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

Alvaro Veizaga[*], Mauricio Alferez[*], Damiano Torre[*], Mehrdad Sabetzadeh[†][*], Lionel Briand[*][†]
[*]University of Luxembourg, Luxembourg
[†]University of Ottawa, Canada
{firstname.lastname}@uni.lu,m.sabetzadeh@uottawa.ca

Elene Pitskhelauri
Clearstream Services SA, Luxembourg
elene.pitskhelauri@clearstream.com

ABSTRACT

In many software and systems development projects, analysts specify requirements using a combination of modeling and natural language (NL). In such situations, systematic acceptance testing poses a challenge because defining the acceptance criteria (AC) to be met by the system under test has to account not only for the information in the (requirements) model but also that in the NL requirements. In other words, neither models nor NL requirements per se provide a complete picture of the information content relevant to AC. Our work in this paper is prompted by the observation that a reconciliation of the information content in NL requirements and models is necessary for obtaining precise AC. We perform such reconciliation by devising an approach that automatically extracts AC-related information from NL requirements and helps modelers enrich their model with the extracted information. An existing AC derivation technique is then applied to the model that has now been enriched by the information extracted from NL requirements.

Using a real case study from the financial domain, we evaluate the usefulness of the AC-related model enrichments recommended by our approach. Our evaluation results are very promising: Over our case study system, a group of five domain experts found 89% of the recommended enrichments relevant to AC and yet absent from the original model (precision of 89%). Furthermore, the experts could not pinpoint any additional information in the NL requirements which was relevant to AC but which had not already been brought to their attention by our approach (recall of 100%).

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Abstraction, modeling and modularity**; **Model-driven software engineering**; **Acceptance testing**; • **Information systems → Information extraction**.

KEYWORDS

Requirements Validation and Verification, Acceptance Testing, Acceptance Criteria, UML, Controlled Natural Language, Gherkin

ACM Reference Format:
Alvaro Veizaga[*], Mauricio Alferez[*], Damiano Torre[*], Mehrdad Sabetzadeh[†][*], Lionel Briand[*][†] and Elene Pitskhelauri. 2020. Leveraging Natural-language Requirements for Deriving Better Acceptance Criteria from Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410953

1 INTRODUCTION

Acceptance testing is aimed at determining whether a system under test (SUT) meets its specified requirements [2]. A key step in acceptance testing is defining the Acceptance Criteria (AC) for the SUT. AC are conditions that the SUT must satisfy in order for the SUT to be accepted by its users or customers. Naturally, AC are derived from the requirements. It is desirable to make the AC derivation process as automated as possible, noting that, without automated support, it would be very tedious for the analysts to define the AC in a systematic and complete manner. This is specially true for complex systems with large numbers of requirements and for systems whose requirements evolve frequently.

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

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

Model: Figure 1 presents a UML Activity Diagram related to setting up an order for purchasing bonds. If the (order) data is correct, an order is created; otherwise, an alerting process kicks in to notify the bond trader about the data anomaly.

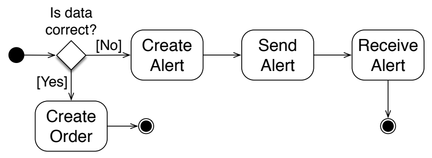
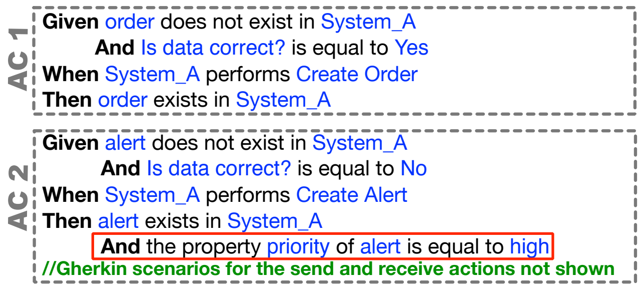


Figure 1: Example of a (requirements) model.

NL Requirement: The NL requirement is as follows:


AC 1
Given order does not exist in System_A
And Is data correct? is equal to Yes
When System_A performs Create Order
Then order exists in System_A

AC 2
Given alert does not exist in System_A
And Is data correct? is equal to No
When System_A performs Create Alert
Then alert exists in System_A
And the property priority of alert is equal to high
//Gherkin scenarios for the send and receive actions not shown

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

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

This NL requirement is concerned with the behavior that is expected when an alert is created. As we are going to discuss in Section 2, in this paper, we use an existing controlled language, named Rimay [15], for writing NL requirements. The above-stated requirement, R, complies with Rimay's grammar.

AC Derivation: To generate AC, we employ an existing technique, named AGAC (Automated Generation of Acceptance Criteria) and its associated tool [1]. The AC produced by AGAC are represented in the Gherkin scenario language [17]. Gherkin scenarios follow a predefined (textual) template: *Given* [initial context], *When* [event or action], *Then* [expected result]. In AGAC, each acceptance criterion is captured by means of a sequence of Gherkin scenarios. Each such sequence exercises an end-to-end system behavior, starting from the initial node of an Activity Diagram and going all the way to a final node. Figure 2 shows the AC, AC 1 and AC 2, that one would intuitively expect for exercising the two alternative flows of the model in Figure 1[1]. To save space, we have truncated AC 2 by hiding the Gherkin scenarios induced by the *Send Alert* and *Receive Alert* actions in the model of Figure 1. In Figure 2, Gherkin's keywords are in bold. The fixed, predefined text coming from AGAC's AC templates is in regular black font. The text obtained from the model of Figure 1 or the NL requirement R is in blue.

The example in Figure 2 highlights the fact that neither models nor NL requirements provide a complete picture of what is relevant to AC generation. Specifically, one cannot expect to find in the NL requirements alone all the information that is pertinent to AC. Notably, control-flow behaviors, e.g., the ordering of the actions in our example model of Figure 1, are often entirely absent from the NL requirements. The result is that, based on NL requirements alone, one typically cannot synthesize end-to-end system behaviors; exercising these behaviors is, however, critical to acceptance testing.

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

[1]AGAC supports AC generation from a model with (1) parallel flows via standard depth- and breadth-first search [4] to traverse non-concurrent and concurrent nodes, and (2) loops via a bounded unwinding of the loops. These are important considerations for deriving end-to-end AC, but are orthogonal to our illustrating example.

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

The paper investigates three Research Questions (RQs):

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

RQ2: How can we systematically enrich models with the (AC-related) information from NL requirements? We answer RQ2 by proposing a systematic method that generates recommendations for model enrichment, based on the information extracted by the rules developed in response to RQ1 (second contribution). The method identifies the model elements that can be enriched with the extracted information and provides guidance to the analysts as to how they can incorporate this additional information into the model. Subsequently, an existing model-based AC derivation technique, AGAC [1], is applied to the enriched model. This way, we make it possible to account for the information in *both* the model and the NL requirements during AC derivation.

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

This rest of this paper is structured as follows: Section 2 provides background. Section 3 presents an overview of our model enrichment approach. Section 4 explains and illustrates the details of the approach. Section 5 reports on the evaluation of the approach in an industrial setting. Section 6 discusses threats to validity. Section 7 compares with related work. Section 8 concludes the paper.

2 BACKGROUND

2.1 Writing NL Requirements in Rimay.

Rimay, proposed by Veizaga et al. [15], is a CNL for writing requirements in the domain of information systems, initially validated in the financial domain. We use Rimay because: (1) it is a suitable match for our case study context. Indeed, our industrial partner is already using Rimay to write NL requirements for some of their banking and securities applications; (2) as a CNL, Rimay comes with precise syntax and semantics. The first characteristic ensures that one has enough expressive power to capture the NL requirements in our case study. The second characteristic enables us to devise structured and highly accurate rules for extracting AC-related information from NL requirements, thereby alleviating the need for heuristics based on natural language processing and machine learning, which are typically less accurate.

Rimay's main grammar rules are inspired by the Easy Approach to Requirements Syntax (EARS) templates [8]. EARS is considered by many practitioners to be a good trade-off between flexibility and precision, due to EARS' relatively low training overhead and the quality and readability of the resultant requirements [7].

The rule `REQUIREMENT` shown in Listing 1 provides the overall syntax for a requirement in Rimay. The rule shows that the presence of the `SCOPE` and `CONDITION_STRUCTURES` is optional, but the presence of an `ACTOR`, `MODAL_VERB` and a `SYSTEM_RESPONSE` is mandatory in all requirements.

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

Listing 1: Overall syntax of a requirement in Rimay.

In a requirement, an actor is expected to achieve a system response if some conditions are true. An actor is a role played by an entity that interacts with the system by exchanging signals, data or information [10]. Moreover, requirements written in Rimay may have a scope to delimit the effects of the system response. One example of a requirement written in Rimay is R1: `When the Order_Issuer (hereafter known as OI) creates an Order of type Subscription_Order, then OI must set the settlement_method of the Order to "FOP"`. Requirement R1 does not have a scope, and has a condition (`When the Order_Issuer...creates an Order of type Subscription_Order`), an actor (`Order_Issuer`), an alias for the actor Order_Issuer (`OI`), and a system response (`OI must set the settlement_method of the Order to "FOP"`).

Rimay's grammar further incorporates some common constructs such as `MODAL_VERB` (e.g., shall, must) and `MODIFIER`, including `ARTICLE` (e.g., a, an, the) and `QUANTIFIER` (e.g., each, all, none, only one, any). For example, the rule `SCOPE`, shown in line one of Listing 2, uses the keyword `For`, and the rules `MODIFIER` and `TEXT`, in the phrase `For the "first 5 columns of the file"`.

```
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 #TEXT is a feature expression
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.

The grammar rule named `CONDITION_STRUCTURE` shown in Listing 2 (Line 2) defines different ways to use system states, triggering events, and features, to express conditions that must hold for the system responses to be triggered. Lines 3-7 of Listing 2 show the rules for the condition structures `WHILE`, `WHEN`, `WHERE`, `IF`, and `TEMPORAL_STRUCTURE` [2]. The grammar rule named `CONDITION_STRUCTURES` in Listing 2 (Line 8) enables the creation of a condition composed of two or more of the above-mentioned condition structures.

2.2 Automated Generation of AC.

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

The second task, *Derive AC*, is automated. This task matches the model created in the first task to a set of predefined AC templates, based on the intents of the actions in the model. The appropriate templates are then instantiated (potentially multiple times), producing AC represented in the Gherkin language [17]. More specifically, the templates define the fixed parts of the text in the Given-When-Then structure of a Gherkin scenario as well as the variable parts (placeholders) that need to be filled with content from the model. What enables the identification of the appropriate template for a given action is the tag `@Intent`, which is specified in every AC template.

3 APPROACH OVERVIEW

In this section, we introduce the inputs to our approach, followed by an overview of the different steps of the approach (Figure 3).

The input to the approach is a *Requirements Specification*. This specification is composed of a set of NL requirements and a requirements model (hereafter referred to as a model). In our context, a

[2]The `TEMPORAL_STRUCTURE` is used when the system responses are triggered before or after an event. For example, the phrase `Before "8:00 am", every "calendar day"` is a temporal structure composed of two prepositional phrases.

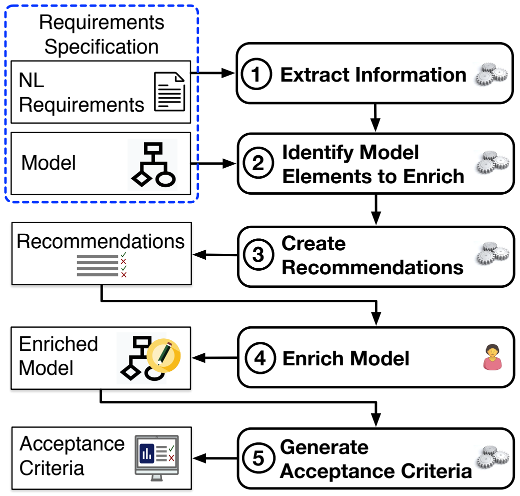


Figure 3: Approach overview.

model is composed of two types of model artifacts: 1) UML diagrams (specifically, CDs, ADs, and UCDs) and 2) a traceability matrix.

NL Requirements. Natural languages (NL), such as English, are commonly used for expressing systems and software requirements [11]. Table 1 shows an example of five NL requirements from the financial domain. The requirements in our example are uniquely identified with Ids composed of the letter R followed by a digit. Each requirement follows the requirement syntax defined by the Rimay language introduced in Section 2. In requirement R1 shown in Table 1, there is a triggering condition (`When the Order_Issuer...creates an ↩ Order of type Subscription_Order`), an actor (`Order_Issuer`) that responds to the trigger, and one system response (`set the ↩ settlement_method of the Order to "FOP"`). Free of payment (FOP) is a securities industry settlement method that is not linked to a corresponding transfer of funds. In this case, only the securities are moved. An example involving a FOP transaction may be gifts or donations. In requirements R2, R3 and R4 (Table 1) , we use the generic names `System`, `Transfer_System`, and `Data_Provider` to anonymize the names of the settlement platform, the secure file transfer connectivity system, and the funds data provider of our industrial partner. Moreover, we use the name `File` to refer to a set of specific information about fund documents. For example, each line of the `File` may include the document type identifier, the codes of the countries in which the document can be published, the `ISIN` (International Securities Identification Number) code defining the share class for the document, a flag indicating if the document is written for a specific group of investors only, or the document URL. Requirement R5 (Table 1) includes the terms `ISIN` and `Share_Class_Identifier`. `ISIN` is a code that uniquely identifies a specific securities issue. `Share_Class_Identifier` is a designation applied to a type of security, such as a mutual fund unit in the settlement platform of our industrial partner.

Model. We assume that the input requirement model has been created by following the AGAC methodology. AGAC uses three types of UML diagrams to represent requirements: UCDs, CDs and ADs. Figure 4 shows *excerpts* of diagrams that can be enriched with information extracted from the NL requirements shown in Table 1. *Actors* are defined in UCDs (e.g., the actor `Order_Issuer`) and execute *actions* that are part of the *activities* represented in ADs (e.g., `Create Order`). *Domain entities* and their properties are characterized by a *domain model*, represented using CDs and referenced within the ADs. For example, the type `Subscription_Order`, shown in the CD, has one property named `settlement_date`. In the AD `Create subscription order` at the top of Figure 4, the action `Create Order` creates the object of type `Subscription_Order`, which is specified in the domain model.

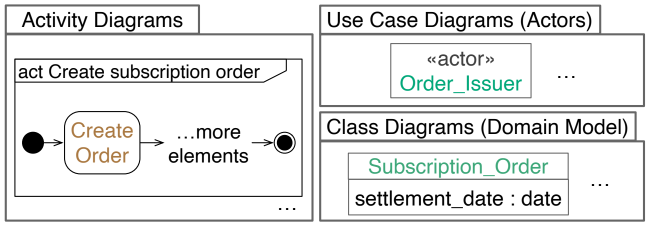


Figure 4: Model excerpts.

Models typically include trace relationships (traces) between model elements that are mainly used in UML for tracking requirements and changes across models [10]. Those traces are usually represented in a traceability matrix. In our context, traces between NL requirements and AD actions are sufficient for our purpose as the extraction rules are driven by the control flow captured by actions (as we explain later in Step 1 of Section 4.1). Table 2 shows an example traceability matrix where the columns represent actions from Table 3 (discussed in Section 4.1), and the rows represent NL requirements from Table 1. An "X" in Table 2 means that the action in the column is *traced to* the NL requirement in the row, and vice versa. We also say that the action in the column is a *traced action* of the NL requirement. The Rimay tool is tightly integrated with the modeling environment and ensures that trace relationships are established correctly during requirements writing.

The Approach. Our approach is composed of five steps, as depicted in Figure 3. All the steps, except "Enrich Model" (Step 4), are performed automatically.

Step 1 is concerned with extracting information (e.g., actors, classes, properties, conditions) from NL requirements expressed using the Rimay language (see Section 2.1). Only the NL requirements that are traced to model elements are analyzed during Step 1.

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

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

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

Table 1: Example NL requirements.

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

Table 2: Traceability matrix.

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

Step 5 is concerned with automatically generating AC from the enriched model. This step is performed via AGAC (see Section 2.2).

4 INFORMATION EXTRACTION APPROACH FOR DERIVING BETTER AC

In this section, we describe in detail and illustrate over a small example the five steps of the approach of Figure 3.

4.1 Step 1. Extract Information

This step aims to answer RQ1. To do so, we propose 13 rules to extract information content from NL requirements that is relevant for generating AC. Our extraction rules were borne out of an analysis of the AC-related conceptual overlaps between the element types in models and the element types in Rimay, discussed in Section 2. To derive the rules, we systematically analyzed the Rimay grammar, identified correspondences between the grammar and the element types in the models, and finally defined the extraction rules.

Table 3 shows our extraction rules. The rules are organized into four categories that correspond to the main grammar rules of Rimay from which the information is extracted. These categories are: (a) `SCOPE`, (b) `CONDITION_STRUCTURE`, (c) `ACTOR`, and (d) `SYSTEM_RESPONSE`.

Each extraction rule in Table 3 has three columns "ID", "Extraction Rule / Recommendation" and "Example". The "ID" column uniquely identifies an extraction rule and is composed of the first letter(s) of the rule's category name and a number. For instance, `S1` and `SR3` are the first and third extraction rules related to the categories `SCOPE` and `SYSTEM_RESPONSE`, respectively.

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

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

Step 1 (Figure 3) is composed of two sub-steps: *Sub-Step 1.1* identifies the NL requirements that are traced to model elements (i.e., AD actions), based on a traceability matrix. If an NL requirement is not traced to any model element, our tool will show a warning message to the analysts. For instance, NL requirements `R1` to `R5` (Table 1) will be selected in Sub-Step 1.1 because they are traced to at least one model element according to the traceability matrix shown in Table 2. *Sub-Step 1.2* extracts information (e.g., conditions, actors, triggers, verb phrases, etc.) from the NL requirements selected in Sub-Step 1.1. The information to be extracted is determined by the rules shown in Table 3. Each rule extracts information from a specific part of an NL requirement. For instance, extraction rule `C1` (Table 3) extracts verb phrases from the `When` structure of NL requirements. In the case of `R2` (Table 1), `C1` extracts the verb phrase "receives a File". To give a more complete example of Sub-Step 1.2, Figure 5 shows requirement `R1` alongside the AC-related information that can be extracted using the extraction rules of Table 3. According to extraction rules `C1`, `A1`, `A2`, and `SR1` to `SR3`, the type of information extracted from `R1` is actor, actor alias, action, object type, property name, object name, and property value. The yellow marks in Figure 5 indicate the information content in `R1` extracted by the extraction rules. For instance, "property value" (e.g., "`FOP`") is a type of information extracted by the rule `SR2`.

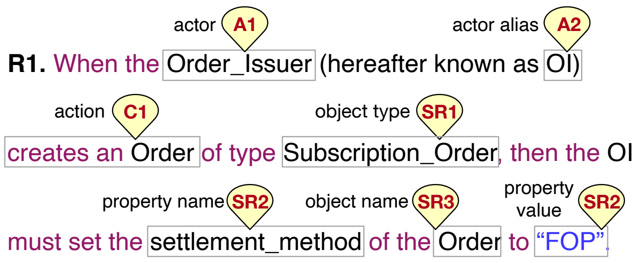
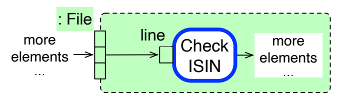

actor
A1
actor alias
A2
R1. When the Order_Issuer (hereafter known as OI)
action
C1
object type
SR1
creates an Order of type Subscription_Order, then the OI
property name
SR2
object name
SR3
property value
SR2
must set the settlement_method of the Order to "FOP".

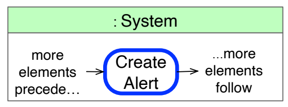
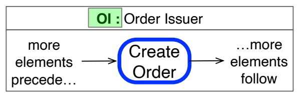
Figure 5: Illustration of information extraction applied to requirement R1 from Table 1: the extracted information is shown in rectangular boxes and the corresponding extraction rules in pin-shaped pointers.

Table 3: Information extraction rules for NL requirements written in Rimay and the associated model-enrichment recommendations. The table is organized according to the different grammar rules of Rimay: (a) SCOPE, (b) CONDITION_STRUCTURE, (c) ACTOR, and (d) SYSTEM_RESPONSE.

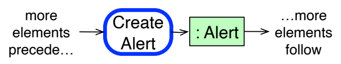
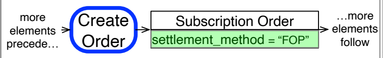
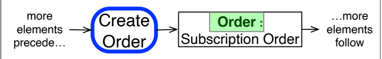
a)

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

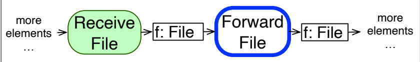
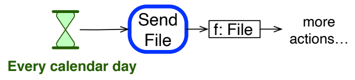
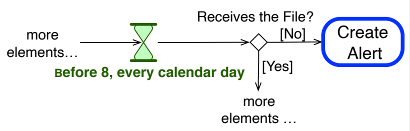
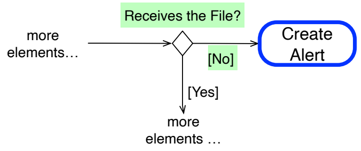
c)

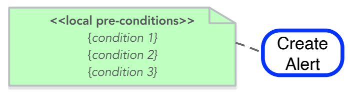
ID	Extraction Rule / Recommendation	Example
A1	**Rule:** If an actor ***A*** in an NL requirement does not match the name of any UML actor linked to the activity partition of the traced action, then extract ***A***. **Recommendation:** (1) Create a UML actor and named it ***A***, (2) create an activity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

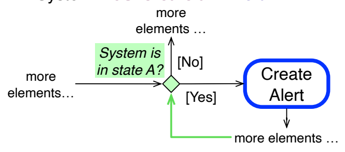
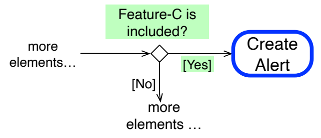
d)

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

b)

ID	Extraction Rule / Recommendation	Example
C1	**Rule:** If the verb phrase ***A*** in a When structure does not match the name of any of the actions preceding the traced action, then extract ***A***. **Recommendation:** Create an action named ***A***.	**R2: When** Transfer_System **receives** a File Transfer_System **must forward** the File **to** System.  more elements … Receive File f: File Forward File f: File more elements …
C2	**Rule:** If a prepositional phrase ***A*** expresses a timed event, and the timed event does not match any of the events or actions preceding the traced action, then extract ***A***. **Recommendation:** Create a time-triggered event named ***A***.	**R3: Every** "calendar day" Data_Provider **must send** a File. 
C3	**Rule:** If two prepositional phrases ***A*** and ***B*** express a timed event and do not match any of the events or actions preceding the traced action, then extract ***A*** and ***B***. **Recommendation:** Create a time-triggered event that combines the information of ***A*** and ***B***.	**R4: Before** "8:00 am", **every** "calendar day", **if** System **does not receive the** File, **then** System **must create an** "Alert".  more elements… Receives the File? [No] Create Alert Before 8, every calendar day [Yes] more elements …
C4	**Rule:** If a condition ***A*** in an If structure does not match any decision node preceding the traced action, then extract ***A***. **Recommendation:** (1) create a decision node, (2) change negative conditions to positive ones (e.g., change from "does not receive" to "receives"), and (3) name the decision node ***A***.	**R4: Before** "8:00 am", **every** "calendar day", **if** System **does not receive the** File **then** System **must create an** "Alert".  Receives the File? more elements… Create Alert [No] [Yes] more elements …

C5	**Rule:** If the condition ***A*** in an If structure does not match any decision node preceding the traced action, then extract ***A***. **Recommendation:** Create a local pre-condition (constraint) in the traced action for each sub-condition in ***A***.	**R4 (modified):** **If** "condition 1", **If** "condition 2", **and If** "condition 3", **then** System **must create an** "Alert".  <<local pre-conditions>>
		{condition 1}
		{condition 2}
		{condition 3}
		Create Alert

C6	**Rule:** If a condition ***A*** in a While structure does not match any decision node in a pre-test loop preceding the traced action, then extract ***A***. **Recommendation:** Name ***A*** the decision node in the pre-test loop.	**R4 (modified): While** "System is in state A", System **must create an** "Alert".  more elements … System is in state A? [No] more elements… Create Alert [Yes] more elements …
C7	**Rule:** If a condition ***A*** in a Where structure does not match any decision node preceding the traced action, then extract ***A***. **Recommendation:** Create a decision node, and name it ***A***	**R4 (modified): Where** "Feature-C is included", System **must create an** "Alert".  Feature-C is included? more elements… Create Alert [Yes] [No] more elements …

4.2 Step 2. Identify Model Elements to Enrich

This step uses the information extracted in Step 1 (Figure 3) in order to identify the model elements (e.g., activity partition, action, object, property, etc) that can be enriched. A model element is enriched when we add to it new AC-related information extracted from NL requirements using the extraction rules in Step 1.

In order to identify what model elements need to be enriched, our approach automatically compares the text sequences extracted from NL requirements (i.e., the information extracted in Step 1) to the names of the elements in the model. Specifically, our approach (1) runs a pre-processing step that includes lowercase text conversion, stemming (a process of reducing inflected words to their word stem), and stop-word removal (words such as articles are removed), and (2) deems two text sequences as matching when they are syntactically identical. To illustrate how the matching works, consider rule `C1` and its example shown in Table 3 (b). According to `C1`, our approach needs to determine if there is an action named "receives a File" that precedes the action "Forward File". If such an action does not exist, our approach classifies the model element as *enrichable.*

For example, given the information extracted from Step 1 (actor: `Order_Issuer`, actor alias: `OI`, object name: `Order`, object type: `Subscription_Order`, property name: `settlement_method`, and property value: `FOP`), our approach identifies six model elements in Figure 4 that are enrichable for the following reasons:

(1) The AD does not have an activity partition linked to the actor "`Order_Issuer`" defined in the UCD (rule `A1`).
(2) There is no activity partition named "`OI`" in the AD (rule `A2`).
(3) The action "Create Order" in the AD does not have any output object node of type "`Subscription_Order`" (rule `SR1`).
(4) The output object node of type "Subscription_Order" does not have the property "settlement_method" (rule `SR2`).
(5) The property "settlement_method" is not set to "FOP" in the output object node of type "Subscription_Order" (rule `SR2`).
(6) There is no object node named "Order" in the AD (rule `SR3`).

4.3 Step 3. Create Recommendations

This step creates recommendations for the analysts regarding how to enrich the model elements classified as enrichable in Step 2 (Figure 3). All the recommendations were shown already in Table 3. Each recommendation describes the tasks that analysts should perform to enrich a model using AC-related information extracted by a rule shown in the same cell of Table 3.

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

The recommendations use mostly terminology from UML e.g., local-precondition, event, condition, expansion region. In addition, other recommendations use terminology derived from programming languages. For instance, the recommendation of the extraction rule `C6` (Table 3) uses the term *pre-test loop.* A pre-test loop is one in which a set of actions is to be repeated until a specified condition is no longer true, and the condition is tested before the set of actions is executed. In modern programming languages, a pre-test loop is implemented using the *while* statement.

Step 3 uses the general recommendations shown in Table 3 to produce specific recommendations for the model being created by the analysts. To illustrate the type of specific recommendations produced by Step 3, Table 4 shows five recommendations to enrich the model shown in Figure 4. The recommendations described in the "Description" column (Table 4) contain predefined text and text extracted from the NL requirements. The predefined text is in normal black and the text obtained from the NL requirements is in bold. For instance, in recommendation `Rec.1` (Table 4) the `Order_Issuer` is a model element extracted from the NL requirement that is used to enrich the model.

Table 4: Recommendations to enrich the model of Figure 4.

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

4.4 Step 4. Enrich Model

In this step, the analysts consider the recommendations generated by our approach. The recommendations contain the steps that the analysts have to perform to enrich the model. This step is carried out manually because the analysts should have the final say as to whether to follow or discard the recommendations. Figure 6 exemplifies the output of Step 4 for the model shown in Figure 4 and requirement `R1` in Figure 5, according to the six recommendations shown in Table 4. Note that, in Figure 6, we have assumed that the analysts would accept all the recommendations. We use comments (identified with the ids of the extraction rules that produced each recommendation) to mark the places in the model that have been enriched by the analysts.

4.5 Step 5. Generate Acceptance Criteria

This step automatically generates AC based on the intents of the AD actions in the model enriched by the analysts (Step 4). To generate AC, we use a set of predefined AC templates provided by AGAC (see Section 2.2). For example, the acceptance criterion shown in Figure 7 is generated from the action named "Create Subscription Order".

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

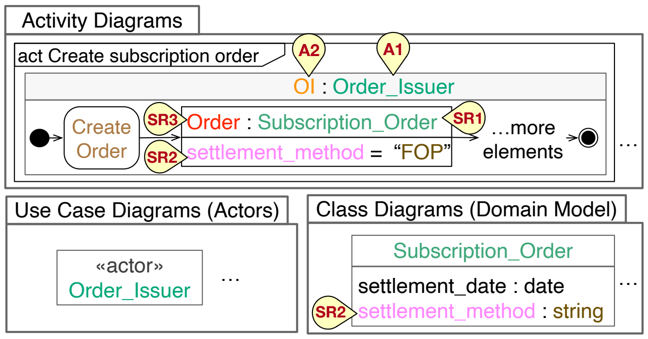

Activity Diagrams
act Create subscription order
A2
A1
OI : Order_Issuer
Create Order
SR3
Order : Subscription_Order
SR1
SR2
settlement_method = "FOP"
...more elements
...
Use Case Diagrams (Actors)
«actor» Order_Issuer
...
Class Diagrams (Domain Model)
Subscription_Order
settlement_date : date
SR2
settlement_method : string
...

Figure 6: Enriched model and the mapping of new elements to the extraction rules of Table 3.

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

Figure 7: Example acceptance criterion related to the model of Figure 6.

5 EMPIRICAL EVALUATION

In this section, we describe the case study we carried out to address RQ3. Throughout the section, we follow best practices for reporting on case study research in software engineering [12].

5.1 Objectives and Design

We evaluated our approach using a real requirements specification developed by our industrial partner. The specification includes three ADs, three CDs, one UCD, and 23 NL requirements. We reflect on the representativeness of this case study in Section 6.

We applied the 13 extraction rules shown in Table 3 to the 23 NL requirements and obtained 27 recommendations for the analysts to enrich the model. We carried out this evaluation in close collaboration with five analysts at our industrial partner. All the analysts were domain experts, with significant experience writing financial system requirements ranging from 7 to 28 years. We asked these analysts to answer the following two questions:

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

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

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

5.2 Preparation for Data Collection

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

5.3 Collecting Evidence and Results

We report on the analysis of the data collected from the questionnaire's answers. Table 5 shows the number of Yes and No answers to Q1 and Q2, given by the group of analysts after they reached consensus.

Table 5: Questionnaire answers.

Question	Yes	No
Q1	24	3
Q2	7	17

With regard to Q1, the analysts agreed to follow 24 of the 27 recommendations to enrich the model. Table 6 compares the original model and the enriched one in terms of numbers of elements of different types. The increase in instances across all element types in the *enriched* model confirms that our approach is effective in creating recommendations that the analysts deem necessary for the purpose of AC generation. After enriching the model, the analysts generated AC. Table 7 provides a comparison of the AC that were generated from the original and the enriched model. In AGAC, one AC is described using a sequence of one or more Gherkin scenarios. Therefore, we compared AC in terms of the total number of Gherkin scenarios (`Given-When-Then` structures), and the pre- and post-conditions in their `Given` and `Then` parts, respectively. Though there is a general, significant increase, the most striking value in Table 7 is the 596% increase in the number of post-conditions. This result was mainly due to the addition of 75 values for the properties of action's output objects (Table 7) that AGAC transforms into post-conditions in the Gherkin scenarios. As a result, along with a more modest increase in pre-conditions, this contributes to making AC more precise. Another noteworthy result is the 22% increase in Gherkin scenarios, thus showing that model refinement leads to more complete AC.

Table 6: Comparison of the original and enriched models.

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

Table 7: Comparison between the original AC (Original) and the AC derived from the enriched model (Augmented).

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

Table 8: Accuracy metrics for our recommendations.

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

5.4 Analysis of Collected Data

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

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

We calculate precision as $TP/(TP + FP)$ and recall as $TP/(TP + FN)$. Table 8 shows our accuracy results: over our case study, the recommendations have a precision of 89% and recall of 100%. There are three FPs, resulting from three recommendations that suggested missing elements in the model, but these elements were identified to be already present by the analysts. Our approach did not yield any FNs due to the systematic derivation of our extraction rules (Section 4.1). Nevertheless, the analysts were asked to try to identify any recommendations that our approach might have missed; they could not identify any.

To answer Q2, we considered only the 24 recommendations that were deemed correct and relevant by analysts (Q2 in Table 5). The analysts found the additional information resulting from 17 of these recommendations not worth visualizing in the model. For example, the analysts agreed that details such as the expected runtime property values extracted by extraction rule `SR2` (Table 3) did not need to be visualized. The analysts agreed that the additional information resulting from the seven remaining recommendations should be visualized as the information was deemed helpful for improving model comprehension. For example, the analysts found the conditions extracted from NL requirements by rule `C4` (Table 3) to be useful information to visualize in decision nodes. To conclude, most of the information extracted from NL requirements, despite being relevant to the generation of AC, was deemed not useful for visualization purposes. This observation provides evidence about the largely complementary nature of the information content captured by models versus NL requirements.

6 THREATS TO VALIDITY

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

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

In our case study, the `While` and `Where` structures – two out of Rimay's eight main grammar rules (see Listings 1 and 2) – were left unused by the NL requirements. Nevertheless, these structures would be treated similarly to the `If` structure (which does get used in our case study). As for the AC, the Gherkin scenarios derived from the enriched model in our case study cover six out of the 11 intent types supported by AGAC. The covered intent types are: "delete", "send", "receive", "update", and "validate". The intent types not covered are "read", "enable", "disable", "display" and "not display". The template for "read" is conceptually similar to that for "send". For example, the post-condition for "read" ("Then [actor] read [object]") is similar to that for "send" ("Then [actor] sent [object]"). The templates for the other four intent types ("enable", "disable", "display" and "not display") are similar to that for "delete". For example, the pre-condition for all these four intent types is "Given [object] exists in [actor]", just like for "delete". We thus anticipate that our results will generalize to other systems of the same type. This being said, further case studies involving other types of systems that are commonly modeled using ADs, CDs and UCDs, e.g., public administration services, remain necessary for improving external validity.

7 RELATED WORK

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

We consider eight studies relevant to our work; seven aim at extracting information from NL requirements to create UML models and one – to enhance existing UML models for the purpose of test-case generation. Table 9 outlines these eight studies. The first column cites the study. The second column indicates whether an empirical evaluation was conducted as part of the study. The third column shows the number of rules, patterns, or heuristics that are included in the study. Finally, the fourth column indicates whether the study uses CNLs, patterns, or templates.

The approaches described in [5, 6, 9, 13] were identified from the systematic review of transformation approaches between user requirements and analysis models conducted by Yue et al. [18]. We further included subsequent studies to this systematic review [3, 14, 16, 20].

We start with the studies that extract information from NL requirements for the creation of UML models. Ilieva et al. [6] propose

Table 9: Summary of related work.

Study Reference	Empirical Evaluation	Number of Rules	Restricted NL?
Ilieva et al. [6]	No	14	No
Moreno et al. [9]	No	9	No
Fliedl et al. [5]	No	5	No
Smialek et al. [13]	No	6	Yes
Arora et al. [3]	Yes	21	No
Thakur et al. [14]	Yes	54	No
Yue et al. [20]	Yes	65	Yes
Wang et al. [16]	Yes	N/A	Yes

a methodology to extract information from unrestricted NL requirements to build UCDs, ADs, and domain models. To this end, they describe 14 heuristics that define the correspondence between UML model elements and pieces of information in the NL requirements. Moreno et al. [9] discuss nine patterns for structuring the information representing the data with which the system works (static information) and the data that describes the system behavior (dynamic information) as extracted from unrestricted NL requirements. In this study, static and dynamic information is used, respectively, for building conceptual and behavioral models.

Fliedl et al. [5] propose a semi-automated approach that considers the user's feedback to linguistically analyze unrestricted NL requirements and translate these requirements into a conceptual pre-design schema. This schema contains static and dynamic information. The study maps this information to UML model diagrams, e.g, ADs, using a set of five extraction rules.

Smialek et al. [13] restrict the representation of UCDs through a concrete syntax. They describe the meta-model of their concrete syntax and define mapping patterns that link the concrete syntax meta-model components to the UML elements of Sequence Diagrams and ADs. They describe six mapping patterns in the study. None of the above approaches report empirical evaluations.

Arora et al. [3] collect from previous work a set of 18 extraction rules and propose three new extraction rules for unrestricted NL requirements. Their rules extract elements for building domain models. The authors evaluate the usefulness of their approach via a case study and expert surveys.

Thakur et al. [14] report an automated approach to extract domain elements from unrestricted UCD specifications. They propose 54 extraction rules and empirically compare their approach with two similar approaches.

Yue et al. [20] propose an approach that, by implementing 65 transformation rules (28 rules for CDs, 18 rules for sequence diagrams, and 19 rules for ADs), automatically generates UML model diagrams from NL requirements. They consider NL requirements that conform to a restricted natural language, named RUCM [19], for writing UCD specifications. They empirically evaluate their approach in terms of consistency, completeness, applicability, and performance through a series of case studies.

All the above-mentioned studies target the construction of UML models from scratch. In contrast, the solution proposed by Wang et al. [16] updates UML models using NL requirements to generate test cases. Their approach enables users, based on UCD specifications written in RUCM [19], to extract model elements for checking completeness and add new information to models, e.g., conditional statements to be inserted as pre- and post-conditions. Additionally, their approach creates a test model for the generation of test cases. An empirical evaluation reported that their approach generates 25% more test case scenarios compared to manually written test case scenarios developed by experts.

To summarize, seven studies [3, 5, 6, 9, 13, 14, 20] discussed in this section suggest rules to fully transform textual requirements to models. One of our rules ($C4$), presented in Table 3, shares the same rationale as a rule introduced by Ilieva and Ormandjieva [6]. The rest of our rules differ from the rules, heuristics, and patterns introduced in the above-cited studies. One approach [16] includes information from NL requirements to enhance conceptual models and generate test cases. In contrast, our approach identifies AC-related information from NL requirements which, if included in the model, can significantly improve the precision and completeness of generated AC in terms of Gherkin scenarios. Moreover, our approach differs from that of Wang et al. in the nature of the NL requirements. Whereas Wang et al. orient their work around textual use-case specifications resulting from object-oriented analysis, the CNL that underlies our work, Rimay, has been designed around more traditional requirements engineering practices where the requirements are expressed as independent statements with modal verbs (e.g., shall, must, will).

8 CONCLUSIONS

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

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

For future work, we are planning to enhance our approach to include semantic analysis, in addition to syntactic comparisons between NL requirements and models.

ACKNOWLEDGMENTS

This paper was supported by Clearstream, Luxembourg's National Research Fund (FNR) under grant BRIDGES18/IS/13234469/IMoReF and NSERC of Canada under the Discovery, Discovery Accelerator and CRC programs.

REFERENCES

[1] Mauricio Alférez, Fabrizio Pastore, Mehrdad Sabetzadeh, Lionel C. Briand, and Jean-Richard Riccardi. 2019. Bridging the Gap between Requirements Modeling and Behavior-Driven Development. In *MoDELS*. IEEE, 239–249.

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.

[3] Chetan Arora, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2016. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, Benoit Baudry and Benoît Combemale (Eds.). ACM, 250–260. http://dl.acm.org/citation.cfm?id=2976769

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

[5] Günther Fliedl, Christian Kop, Heinrich C. Mayr, Alexander Salbrechter, Jürgen Vöhringer, Georg Weber, and Christian Winkler. 2007. Deriving static and dynamic concepts from software requirements using sophisticated tagging. *Data Knowl. Eng.* 61, 3 (2007), 433–448. https://doi.org/10.1016/j.datak.2006.06.012

[6] M. G. Ilieva and Olga Ormandjieva. 2006. Models Derived from Automatically Analyzed Textual User Requirements. In *Fourth International Conference on Software Engineering, Research, Management and Applications (SERA 2006), 9-11 August 2006, Seattle, Washington, USA*. IEEE Computer Society, 13–21. https://doi.org/10.1109/SERA.2006.51

[7] Alistair Mavin, Philip Wilkinson, Sarah Gregory, and Eero Uusitalo. 2016. Listens Learned (8 Lessons Learned Applying EARS). In *RE*. IEEE Computer Society, 276–282.

[8] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. Easy Approach to Requirements Syntax (EARS). In *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*. IEEE Computer Society, 317–322. https://doi.org/10.1109/RE.2009.9

[9] Ana M. Moreno-Capuchino, Natalia Juristo Juzgado, and Reind P. van de Riet. 2000. Formal justification in object-oriented modelling: A linguistic approach. *Data Knowl. Eng.* 33, 1 (2000), 25–47. https://doi.org/10.1016/S0169-023X(99)00046-4

[10] OMG. 2017. Unified Modeling Language. Version 2.5.1. https://www.omg.org/spec/UML/

[11] Klaus Pohl. 2010. Requirements engineering : fundamentals, principles, and techniques.

[12] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples* (1st ed.). Wiley Publishing.

[13] Michal Smialek, Jacek Bojarski, Wiktor Nowakowski, Albert Ambroziewicz, and Tomasz Straszak. 2007. Complementary Use Case Scenario Representations Based on Domain Vocabularies. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4735)*, Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil (Eds.). Springer, 544–558. https://doi.org/10.1007/978-3-540-75209-7_37

[14] Jitendra Singh Thakur and Atul Gupta. 2016. Identifying domain elements from textual specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 566–577. https://doi.org/10.1145/2970276.2970323

[15] Alvaro Veizaga, Mauricio Alferez, Damiano Torre, Mehrdad Sabetzadeh, and Lionel Briand. 2020. On Systematically Building a Controlled Natural Language for Functional Requirements. *arXiv preprint arXiv:2005.01355* (2020).

[16] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel C. Briand, and Muhammad Zohaib Z. Iqbal. 2015. UMTG: a toolset to automatically generate system test cases from use case specifications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 942–945. https://doi.org/10.1145/2786805.2803187

[17] Matt Wynne and Aslak Hellesoy. 2017. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers.* Pragmatic Bookshelf.

[18] Tao Yue, Lionel C. Briand, and Yvan Labiche. 2011. A systematic review of transformation approaches between user requirements and analysis models. *Requir. Eng.* 16, 2 (2011), 75–99. https://doi.org/10.1007/s00766-010-0111-y

[19] Tao Yue, Lionel C. Briand, and Yvan Labiche. 2013. Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Trans. Softw. Eng. Methodol.* 22, 1 (2013), 5:1–5:38. https://doi.org/10.1145/2430536.2430539

[20] Tao Yue, Lionel C. Briand, and Yvan Labiche. 2015. aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models. *ACM Trans. Softw. Eng. Methodol.* 24, 3 (2015), 13:1–13:52. https://doi.org/10.1145/2699697