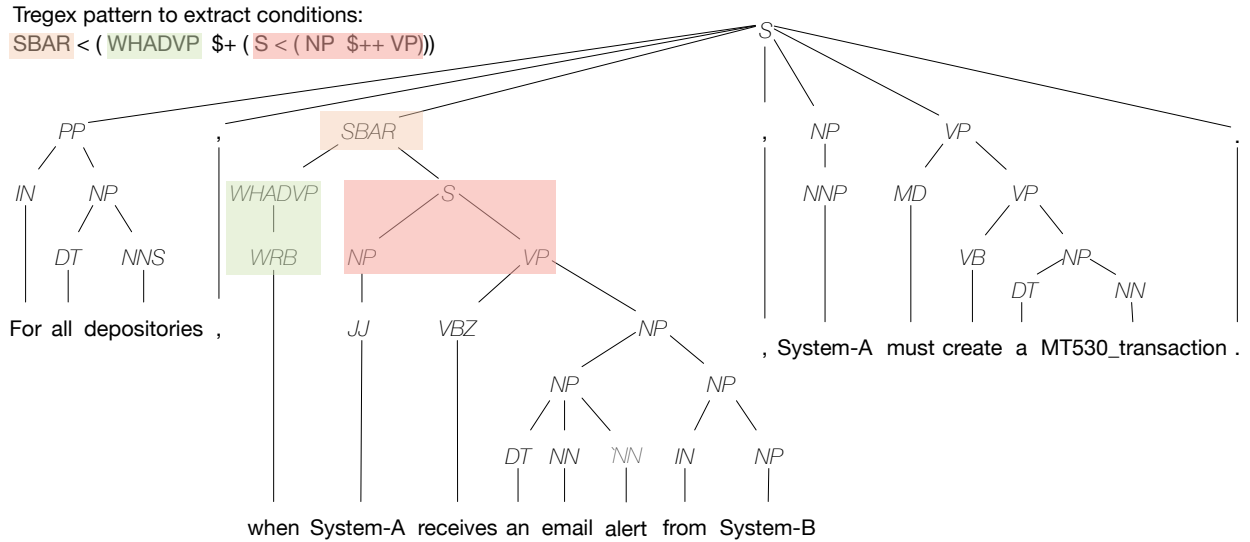
R: For all depositories, when System-A receives an email alert from System-B, System-A must create a MT530-transaction.

Fig. 4: An example Tregex pattern that matches a constituency parsing tree of a condition.

TABLE 8: Tregex pattern to detect incomplete conditions

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

As shown in Table 9, we define 14 structural patterns to detect the above smells. We analyzed the 1404 requirements by categorizing them into different smell groups. For each group, we inspected the requirements at the segment level (e.g., condition and system response) based on Rimay's concepts and constructs and identified specific structural patterns to detect the 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. In addition, we define three patterns that check for the smell "Incomplete condition", and three patterns to detect the smell "Incomplete system response".

We analyze the segments obtained from Step 2 of Figure 2 using the structural patterns. For example, the structural pattern "Passive voice #7" matches the condition of the following requirement "When/WRB an/DT Order/NNS has/VBZ been/VBN assigned/VBN via/IN propagation/NNP ..". The condition contains verb "has"(v_1) in the present tense, followed by verb "been"(v_3) in the past participle, and followed by verb "assigned"(v_4) in the past participle.

In the example shown in Figure 3 (Step 3b), Paska applies our structural patterns to the segments of the requirement. Specifically, the structural pattern "Passive Voice 1" matches segment 003. This segment has the verb "be"(v_1) in its base form and the verb "set"(v_4) in its past participle form.

Rules. We propose a set of rules to analyze the segments of a requirement identified by Step 2 of Figure 2, aiming at detecting the following smells: "Non-atomic", "Incomplete requirement", "Incorrect order requirement", "Coordination ambiguity", and "Not a requirement". Our rules analyze how the segments are connected to each other and determine the sequence of the segments of the requirement. In the following, we describe these rules:

1) **Non-atomic:** The requirement has more than one segment for 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 condition segments after its system response.
4) **Coordination ambiguity:** When the requirement has two or more subsequent conditions, Paska extracts the word(s) or character(s) that separate the conditions. If the separator(s) is the conjunction *"or"*, then Paska triggers the smell "Coordination ambiguity".
5) **Not a requirement:** All segments of the requirement are neither scope, condition, nor system response.

In the example depicted in Figure 3 (Step 3c), Paska applies our rules to the requirement. The rule "Non-atomic" identifies two system responses in the requirement that triggers the smell "Non-atomic".

Glossary search. This method aims to identify the smell: "Not a precise verb". For this purpose, we create a glossary of verbs that do not describe precise actions, are difficult to test, and carry several meanings. Note that these criteria are used to define vague words in the literature [35]. For example, according to the English dictionary, the verb "process"

TABLE 9: Structural patterns for smell detection

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..."

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

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 gather all the verbs of the requirements used in Section 4.1. We search for verbs that do not have precise actions and are 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 glossary also includes verbs that have several meanings but only one etymology (polysemy). These verbs in our glossary are: come and get.

We have elaborated our glossary in collaboration with two experts working for our industrial partner. The experts agreed that they 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, Paska automatically extracts verbs from the requirement segments condition and system response. Next, Paska obtains the lemmas of these verbs. Finally, Paska searches for the lemma in our glossary. If there is a match, Paska triggers the smell "Not a precise verb".

In the example shown in Figure 3 (Step 3d), Paska analyzes the segments 002 and 003 because they are system responses; then, Paska extracts the verb "process" because it belongs to our glossary. Hence, the smell "Not a precise verb" is detected.

> *The answer to RQ2 is that* Paska detects smells in NL requirements using several complementary techniques: Tregex patterns, structural patterns, rules, and glossary search. Paska combines these techniques to ensure that various requirement smells are accurately identified since no single technique is comprehensive enough to detect all of them.

5.4 Step 4. Suggesting Rimay Patterns

This section answers **RQ3: How can we suggest recommendations to improve requirement quality?** To suggest Rimay

---

Algorithm 2 Suggest Rimay Pattern.

Input:
  r: NL requirement
Output:
  p: suggested Rimay pattern

1: $SEG \leftarrow$ SEPARATESEGMENTS(r) // Step 2
2: // Count segment frequencies
3: $cnt_s \leftarrow$ COUNTSCOPES(SEG)
4: $cnt_c \leftarrow$ COUNTCONDITIONS(SEG)
5: $cnt_r \leftarrow$ COUNTSYSTEMRESPONSES(SEG)
6: // Identify condition type
7: $ctype \leftarrow$ IDENTIFYCONDITIONTYPE(SEG)
8: // Match Rimay pattern
9: $p \leftarrow$ MATCHRIMAYPATTERN(cnt_s, cnt_c, cnt_r, $ctype$)
10: **return** p

---

patterns as recommendations, Paska analyzes the segments of a requirement identified by Step 2 to match one of the 10 Rimay patterns (Section 4.2). The matched pattern will guide analysts when fixing any smell detected in a requirement and converting it into a Rimay requirement.

Algorithm 2 shows Paska's automated procedure for suggesting a Rimay pattern for a given NL requirement. To identify a suitable Rimay pattern, Paska first counts the frequencies of the segments in a requirement, since the Rimay patterns are defined in terms of segments (see lines 1-5). More concretely, Paska counts the number of segments— scopes, conditions, and system responses—that appear in a requirement. For the condition segments, Paska further classifies them into trigger, time, and precondition (Section 2) (see lines 6-7). To identify "time condition", Paska checks if a segment is matched by the pattern "C8" (Table 6) in Step 2. To identify "precondition condition", Paska extracts the verb phrase (VP) from a condition segment. If the VP contains one of the operators [21], i.e., "is equal to", "less or equal to", "contain", and "have", then the condition segment is identified as "precondition condition". Moreover, to identify "trigger condition", we extract the VP from a condition seg-
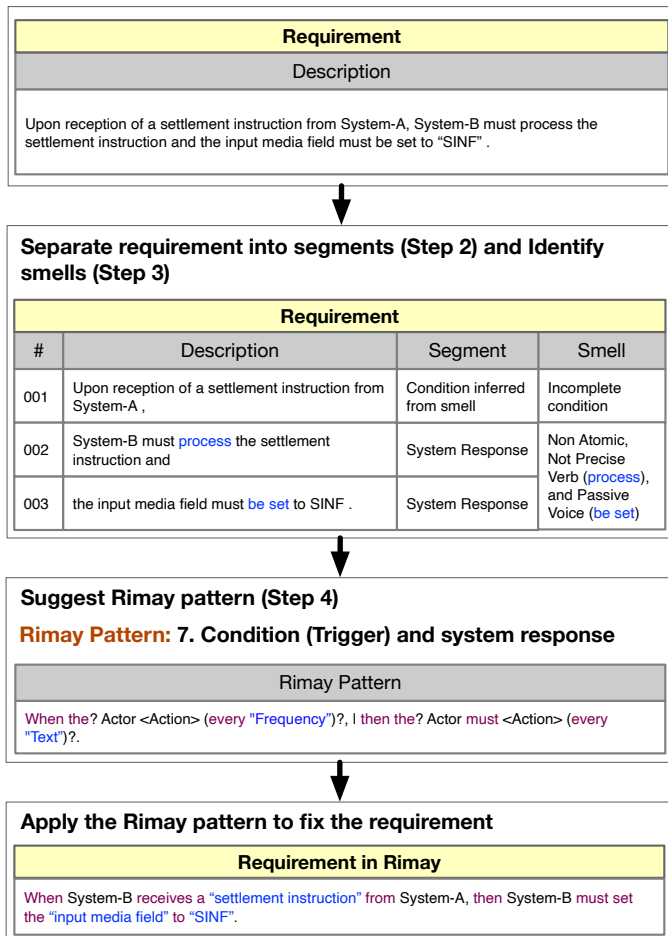
Requirement

Description

Upon reception of a settlement instruction from System-A, System-B must process the settlement instruction and the input media field must be set to "SINF" .

Separate requirement into segments (Step 2) and Identify smells (Step 3)

Requirement

#	Description	Segment	Smell
001	Upon reception of a settlement instruction from System-A ,	Condition inferred from smell	Incomplete condition
002	System-B must process the settlement instruction and	System Response	Non Atomic, Not Precise Verb (process), and Passive Voice (be set)
003	the input media field must be set to SINF .	System Response	

Suggest Rimay pattern (Step 4)

Rimay Pattern: 7. Condition (Trigger) and system response

Rimay Pattern

When the? Actor <Action> (every "Frequency")?, | then the? Actor must <Action> (every "Text")?.

Apply the Rimay pattern to fix the requirement

Requirement in Rimay

When System-B receives a "settlement instruction" from System-A, then System-B must set the "input media field" to "SINF".

Fig. 5: Application of a Rimay pattern.

ment. If the VP contains verbs other than the ones used by the "precondition condition", then the segment is classified as "trigger condition". Paska also counts the frequencies of incomplete segments with the following smells: "incomplete condition" and "incomplete system response". Once the frequencies are calculated, Paska maps the frequencies to any of the 10 Rimay patterns (see lines 8-9). Table 10 presents the frequency of segments for each of the 10 Rimay patterns. The first column shows the name of the pattern. From the second to the sixth column, we show the frequency of each segment contained in each of the 10 Rimay patterns.

In the example shown in Figure 3 (Step 4), Paska analyzes the three segments 001, 002, and 003 in the requirement. Segment 001 is an incomplete condition as Paska does not detect any verb in the segment. However, the Tregex pattern (i.e., IC2 in Table 8) identifies the smell "Incomplete condition", indicating that the condition uses the noun "reception" instead of a verb. This analysis result suggests that the verb is an action verb, indicating that it is a condition trigger. To summarize, the requirement in Figure 3 (Step 4) has a condition trigger and two system responses. Since Rimay discourages analysts from writing non-atomic requirements, Paska suggests Rimay pattern "7. Condition(Trigger) and system response" to help analysts rewrite the requirement.

Figure 5 illustrates how an analyst applies the suggested Rimay pattern to address smells in an

NL requirement. Given the suggested Rimay pattern "7. Condition(Trigger) and system response" (see Table 5), the analyst rewrites the NL requirement in Rimay to address the following smells: "incomplete condition", "non-atomic requirement", "not precise verb", and "passive voice". The Rimay requirement includes the trigger condition: `When System-B receives a "settlement instruction" from System-A`, addressing the "incomplete condition" smell, and the system response: `System-B must set the "input media field" to "SINF"`, addressing the "non-atomic requirement" and "passive voice" smells. In this example, we note that the analyst identified segment 002 in the NL requirement as unnecessary and consequently removed it.

The answer to RQ3 is that Paska suggests recommendations in the form of Rimay patterns by analyzing segments (i.e., scope, condition, and system response) in an NL requirement. The recommended Rimay patterns are meant to guide analysts when fixing identified smells in an NL requirement and converting it into a Rimay requirement. In summary, Paska indicates segments in the requirements that may be missing, are incorrectly ordered, or that require change.

6 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 [36].

6.1 Case Study Design

Our evaluation aims to answer the following RQs:

RQ4. Can Paska accurately indicate the occurrence of smells?

RQ5. How accurate is Paska in recommending requirement patterns to fix smells?

To answer RQ4 and RQ5, we measured the accuracy of Paska in detecting smells and suggesting Rimay patterns, using an annotated dataset as a ground truth for comparison. To construct our ground truth GT, we annotated the 13 SRSs (described in Section 3.2) provided by our industrial partner. To annotate the SRSs, we had four annotators who manually analyzed the syntax and semantics of each requirement to detect smells and suggest a Rimay pattern. Sections 6.2 and 6.3 provide further details about our annotation process and results.

Once the ground truth was completed, we developed Paska using the development set S^D of requirements (i.e., SRS1-SRS6), as described in Section 6.5. We then evaluated the accuracy of Paska in detecting smells and suggesting Rimay patterns. To this end, we compared the results obtained by Paska using the evaluation set S^E (i.e., SRS7-SRS13) against the ground truth GT. We thus calculated the accuracy of smell detection and pattern recommendation using S^E, which we report in Section 6.6.

6.2 Data Annotation

To annotate the 13 SRSs described in Section 3.2, we hired and trained three external annotators to identify smells in

TABLE 10: Segment frequencies in Rimay patterns.

Pattern	Scope	Condition			System Response
		Pre-condition	Trigger	Time	
1. Scope and system response	1	0	0	0	1
2. Scope, condition (precondition), and system response	1	1	0	0	1
3. Scope, condition (trigger), and system response	1	0	1	0	1
4. Scope, condition (time) and system response	1	0	0	1	1
5. System response	0	0	0	0	1
6. Condition(precondition) and system response	0	1	0	0	1
7. Condition (trigger) and system response	0	0	1	0	1
8. Condition (time) and system response	0	0	0	1	1
9. Scope, multiple conditions, and system response	1	2 or more			1
10. Multiple conditions and system response	0	2 or more			1

SRSs and recommend Rimay patterns to fix such smells. Two out of the three external annotators have more than two years of experience in requirements engineering. The other external annotator has knowledge about the domain of our industrial partner and also has more than two years of experience in requirements engineering. To train the annotators, we randomly selected and used 70 requirements from S^D. During the training, we applied Cohen's kappa [37] to measure inter-annotator agreement and obtained a score of 0.89, indicating strong agreement.

After the training, the three external annotators and the first author of this article conducted a collective 312-hour annotation process. We note that the annotations made by the first author of this article are included in the subset S^D and hence were used only to develop Paska. We assigned the SRSs to the annotators accounting for their individual situations and contracts, which resulted in different numbers of SRSs being assigned to each annotators. The annotators manually analyzed the syntax and semantics of each requirement in the SRSs of S^D and S^E to detect smells and assign a Rimay pattern. After the annotators completed annotating each SRS, we analyzed the annotation results by having an in-person monitoring session (30-60 minutes). In each session, the annotators pointed out 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, indicating agreement among the annotators. For the unsatisfactory annotations (less than 20%), we then identified their causes and asked the annotators to correct them throughout all SRSs in S. We finally accepted the annotations.

6.3 Annotated Requirements

Table 11 shows the annotation results of the smells detected in S. The first column indicates the smell name. The second through seventh columns present the number of requirements containing the smell listed in each row for each batch in S. The last column displays the total number of occurrences of each of the nine smells in S. Table 11 confirms that the SRSs used in our study are highly diverse in terms of smells since there is significant variation across batches in terms of the distribution of smells. Further, smells occur according to very different frequencies. We found that "8. Passive Voice" is the smell with the highest frequency

in S (accounting for 40.9% of the requirements) while, in contrast, we have smells such as "5. Not a requirement", with very low frequencies (0.28% of the requirements).

Table 12 shows the frequencies of Rimay patterns assigned by the annotators to requirements in S. The first column shows the Rimay pattern. The second through seventh columns present the number of requirements assigned to each Rimay pattern for each batch in S. The last column displays the total number of occurrences of each of the 10 Rimay patterns in S. Table 12 suggests that, similar to smells, the SRSs used in our case study have significant diversity in their structural composition and patterns occur according to widely different frequencies. For example, "10. Multiple conditions and system response" is the most frequently assigned Rimay pattern with 803 occurrences in set S. The least frequently assigned Rimay patterns (less than 20 times) are "2. Scope, condition (precondition), and system response" and "8. Condition (Time) and system response". In Table 12, we found that the pattern "4. Scope, condition (Time) and system response" is not observed in S. Recall from Section 4.2 that Rimay patterns represent valid sequences of Rimay concepts used to write requirements in Rimay. Even though the SRSs do not have requirements that can be rewritten by applying this pattern, we opted to keep the pattern in Paska to provide the complete list of Rimay patterns.

6.4 Metrics

Our analysis of the accuracy of Paska, when detecting smells and suggesting Rimay patterns, is based on the precision and recall metrics. We applied Paska to the subsets of requirements S_D and S_E to detect smells and suggest Rimay patterns. We compared these results against the ground truth GT (Section 6.1) using precision and recall. For each smell s and Rimay pattern p, we first classified Paska's predictions for a requirement r into the following categories:
True positives (TP) are the correct predictions. In smell detection, a TP occurs when Paska detects the same smell s as the ground truth GT for requirement r. In pattern suggestion, a TP occurs when Paska assigns the same Rimay pattern p as GT to requirement r.
False negatives (FN) are missed annotations. In smell detection, a missed annotation occurs when Paska fails to detect the smell s that is annotated for requirement r in the ground truth GT. In pattern suggestion, a missed annotation occurs when Paska fails to suggest the pattern p that is annotated for r in GT.

TABLE 11: Smells - Annotation results for set S

Smell	Batch 1	Batch 2	Batch 3	Batch 4	Batch 5	Batch 6	Total
1. Non-atomic	14	79	107	6	23	78	310
2. Incomplete requirement	2	8	1	1	1	2	15
3. Incorrect order requirement	21	40	34	9	7	14	125
4. Coordination ambiguity	0	4	0	1	5	4	14
5. Not a requirement	2	0	0	1	1	1	5
6. Incomplete condition	18	72	70	9	49	104	322
7. Incomplete system response	4	9	0	2	1	3	19
8. Passive voice	174	219	59	33	95	141	721
9. Not precise verb	2	111	27	26	33	32	231

TABLE 12: Rimay patterns - Annotation results for set S

Rimay Pattern	Batch 1	Batch 2	Batch 3	Batch 4	Batch 5	Batch 6	Total
1. Scope and system response	6	16	7	3	1	7	40
2. Scope, condition (precondition), and system response	9	3	0	0	0	0	12
3. Scope, condition (trigger), and system response	13	28	113	0	0	83	237
4. Scope, condition (time), and system response	0	0	0	0	0	0	0
5. System response	91	45	40	203	111	37	527
6. Condition (precondition) and system response	75	13	5	21	35	7	156
7. Condition (trigger) and system response	116	158	73	202	105	104	758
8. Condition (time) and system response	0	2	4	0	0	0	6
9. Scope, multiple conditions, and system response	5	29	86	0	4	39	163
10. Multiple conditions and system response	63	310	79	25	120	206	803

TABLE 13: Batch distribution of the SRSs in S

Subset	Batch #	SRS ID	# Requirements
S^D	1	SRS1	192
		SRS2	188
	2	SRS3	118
		SRS4	161
	3	SRS5	451
		SRS6	294
S^E	4	SRS7	367
		SRS8	90
	5	SRS9	167
		SRS10	192
		SRS11	19
	6	SRS12	340
		SRS13	146

False positives (FP) are misclassified annotations. In smell detection, an FP occurs when Paska incorrectly indicates the presence of the smell s that is not annotated for requirement r in the ground truth GT. In pattern suggestion, an FP occurs when Paska incorrectly suggests the Rimay pattern p that is not annotated for r in GT.

Next, for each detected smell (or assigned Rimay pattern), denoted by i, we calculated the precision (P_i) as $P_i = TP_i/(TP_i+FP_i)$ and the recall (R_i) as $R_i = TP_i/(TP_i+FN_i)$, where TP_i, FP_i, and FN_i denote, respectively, the number of true positives, false positives, and false negatives in predicting i.

Furthermore, we calculated the overall precision as `Overall-P` $= \sum_{i=1}^{l} TP_i / \sum_{i=1}^{l}(TP_i + FP_i)$, where $i, \ldots, l$ denote either the nine smells for smell detection or the 10 Rimay patterns for pattern suggestion. The overall recall was calculated as `Overall-R` $= \sum_{i=1}^{l} TP_i / \sum_{i=1}^{l}(TP_i + FN_i)$.

6.5 Paska Development

We employed an iterative process to develop Paska, as depicted in Figure 6. For this iterative development, we divided subset S^D into three batches as detailed in Table 13. Further, we also divided S^E into three batches to prepare for the possibility that Paska would not become stable after using S^D, something difficult to predict at the development stage. However, it turned out that Paska was stable after using S^D and therefore no batches from S^E were needed to reach a stable version of Paska.

As shown in Figure 6, to build a stable version of Paska, we followed four steps: **(Step 1)** We applied Paska to a first batch of requirements listed in Table 13 to detect smells and suggest Rimay patterns. **(Step 2)** We compared our results against the ground truth by computing precision and recall. **(Step 3)** We evaluated Paska's stability by applying the concept of saturation, which determines the point in the iterative development process of Paska where we have analyzed a sufficient number of SRSs to confidently identify smells and recommend Rimay patterns. In general, saturation refers to the point in a qualitative study when no new information emerges from the data being analyzed, i.e., when no new properties, dimensions, conditions, actions/interactions, or consequences are found in the data [33]. In our context, during the development of Paska, we determined that the saturation point was reached when we observed stability from one batch of SRSs to the next in terms of overall precision and recall when identifying smells and recommended patterns. When such precision and recall were significantly worse than in the previous batch, we analyzed cases showing disagreements between Paska and the ground truth in order to account for new situations. **(Step 4)** We then improved Paska by correcting any disagreements with the ground truth. We repeated Steps 1 to 4 using each time a new batch as listed in Table 13 until Paska became stable. For example, the results for batch 2
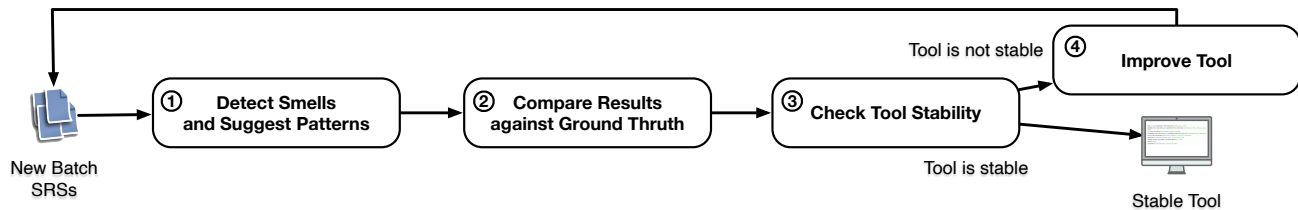
Fig. 6: Paska development overview

were initially much worse for smell detection than those of batch 1, whereas the results for batch 3 were right away comparable to those of batch 2 after improvement, thus suggesting no new situations needed to be investigated and accounted for. Below, we describe in detail how we improved Paska.

Table 14 shows precision (P) and recall (R) values obtained by Paska, after the iterative improvement mentioned above, for detecting smells in S^D. These results were calculated based on the ground truth GT. The first column of Table 14 lists the smell names. The P and R values for each of the nine smells found in the first batch are shown in the second and third columns, while the fourth and fifth columns show P and R values for the smells found in the second batch. The sixth and seventh columns provide the P and R values for each of the smells found in the third batch of S^D. The overall P and R values for all three batches of S^D are reported in the last two rows of Table 14.

Following the iterative development process depicted earlier, we first developed Paska using the first batch of S^D and compared our results with GT. In the first iteration, we obtained a precision (P) of 99% and a recall (R) of 99%. We then applied Paska to the second batch and obtained low P and R values (i.e., a P of 69% and an R of 58%), which were considered unsatisfactory. To address this, we analyzed the false positives (FP) and false negatives (FN) (described in Section 6.4). Most of the FPs and FNs cases resulted from new scenarios that were not considered during the analysis of Batch 1. These new scenarios refer to requirements that require further investigation to improve Paska. We improved Paska to support these new scenarios and detect smells in the second batch, resulting in significant improvements with a precision of 92% and a recall of 94%. Further, we applied Paska to the third batch and obtained P of 90% and an R of 88%, which were deemed acceptable. These high precision and recall scores indicate that Paska detects most relevant smells in requirements with low probabilities of false positives and false negatives. At that point, we then stopped analyzing more SRSs and improving Paska as we reached saturation in terms of accuracy.

The precision (P) and recall (R) scores obtained by the stable version of Paska on S^D when suggesting appropriate Rimay patterns to analysts, are presented in Table 15. P and R values are reported for each Rimay pattern (column 1) suggested in the first, second, and third batches (columns 2-7), as well as the overall P and R scores for Batches 1, 2, and 3 in S^D shown in the last row of the table.

Similarly to what we did for smell detection, we first developed Paska using the first batch of S^D to recommend suitable Rimay patterns and compared our results with GT.

The results were satisfactory. We obtained a precision (P) of 95% and a recall (R) of 92% in the first batch. Next, we applied Paska to the second and third batches, and the P and R values were around 90%. We therefore reached saturation after the first batch, and no further enhancement to Paska was necessary. In terms of the overall accuracy, the suggestions for Rimay patterns yield an overall P of 92% and an R of 90% (Table 15).

We note that Tables 14 and 15 show that in a few cases there is no data to evaluate certain smells (such as "5. Not a requirement" from Table 14) and to suggest certain Rimay patterns (such as "4. Scope, condition (time), and system response" from Table 15). This is indicated as N/A in the tables when the denominators of the precision and recall metrics are zero. We further discuss this issue in Section 7.

Paska implementation and availability. We implemented Paska using Python and Java. Specifically, we utilized spaCy [38], Stanford CoreNLP [39], and AllenNLP [34] to implement the preprocessing steps (Section 5.1), including tokenization, post-tagging, and constituency parsing, respectively. Further, we employed the Stanford Tregex API for Java [23] to implement the Tregex patterns (Section 5.2). Paska is available online [40].

6.6 Analysis and Results

This section assesses the accuracy of Paska in detecting smells in NL requirements (RQ4) and suggesting Rimay patterns to analysts (RQ5). We applied Paska to the evaluation set S^E of requirements to detect smells and suggest Rimay patterns. Recall that S^E was not used when developing Paska. We compared these results against the ground truth GT (Section 6.1) using the precision and recall metrics (Section 6.4).

RQ4 results. Table 16 shows precision (P) and recall (R) values when detecting smells (RQ4). The first column of Table 16 shows the smell name. The second and third columns show the P and R values for each of the nine smells found in the fourth batch. The fourth and fifth columns show the P and R values for each of the nine smells found in the fifth batch. The sixth and seventh columns show the P and R values for each of the nine smells found in the sixth batch of S^E. The last row of Table 16 shows the overall P and R values for batches 4, 5, and 6 of S^E.

As shown in Table 16, we achieved a precision (P) of 87% and a recall (R) of 90% for the fourth batch, a P of 88% and an R of 84% for the fifth batch, and a P of 91% and an R of 91% for the sixth batch. Overall, the detection of smells yields $P = R = 89\%$.

TABLE 14: Smell detection results for set S^D (development) - precision and recall. The set S^D contains batches 1, 2, and 3 listed in Table 13.

Smell	Batch 1		Batch 2		Batch 3		All (S^D)	
	P	R	P	R	P	R	P	R
1. Non-atomic	1.00	1.00	0.92	0.92	0.94	0.94	0.94	0.94
2. Incomplete requirement	1.00	1.00	0.89	1.00	1.00	1.00	0.92	1.00
3. Incorrect order requirement	0.95	0.95	0.85	0.82	0.72	0.76	0.82	0.83
4. Coordination ambiguity	N/A	N/A	1.00	1.00	N/A	N/A	1.00	1.00
5. Not a requirement	1.00	1.00	N/A	N/A	N/A	N/A	1.00	1.00
6. Incomplete condition	1.00	0.94	0.75	0.93	0.90	0.77	0.83	0.86
7. Incomplete system response	1.00	1.00	1.00	0.89	N/A	N/A	1.00	0.91
8. Passive voice	1.00	0.99	0.98	0.95	0.93	0.90	0.98	0.96
9. Not a precise verb	1.00	1.00	0.96	0.97	0.87	0.96	0.94	0.97
Overall	0.99	0.99	0.92	0.94	0.90	0.88	0.93	0.93

TABLE 15: Rimay pattern suggestion results for set S^D (development) - precision and recall. The set S^D contains batches 1, 2, and 3 listed in Table 13.

Rimay Pattern	Batch 1		Batch 2		Batch 3		All (S^D)	
	P	R	P	R	P	R	P	R
1. Scope and system response	1.00	1.00	1.00	0.88	1.00	1.00	1.00	0.93
2. Scope, condition (precondition), and system response	0.80	0.89	1.00	1.00	N/A	N/A	0.79	0.92
3. Scope, condition (trigger), and system response	1.00	0.85	0.90	1.00	0.99	0.97	0.97	0.97
4. Scope, condition (time) and system response	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5. System response	1.00	0.97	0.95	0.93	0.98	1.00	0.98	0.97
6. Condition (precondition) and system response	0.95	0.92	0.80	0.92	1.00	1.00	0.92	0.92
7. Condition (trigger) and system response	0.92	0.90	0.81	0.89	0.86	0.96	0.85	0.91
8. Condition (time) and system response	N/A	N/A	1.00	1.00	1.00	0.75	1.00	0.83
9. Scope, multiple conditions, and system response	0.83	1.00	1.00	0.93	0.89	0.84	0.91	0.87
10. Multiple conditions and system response	0.95	0.89	0.96	0.87	0.83	0.78	0.94	0.86
Overall	0.95	0.92	0.91	0.89	0.91	0.91	0.92	0.90

TABLE 16: Smell detection results for set S^E (evaluation) - precision and recall. The evaluation set S^E contains batches 4, 5, and 6 listed in Table 13.

Smell	Batch 4		Batch 5		Batch 6		All (S^E)	
	P	R	P	R	P	R	P	R
1. Non-atomic	0.71	0.83	1.00	0.83	0.88	0.92	0.89	0.90
2. Incomplete requirement	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3. Incorrect order requirement	0.90	1.00	0.88	1.00	0.74	1.00	0.81	1.00
4. Coordination ambiguity	1.00	1.00	0.80	0.80	0.80	1.00	0.82	0.90
5. Not a requirement	1.00	1.00	0.50	1.00	1.00	1.00	0.75	1.00
6. Incomplete condition	0.90	1.00	0.75	0.67	0.85	0.87	0.82	0.81
7. Incomplete system response	1.00	1.00	0.33	1.00	0.67	1.00	0.62	1.00
8. Passive voice	0.80	0.85	0.91	0.91	0.99	0.91	0.94	0.90
9. Not a precise verb	0.96	0.88	0.97	0.85	1.00	1.00	0.98	0.91
Overall	0.87	0.90	0.88	0.84	0.91	0.91	0.89	0.89

The answer to RQ4 is that, on a large number of real requirements from the financial domain, Paska shows a high degree of accuracy in detecting smells in NL requirements, with an overall precision and recall of 89%. Therefore, Paska detects most smells in NL requirements with a small number of false positives and false negatives.

From the data in Table 16, we observe that there are low precision (P) scores in the Batch 5 (i.e., "5. Not a requirement" with P = 50% and "7. Incomplete system response" with P = 33%) and Batch 6 (i.e., "7. Incomplete system response" with P = 67%). Furthermore, we observe a low recall (R) value in Batch 5 (i.e., "6. Incomplete condi-

tion" with an R of 67%). These are mainly caused by new scenarios that were not observed in the SRSs (S^D) used when developing Paska and limitations of the NLP tools employed by Paska. We further discuss the reasons for such low precision and recall cases in Section 7.

RQ5 results. Table 17 shows the precision (P) and recall (R) scores obtained by Paska when suggesting suitable Rimay patterns to analysts (RQ5). Once again, the results were calculated by applying Paska to the evaluation set S^E and comparing the results to the ground truth GT by using the precision and recall metrics (Section 6.4). Table 17 shows, for all Rimay patterns (column 1), the P and R values for the patterns suggested in Batch 4 (columns 2-3), Batch 5 (columns 4-5), Batch 6 (columns 6-7), and all batches

TABLE 17: Rimay pattern suggestion results for set S^E (evaluation) - precision and recall. The evaluation set S^E contains batches 4, 5, and 6 listed in Table 13.

Rimay Pattern	Batch 4		Batch 5		Batch 6		All (S^E)	
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
1. Scope and system response	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2. Scope, condition (precondition), and system response	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
3. Scope, condition (trigger), and system response	N/A	N/A	N/A	N/A	0.98	1.00	0.98	1.00
4. Scope, condition (time) and system response	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5. System response	1.00	0.99	0.98	1.00	0.97	1.00	0.99	0.99
6. Condition (precondition) and system response	1.00	0.95	0.89	0.94	0.75	0.86	0.91	0.94
7. Condition (trigger) and system response	0.98	0.93	0.86	0.83	0.99	0.87	0.95	0.89
8. Condition (time) and system response	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
9. Scope, multiple conditions, and system response	N/A	N/A	0.80	1.00	0.94	0.79	0.92	0.81
10. Multiple conditions and system response	0.76	1.00	0.96	0.85	0.94	0.99	0.93	0.94
Overall	0.97	0.96	0.93	0.90	0.96	0.95	0.96	0.94

(columns 8-9). The last row of Table 17 shows the overall scores of P and R for batches 4, 5, and 6 in S^E. We obtained P = 97% and R = 96% in Batch 4, P = 93% and R = 90% in Batch 5, and P = 96% and R = 95% in Batch 6. In terms of overall accuracy, the suggestions for Rimay patterns yield an overall P = 96% and R = 94% (Table 17).

We note that Table 17 shows the results obtained by applying Paska to all 1321 requirements in S^E, including requirements with and without smells. Paska is indeed applicable for consistently rewriting requirements in Rimay, with or without smells. If we consider only the requirements with smells in S^E, we have a subset of 1165 such requirements for which we obtained P = 90% and R = 87%.

The answer to RQ5 is that Paska is accurate in suggesting Rimay patterns, achieving an overall precision of 96% and recall of 94%. Hence, most of the time, Paska provides analysts with appropriate Rimay patterns to fix smells in NL requirements, with relatively few false positives and false negatives.

As shown in Table 17, there are low precision (P) values in Batch 4 (i.e., "10. Multiple conditions and system response" with a P of 76%) and the Batch 6 (i.e., "6. Condition (precondition) and system response" with a P of 75%). These low values are mainly caused by new scenarios that were not observed in the SRSs S^D used when developing Paska. In Section 7, we further examine the factors that lead to low P values. In addition, we found that some Rimay patterns are not applicable to any of the requirements in the SRSs S^E, as indicated in Table 17 by N/A, e.g., "2. Scope, condition (precondition) and system response". Section 7 further discusses such situations.

Remark. In our evaluation results, we observed that Paska obtained high overall precision and recall scores: respectively, 89% and 89% for smell detection, and 96% and 94% for pattern suggestion. Even though we would ideally like to achieve 100% accuracy, our results are promising and demonstrate the potential of Paska to support requirements quality assurance.

7 DISCUSSION
7.1 Approach Performance.

The results of RQ4 presented in Section 6 show that Paska is accurate in terms of detecting smells in NL requirements (P = 89% and R = 89%). However, we also observed that it achieved low precision scores for the detection of particular smells, "5. Not a requirement" (50% in Batch 5), "7. Incomplete system response" (33% in the Batch 5 and 67% in the Batch 6), and a low recall score for detecting the smell "6. Incomplete condition" (67% in the Batch 5). To determine the root causes of such low precision and recall values, we analyzed the false positives and false negatives for each of the smells mentioned above. Note that, for the smell "5. Not a requirement" in Batch 5, our results show 1 true positive, 1242 true negatives, 1 false positive, and no false negatives. For the smell "7. Incomplete system response" in Batch 5, we have 1 true positive, 1241 true negatives, 2 false positives, and no false negatives. Regarding the smell "7. Incomplete system response" in Batch 6, the results have 2 true positives, 2222 true negatives, 1 false positive, and no false negatives. For the smell "6. Incomplete condition" in Batch 5, we obtained 33 true positives, 1184 true negatives, 11 false positives, and 16 false negatives.

5. Not a requirement. Recall from Section 4.1 that this smell occurs when there is a statement that does not contain any requirement segment, i.e., scope, condition, and system response. We observed that Batch 5 has few requirements related to the smell "5. Not a requirement". For the smell, Batch 5 has 1 true positive, 1242 true negatives, 1 false positive, and no false negatives, thus giving a low precision of 50%. The false positive (i.e., incorrectly detected smells) was related to inaccurate POS tags assigned to the verb of the system response. Table 18 shows requirement R1. This requirement has only a system response. The verb of the system response of R1 (i.e., route) was incorrectly identified as a noun. Since no verb was found in the system response of the requirement, Paska detected the smell "5. Not a requirement".

6. Incomplete Condition. Recall from Section 4.1 that this smell occurs when the condition of the requirement misses the verb or the actor. We observed that the cause of false negatives (i.e., Paska failed to detect smells) was related to scenarios not observed during the development of Paska. For the smell "6. Incomplete condition", Batch 5

TABLE 18: Examples of (anonymized) requirements that lead to Paska failing to accurately detect smells and suggest Rimay patterns.

ID	Requirement Description
R1	`The System-A must route the outbound messages to System-B instead of System-C.`
R2	`Upon receipt of a valid C01 cancellation from System-A Participant, then the ↩ System-B must route the cancellation to the same destination.`
R3	`if the System-A Order Issuer Ordering data = Value-A`
R4	`When the user clicks on the Edit icon of Screen-1, System-A must set in updatable ↩ mode the following fields: • Include portfolio in the S-Order, • Alert to operations ↩ (e.g., when Order is rejected), ...`
R5	`If the Participant Status = Delete, then System-A must populate the field Status ↩ with the value inactive.`

has 33 true positives , 1184 true negatives, 11 false positives, and 16 false negatives, resulting in a low recall of 67%. All false negatives were related to new scenarios that were not observed during the development of Paska. Table 18 shows an example (R3) of a false negative. The condition 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". According to the POS Tagger, the word "Ordering" of the compound noun is recognized as a verb, indicating that the condition is complete. Therefore, Paska did not trigger any smells. However, R3 actually misses a verb.

7. Incomplete system response. Recall from Section 4.1 that this smell occurs when the system response misses the actor, the modal verb or the verb. We noted that the cause of false positives (i.e., Paska incorrectly detected smells) was the assignment of incorrect POS tags to the verbs in the system response. Such absence of verb in the system response triggers the smell "7. Incomplete system response". Regarding the smell, Batch 5 has 1 true positive, 1241 true negatives, 2 false positives, and no false negatives, resulting in a low precision of 33%. Furthermore, for the same smell, Batch 6 has 2 true positives, 2222 true negatives, 1 false positive, and no false negatives, resulting in a low precision of 67%. We found that all false positives were related to incorrect POS tags assigned to the verb in the system response. Table 18 shows requirement R2. The verb of the system response of R2 (i.e., route) was incorrectly identified as a noun. Since no verb was found in the system response of the requirement, Paska incorrectly triggered the smell "7. Incomplete system response".

Similarly, the results of RQ5 show that Paska accurately suggests requirement patterns in most cases ($P = 96\%$ and $R = 94\%$). However, we noted that low precision scores were obtained by Paska when suggesting requirement patterns "10. Multiple conditions and system response" (76% in the Batch 4) and "6. Condition (precondition) and system response" (75% in the Batch 6). To determine the root causes of such low precision, we analyzed cases in which Paska yielded false positives (i.e., incorrectly suggested patterns). Note that, for the pattern "10. Multiple conditions and system response", Batch 4 has 25 true positives, 421 true negatives, 8 false positives, and no false negatives. Regarding the pattern "6. Condition (precondition) and system response", Batch 6 has 6 true positives, 474 true negatives, 2 false positives, and 1 false negative.

Pattern: 10. Multiple conditions and system response. This pattern is suggested when a requirement has the following segments: two or more conditions and a system response. For the pattern "10. Multiple conditions and system response", Batch 4 has 25 true positives, 421 true negatives, 8 false positives, and no false negatives, thus obtaining a low precision of 76%. The majority of false positives (i.e., six out of eight) were due to new scenarios that were not observed during the development of Paska. The remaining two false positives were related to human errors in the annotation. Requirement R4 in Table 18 shows an example that causes Paska to incorrectly suggest a Rimay pattern. R4 has a condition with a system response containing bullet points. The second bullet point contains the condition "when Order is rejected", which only applies to the information in the bullet point. However, Paska incorrectly identified it as a condition that applies to the entire requirement; therefore, Paska suggested the Rimay pattern "10. Multiple conditions and system response." Nevertheless, in reality, R4 has only one condition.

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. For the pattern "6. Condition (precondition) and system response", Batch 6 has 6 true positives, 474 true negatives, 2 false positives, and 1 false negative, resulting in a low precision of 75%. False positives were related to new scenarios that were not observed during the development of Paska. Requirement R5, in Table 18, is an example of a false positive (R5). The condition of R5 lacks a verb; instead, R5 has the operator "=" which denotes "equals to". Furthermore, R5 is made up of a compound noun, "Participant Status". The word "Status" of the compound noun is identified as a verb by the POS Tagger which suggests that the condition is a condition of type trigger (Section 5.4). Therefore, Paska suggested the Rimay pattern "7. Condition (trigger) and system response". However, the condition of R5 is a condition of type precondition because of the symbol "=" which denotes "equals to".

In summary, we identified two main reasons for low precision and recall in the cases mentioned above. The first is POS Tagger limitations: POS Tagger incorrectly assigns POS tags to words, which causes Paska to incorrectly identify the smells and syntax of the requirement. Paska does not have control over the accuracy of the POS Tagger, since it is a third-party component. Second, we found several new scenarios that were not previously observed. These scenar-

ios include different structures of the requirement segments and the presence of additional information in requirements. We could enhance Paska to support such new scenarios. However, some of them are examples of bad practices in specifying requirements. For example, in requirement R4 of Table 18, the analyst has inserted a condition as an additional information in the system response. In requirement R5, the symbol "=" is used. However, in general, using such symbols may result in ambiguous interpretations among stakeholders. Hence, it is recommended to avoid using such symbols when writing requirements to prevent confusion.

7.2 Lack of Testing Data

We observed during the development of Paska (Table 14) the lack of testing data to evaluate some smells for certain batches (i.e., "4. Coordination ambiguity" in batches 1 and 3, "5. Not a requirement" in batches 2 and 3, and "7. Incomplete system response" in Batch 3). We were unable to test Paska in the above cases because S^D lacks requirements that contain these smells. However, the aforementioned cases did not occur in all batches of S^D. The smells detected by Paska were all tested in at least one batch during the development of Paska. We also noted that test data were missing to evaluate the suggestion of some Rimay patterns (Table 15): "2. Scope, condition (precondition), and system response" in Batch 3, "4. Scope, condition (time) and system response" in batches 1 to 3, and "8. Condition (time) and system response" in Batch 1. However, the Rimay patterns "2. Scope, condition (precondition), and system response" and "8. Condition (time) and system response" were tested in other batches in S^D during the development of Paska. Regarding the pattern "4. Scope, condition (time) and system response", although the SRSs do not have requirements that can be rewritten by applying the pattern, we opted to keep the pattern in Paska to support the complete list of Rimay patterns. Recall from Section 4.2 that Rimay patterns represent valid sequences of Rimay concepts used to write requirements in Rimay.

Similarly, the results of RQ5 (Table 17) show that Paska was not tested when suggesting the Rimay patterns "2. Scope, condition (precondition), and system response" in batches 4 to 6, "3. Scope, condition (trigger), and system response" in batches 4 and 5, "4. Scope, condition (Time), and system response" in batches 4 to 6, "8. Condition (time) and system response" in batches 4 to 6, and "9. Scope, multiple conditions, and system response" in Batch 4. Paska could not be tested in the above cases because S^E did not include requirements with the syntax necessary to suggest the corresponding Rimay patterns. As described in Section 6.2, the 13 SRSs used to evaluate Paska were collected from our industrial partner, who deemed them representative of recent SRSs and over which we had no control.

8 THREATS TO VALIDITY

Internal validity is of concern when examining causal relations [41]. Our results depend heavily on the quality of annotations, which are susceptible to annotation biases. To minimize any potential biases, we hired three external annotators who did not have access to Paska in our experiments.

To ensure the high quality of their annotations, we provided training sessions and monitored the annotation agreement between annotators using Cohen's Kappa metric. To minimize any biases introduced by our monitoring activities, we limited our inspection of their annotations to the requirements for which the annotators indicated having difficulty with. In addition, at the final stage of the annotation process, we randomly selected 10% of the annotated requirements for inspection.

In our experiments, two external annotators were employed to annotate Rimay patterns. For smells, however, another external annotator and the first author of this article were responsible for annotating the SRSs. To mitigate any experimenter bias introduced by the first author, we included the author's annotations only in the development set (i.e., S^D described in Section 6.2) that was used to develop Paska. Hence, our results obtained from the evaluation set (i.e., S^E described in Section 6.2) were not impacted by the first author's annotations.

Another threat to internal validity concerns potential biases introduced by specific researchers. Recall from Section 4.1 that the first author of this article defined a catalog of nine smells that Paska detects. To mitigate this threat, we validated the nine smells with our industrial partner to ensure their relevance to the errors commonly observed in NL requirements writing. Further, the other authors of this article closely monitored the smell identification process.

Recall from Section 6.5 that we applied the concept of saturation to develop a stable version of Paska. To make this process rigorous and objective, we measured precision and recall after each batch, using a separate test dataset. The saturation point was considered to be reached when we started to observe consistent overall precision and recall from one batch of SRSs to the next.

Paska employs NLP techniques such as tokenization, POS tagging, and constituency parsing, over which it does not have direct control regarding their accuracy. Recognizing the importance of such accuracy and accounting for the latest advances of these techniques, we implemented Paska using well-maintained, state-of-the-art NLP libraries: spaCy [38], Stanford CoreNLP [39], and AllenNLP [34]. Further, these libraries have also been widely applied in various domains [9, 19, 20]. In the end, our choice of techniques and libraries led to an implementation showcasing high precision and recall, indicating that Paska is a promising solution. Nevertheless, we note that supplemental preprocessing of NL requirements (e.g., grammar corrections), which can be executed independently of Paska, has the potential to enhance the accuracy of these NLP techniques. Such enhancements could thus further optimize the performance of Paska.

External validity concerns the degree to which our results can be generalized to other contexts [41]. In our experiments, we evaluated Paska using industrial SRSs that contain NL requirements from 13 systems (SRSs) in the financial domain. Specifically, out of the 13 SRSs, six (S^D) were used for developing Paska, and the remaining seven (S^E) were used in our evaluation to answer RQ4 and RQ5. These requirements are however representative of a broader class of information systems, such as those used by our industrial partner for data management, security compliance,

and communication. Furthermore, the requirements were written by different analysts with different backgrounds, which increases the diversity of the SRSs. We note that, of the 13 SRSs, six were previously used to develop the Rimay language [21]. From these six SRSs, two were allocated to the development set S^D and four to the evaluation set S^E. For this study, we obtained seven new SRSs from our industrial partner. The overlap in SRSs used to define Rimay and to develop and assess Paska is not a threat to validity since, in this work, we assume Rimay is already valid and complete, as previously studied, and here we only assess our ability to detect smells and recommend patterns, clearly dividing SRSs into development and test sets. Though our results should be generalizable to information systems in other domains, future investigations are nevertheless necessary to determine how Paska fares outside finance. In the future, despite the large number of requirements we used in our study, when working on other SRSs, we might uncover patterns we have not identified yet. This would require that we augment or modify the smells related to our catalog of patterns.

Recall from Section 5 that, to identify the beginning of each segment and imprecise verbs in a requirement, Paska uses keyword-based analysis techniques. These techniques rely on glossaries that are created based on the keywords defined in Rimay and our inspection of the representative SRSs used in our study. Furthermore, our industrial partner validated the glossaries. However, further research is needed in order for Paska to rely on more complete glossaries. Due to the simplicity of the techniques, one can easily expand our glossaries by leveraging those defined in existing work (e.g., Smella [9] for detecting requirements smells).

Regarding the use of Tregex, which defines regular expression-like patterns on the syntax tree of a requirement, incorrectly defined patterns may miss the syntax sub-trees they aim to identify. In Paska, these inaccuracies can result in false positives and false negatives when detecting smells and suggesting recommendations. To ensure that we correctly defined the Tregex patterns, we used the development set S^D, which contains diverse requirements written by various analysts for six different systems, as a basis for defining and verifying the patterns. In addition, we inspected the false positives and false negatives from our experiment results on S^E, which show that Paska detected smells with a precision of 89% and a recall of 89%, while suggesting recommendations with a precision of 96% and a recall of 94%. This careful inspection indicates that these false positives and false negatives come from incorrect POS tags, the inclusion of symbols (e.g., "=") in a requirement, and annotation errors, rather than from any potential mistakes we might have introduced in the Tregex patterns. Nevertheless, further studies that analyze actual requirements from different domains are needed to assess the completeness of the Tregex patterns.

9 RELATED WORK

In this section, we compare our work with existing studies related to improving the quality of NL requirements. In particular, we will discuss research strands in the areas

of assisting analysts in (1) writing requirements based on templates, (2) assuring the quality of requirements by analyzing incompleteness and ambiguity in requirements, and (3) identifying smells in requirements.

Requirements templates. Requirements templates (e.g., EARS [24] and Rupp [31]) have been widely used in many studies and in practice [3, 42, 25, 43, 21]. These templates provide a structured approach to writing requirements, which can help reduce ambiguity, increase clarity, and ensure consistency across requirements. However, since requirements templates only provide coarse-grained concepts and constructs (e.g., the EARS condition template `WHEN <event>` does not specifically specify the content of the event that initiates the requirement, allowing analysts to introduce free text), only a limited number of automated analysis techniques that rely on such templates have been introduced [42]. Arora et al. [42] presented an automated approach for checking conformance to requirements templates (e.g., EARS and Rupp). The approach relies on an NLP technique, known as text chunking. In contrast, we rely on Rimay, a state-of-the-art CNL for specifying requirements. Rimay provides structures with fine-grained concepts and constructs that enabled us to develop an automated tool (Paska) for effectively detecting requirements smells and providing Rimay patterns as recommendations for removing smells in requirements.

Quality assurance. Among the many strands of quality assurance research [29] in requirements engineering, the most pertinent ones introduce automated methods for detecting quality problems in requirements. In particular, we discuss prior work that aims at detecting problems of incompleteness and ambiguity in requirements.

Incompleteness analysis. Completeness of requirements is often viewed from two perspectives: external and internal [28]. External completeness ensures that all necessary functionalities of a system are specified in the requirements. Many of the existing studies on the completeness of requirements belong to this research strand [44, 12, 13, 15, 16]. For example, Dalpiaz et al. [15] combined NLP and information visualization techniques to identify missing requirements. Their approach relies on the notion of stakeholders' viewpoints, which helps analysts identify cases in which one viewpoint mention concepts that are not present in other viewpoints. Arora et al. [16] empirically evaluated the usefulness of domain models in detecting incompleteness of requirements. They conducted experiments by seeding some omissions to the requirements and checked whether the domain model can be used to detect the omissions. In contrast to these prior studies, our work is related to the research strand on internal completeness of a requirement, which is concerned with ensuring that the requirement is self-contained. This means that the requirement contains all the information contents required to express its function completely. Recent smell detection work [9, 10] include techniques to detect missing contents in a requirement, such as measurement units and references. In contrast, Paska is able to identify different information contents that are missed in a requirement: i.e., (1) missing actors or verbs in conditions, (2) missing actors, modal verbs, or verbs in system requirements, and (3) missing system responses in requirements, as it relies on Rimay's concepts and constructs. We further

discuss these smell detection approaches below in the smell detection paragraphs.

Ambiguity analysis. Ambiguity is a persistent issue in NL requirements. Hence, such ambiguity has been extensively studied in the literature [17, 18, 19, 11, 20]. For example, recently, Ezzini et al. [20] proposed six alternative solutions for automating the handling of anaphoric ambiguity in requirements. These solutions incorporate both traditional and state-of-the-art NLP and ML techniques, such as Span-BERT [45]. Osama et al. [19] introduced a technique for detecting syntatic ambiguity in a requirement using scored interpretations of the requirement, which provide users with most likely interpretations. More precisely, the technique relies on an NLP algorithm that generates parsing trees with confidence scores to provide scored interpretations of a requirement. However, the research strands in this area differ from our work, which focuses on identifying and addressing requirements smells.

While incompleteness and ambiguity are types of requirements smells, our study aims at detecting a broader range of smells that indicate quality problems in requirements and providing recommendations to solve them. When a requirement has multiple smells, analysts need to account for them together to obtain a complete picture of the overall quality of the requirement, a necessary condition to fix it properly. Paska helps analysts in this process by automatically detecting smells and suggesting patterns to fix them.

Smell detection. The research strands most related to our work are those that aim at detecting smells in NL text, such as requirements, feature requests, and use-case descriptions. Below, we discuss recent studies in this research area: two studies on NL requirements [9, 10], and two others on feature requests [46] and use-case descriptions [47]. All these studies shared the objectives of defining catalogs of smells in NL descriptions and presenting automated smell detection techniques.

Femmer et al. [9] introduced a tool, named Smella, that detects nine smells in NL requirements: subjective language, ambiguous adverbs and adjectives, loopholes, non-verifiable terms, superlatives, comparatives, negative words, vague pronouns, and incomplete references. Smella relies on POS tagging, dictionaries, and lemmatization. Femmer et al. [9] evaluated Smella on 336 requirements, 53 use cases, and 1082 user stories collected from three companies and several university students. Their evaluation results showed that Smella achieved, on average, a precision of 59% and a recall of 82%.

Ferrari et al. [10] presented an approach that detects the following requirement smells: anaphoric ambiguity, coordination ambiguity, vague terms, modal adverbs, passive voice, excessive length, missing condition, missing unit of measurement, missing reference, and undefined term. To detect these smells, they defined a set of smell-detection patterns, i.e., sequences of tokens to be matched within a requirement, relying on NLP techniques such as tokenization, POS tagging, and shallow parsing. They applied the patterns to 1866 requirements obtained from a railway company and obtained a precision of 83% and a recall of 85%.

Mu et al. [46] introduced a tool, named NERO, that annotates contents and detects smells in NL feature requests. For content annotation, NERO uses a rule matcher developed in their previous work [48] to classify a sentence into six categories (e.g., intent and explanation). Using NLP techniques, such as POS tagging, regular expression, and lemmatization, NERO detects 10 smells: vagueness, weakness, generality, coordination ambiguity, referential ambiguity, passive voice, missing condition, missing description, unreadability, and partial content. To evaluate NERO, they applied it to 10 feature requests collected from an issue tracking system.

Seki et al. [47] developed a technique for detecting smells in use case descriptions. They defined a catalog of smells based on seven smell characteristics and five smell scopes. The seven smell characteristics are ambiguity, incorrectness, granularity, redundancy, lack, misplacement, and inconsistency. The five smell scopes are use case, section, flow, sentence, and word. Seki et al. [47] analyzed 30 use case descriptions written in Japanese to define the smell catalog. They applied Goal-Question-Metric (GQM) paradigm [49] to automatically detect a subset of their smell catalog and evaluated their prototype tool using eight use case descriptions written in Japanese.

In contrast to these smell detection techniques, Paska suggests appropriate requirements patterns (based on Rimay) to fix any detected smells in an NL requirement and thus improve the quality of the requirement. In addition, our work relies on a large set of 2725 information system requirements obtained from a financial company. Our experiment results show that Paska is accurate in detecting smells with a precision and recall of 89%. It is worth noting that existing techniques [9, 10] for detecting requirements smells achieved significantly less accurate experimental results. Because of their different focus, Mu et al. [46] and Seki et al. [47] evaluated their techniques on ten feature requests and eight use case descriptions, respectively. We defined our smell catalog based on an analysis of Rimay and 1404 requirements, and then evaluated Paska on the remaining 1321 requirements. We also note that 6 out of 9 smells detected by Paska are not addressed by any of the existing works. These smells are "Incomplete System Response", "Incomplete Condition", "Not Requirement", "Incorrect Order Requirement", "Incomplete Requirement", and "Non-atomic". These smells violate quality attributes — completeness, clarity, atomicity, and correctness — that Rimay enforces in writing requirements (see Section 4).

Since we defined our smell catalog by analyzing Rimay's concepts and constructs, Paska can only detect smells that violate recommendations and best practices guided by Rimay, a language that was defined by qualitatively analyzing information system requirements' needs and practices. Though this approach enables us to provide recommendation as Rimay patterns, we acknowledge that other smells (e.g., non-verifiable terms supported by Smella [9]) or other types of ambiguities (e.g., attachment and analytic ambiguities [19]) are not supported. However, one can easily combine these approaches with Paska and get the combined benefits of all these approaches.

10 CONCLUSIONS

The goal of our work is to better support business analysts in the specification of natural language (NL) requirements by detecting smells in them and by guiding the fixing of such smells. To achieve these objectives, we propose a set of nine smells that represent the most common syntactic and semantic errors found in NL requirements from financial applications. Furthermore, we derived 10 patterns aiming at fixing the smells present in NL requirements and converting NL requirements into requirements expressed in Rimay, a controlled natural language (CNL) that was recently proposed to help define unambiguous and complete requirements. We then devised an automated approach to detect our proposed smells and suggest Rimay patterns to improve the quality of requirements.

We evaluated Paska in a large industrial case study involving 13 system requirements specifications (SRSs) from information systems in the financial domain, containing 2725 human-annotated NL requirements. This evaluation measured the performance of Paska in detecting smells and suggesting accurate Rimay patterns. Our experiment results show that Paska detected smells with a precision and a recall of 89%. Furthermore, Paska suggested Rimay patterns with a precision of 96% and a recall of 94%. Such patterns help the analyst identify what requirement segments are missing, warrant change, or must be re-ordered.

In future work, we intend to expand our list of smells to provide broader coverage of smell detection. Our proposed smells, associated with the quality attributes enforced by Rimay, identify common problems found in the NL requirements of financial applications, which in all likelihood are not specific to that domain. However, they may not represent all syntactic and semantic errors present across all NL requirements. We plan to account for other quality attributes (e.g., comprehensibility and feasibility) [29, 27] to identify and rectify relevant requirement smells. Furthermore, it would be important to conduct a user study to assess the economic benefits that organizations and analysts might reap from integrating Paska into their requirements engineering process. This would provide a more holistic view of Paska's utility and potential return on investment for its users in the development of their IT systems. Lastly, interesting research directions include the use of chatbot interfaces and large language models (LLMs) in Paska. A chatbot interface would allow analysts to query and receive explanations for detected smells in real-time, thereby facilitating the rectification of identified issues through conversational guidance. Furthermore, the use of LLMs could improve Paska's comprehension of NL requirements, enabling the detection of additional quality issues and the suggestion of corresponding recommendations to address them.

REFERENCES

[1] K. Pohl, *Requirements Engineering - Fundamentals, Principles, and Techniques.* Springer, 2010.

[2] M. Kassab, C. J. Neill, and P. A. Laplante, "State of practice in requirements engineering: contemporary data," *Innovations in Systems and Software Engineering*, vol. 10, no. 4, pp. 235–241, 2014.

[3] A. Mavin and P. Wilkinson, "Big ears (the return of "easy approach to requirements engineering")," in *Proceedings of the 18th IEEE International Requirements Engineering Conference*, 2010, pp. 277–282.

[4] D. M. Fernández, S. Wagner, M. Kalinowski, M. Felderer, P. Mafra, A. Vetrò, T. Conte, M. Christiansson, D. Greer, C. Lassenius, T. Männistö, M. Nayabi, M. Oivo, B. Penzenstadler, D. Pfahl, R. Prikladnicki, G. Ruhe, A. Schekelmann, S. Sen, R. O. Spínola, A. Tuzcu, J. L. de la Vara, and R. J. Wieringa, "Naming the pain in requirements engineering - contemporary problems, causes, and effects in practice," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2298–2338, 2017.

[5] J. J. Ahonen and P. Savolainen, "Software engineering projects may fail before they are started: Post-mortem analysis of five cancelled projects," *Journal of Systems and Software*, vol. 83, no. 11, pp. 2175–2187, 2010.

[6] E. C. Hull, K. Jackson, and J. Dick, *Requirements Engineering*, 3rd ed. Springer, 2011.

[7] G. Génova, J. M. Fuentes, J. L. Morillo, O. Hurtado, and V. Moreno, "A framework to measure and improve the quality of textual requirements," *Requirements Engineering*, vol. 18, no. 1, pp. 25–41, 2013.

[8] H. Femmer, D. M. Fernández, E. Jürgens, M. Klose, I. Zimmer, and J. Zimmer, "Rapid requirements checks with requirements smells: two case studies," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, 2014, pp. 10–19.

[9] H. Femmer, D. M. Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," *Journal of Systems and Software*, vol. 123, pp. 190–213, 2017.

[10] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, A. Fantechi, and S. Gnesi, "Detecting requirements defects with NLP patterns: an industrial experience in the railway domain," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3684–3733, 2018.

[11] S. Ezzini, S. Abualhaija, C. Arora, M. Sabetzadeh, and L. C. Briand, "Using domain-specific corpora for improved handling of ambiguity in requirements," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*, 2021, pp. 1485–1497.

[12] H. Kaiya and M. Saeki, "Ontology based requirements analysis: Lightweight semantic processing approach," in *Proceedings of the 5th International Conference on Quality Software*, 2005, pp. 223–230.

[13] A. Ferrari, F. Dell'Orletta, G. O. Spagnolo, and S. Gnesi, "Measuring and improving the completeness of natural language requirements," in *Proceedings of the 20th International Working Conference of Requirements Engineering: Foundation for Software Quality*, vol. 8396, 2014, pp. 23–38.

[14] J. Eckhardt, A. Vogelsang, H. Femmer, and P. Mager, "Challenging incompleteness of performance requirements by sentence patterns," in *Proceedings of the 24th IEEE International Requirements Engineering Conference*, 2016, pp. 46–55.

[15] F. Dalpiaz, I. V. D. Schalk, and G. Lucassen, "Pinpointing ambiguity and incompleteness in requirements engineering via information visualization and NLP," in *Proceedings of the 24th International Working Conference of Requirements Engineering: Foundation for Software Quality*, 2018, pp. 119–135.

[16] C. Arora, M. Sabetzadeh, and L. C. Briand, "An empirical study on the potential usefulness of domain models for completeness checking of requirements," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2509–2539, 2019.

[17] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," *Requirements Engineering*, vol. 13, no. 3, pp. 207–239, 2008.

[18] H. Yang, A. N. D. Roeck, V. Gervasi, A. Willis, and B. Nuseibeh, "Analysing anaphoric ambiguity in natural language requirements," *Requirements Engineering*, vol. 16, no. 3, pp. 163–189, 2011.

[19] M. Osama, A. Zaki-Ismail, M. A. Abdelrazek, J. C. Grundy, and A. S. Ibrahim, "Score-based automatic detection and resolution of syntactic ambiguity in natural language requirements," in *Proceedings of 2020 the IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 651–661.

[20] S. Ezzini, S. Abualhaija, C. Arora, and M. Sabetzadeh, "Automated handling of anaphoric ambiguity in requirements: A multi-solution study," in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 187–199.

[21] A. Veizaga, M. Alférez, D. Torre, M. Sabetzadeh, and L. C. Briand, "On systematically building a controlled natural language for functional requirements," *Empirical Software Engineering*, vol. 26, no. 4, p. 79, 2021.

[22] A. Veizaga, M. Alférez, D. Torre, M. Sabetzadeh, L. C. Briand, and E. Pitskhelauri, "Leveraging natural-language requirements for deriving better acceptance criteria from models," in *Proceedings of the ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 218–228.

[23] R. Levy and G. Andrew, "Tregex and tsurgeon: tools for querying and manipulating tree data structures," in *Proceedings of the Fifth International Conference on Language Resources and Evaluation*, 2006, pp. 2231–2234.

[24] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Proceedings of the 17th IEEE International Requirements Engineering Conference*, 2009, pp. 317–322.

[25] A. Mavin, P. Wilkinson, S. Gregory, and E. Uusitalo, "Listens learned (8 lessons learned applying EARS)," in *Proceedings of the 24th IEEE International Requirements Engineering Conference*, 2016, pp. 276–282.

[26] OMG, "Unified modeling language. version 2.5.1," 2017, accessed 13 February 2023. [Online]. Available: https://www.omg.org/spec/UML/

[27] L. Montgomery, D. Fucci, A. Bouraffa, L. Scholz, and W. Maalej, "Empirical research on requirements quality: a systematic mapping study," *Requirements Engineering*, vol. 27, no. 2, pp. 183–209, 2022.

[28] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," *Information and Software Technology*, vol. 45, pp. 993–1009, 2003.

[29] C. Denger and T. Olsson, *Quality Assurance in Requirements Engineering*. Springer Berlin Heidelberg, 2005, pp. 163–185.

[30] "Enterprise architect," https://www.sparxsystems.eu/, accessed: 2023-03-22.

[31] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals - A Study Guide for the Certified Professional for Requirements Engineering Exam: Foundation Level - IREB compliant*. rockynook, 2011.

[32] I. Sommerville, *Software Engineering*, 9th ed. Dorling Kindersley, 2011.

[33] B. Glaser and A. Strauss, *The discovery of grounded theory: strategies for qualitative research*, 1st ed. Aldine Transaction, 2017.

[34] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. Peters, M. Schmitz, and L. S. Zettlemoyer, "Allennlp: A deep semantic natural language processing platform," in *Proceedings of the 2018 Workshop for NLP Open Source Software*, 2018.

[35] D. M. Berry, E. Kamsties, and M. M. Krieger, "From contract drafting to software specification: Linguistic sources of ambiguity," *Handbook*, pp. 1–80, 2003.

[36] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012.

[37] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[38] Explosion, "spacy: Industrial-strength natural language processing in python," https://spacy.io/, 2021, accessed: 2023-04-03.

[39] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, M. A. Hearst and M. Ostendorf, Eds., 2003.

[40] A. Veizaga, S. Y. Shin, and L. C. Briand, "Paska - automated smell detection and recommendation in natural language requirements," https://doi.org/10.6084/m9.figshare.22731707.v1, 2023.

[41] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley Publishing, 2012.

[42] C. Arora, M. Sabetzadeh, L. C. Briand, and F. Zimmer, "Automated checking of conformance to requirements templates using natural language processing," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 944–968, 2015.

[43] A. Sleimi, M. Ceci, M. Sabetzadeh, L. C. Briand, and J. Dann, "Automated recommendation of templates for legal requirements," in *Proceedings of the 28th IEEE*

International Requirements Engineering Conference, 2020, pp. 158–168.

[44] R. J. Costello and D. Liu, "Metrics for requirements engineering," *Journal of Systems and Software*, vol. 29, no. 1, pp. 39–63, 1995.

[45] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, "Spanbert: Improving pre-training by representing and predicting spans," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 64–77, 2020.

[46] F. Mu, L. Shi, W. Zhou, Y. Zhang, and H. Zhao, "NERO: A text-based tool for content annotation and detection of smells in feature requests," in *Proceedings of the 28th IEEE International Requirements Engineering Conference*, 2020, pp. 400–403.

[47] Y. Seki, S. Hayashi, and M. Saeki, "Detecting bad smells in use case descriptions," in *Proceedings of the 27th IEEE International Requirements Engineering Conference*, D. E. Damian, A. Perini, and S. Lee, Eds., 2019, pp. 98–108.

[48] L. Shi, C. Chen, Q. Wang, S. Li, and B. W. Boehm, "Understanding feature requests by leveraging fuzzy method and linguistic analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 440–450.

[49] V. R. Basili, G. Caldiera, and H. D. Rombach, "Goal, question, metric paradigm," *Encyclopedia of Software Engineering*, vol. 2, pp. 528–532, 1994.

Lionel C. Briand (Fellow, IEEE) is professor of software engineering and has shared appointments between (1) School of Electrical Engineering and Computer Science, University of Ottawa, Canada and (2) The Lero SFI Centre for Software Research, University of Limerick, Ireland. He is a Canada research chair in Intelligent Software Dependability and Compliance (Tier 1) and the director of Lero. He has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. He is a fellow of the IEEE, ACM, and Royal Society of Canada. He was also granted the IEEE Computer Society Harlan Mills Award (2012), the IEEE Reliability Society Engineer-of-the-year award (2013), and the ACM SIGSOFT Outstanding Research Award (2022) for his work on software testing and verification. His research interests include: software testing and verification (including security aspects), trustworthy AI, applications of AI in software engineering, model-driven software development, requirements engineering, and empirical software engineering. Further details can be found on: www.lbriand.info.

Alvaro Veizaga has been a research associate at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. Alvaro received his MSc from Leibniz University Hannover (Germany) and his Ph.D. in Computer Science from the University of Luxembourg in 2018 and 2022, respectively. His research interests are requirements engineering, software testing, machine learning, and natural language processing.

Seung Yeob Shin (Member, IEEE) is a Research Scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He received his Ph.D. degree from the Laboratory for Advanced Software Engineering Research (LASER) in the College of Information and Computer Sciences at the University of Massachusetts Amherst in 2016. His research interests are in software engineering, focusing on model-driven software development, search-based software engineering, empirical software engineering, requirements engineering, and analysis of complex systems.