A Natural Language Processing Technique for Formalization of Systems Requirement Specifications

Viktoria Koscinski*, Celeste Gambardella*, Estey Gerstner*,
Mark Zappavigna†, Jennifer Cassetti† and Mehdi Mirakhorli*
*Rochester Institute of Technology, Rochester, NY 14623–5603
Email: {vk2635, cg7346, ewgvse, mxmvse}@rit.edu,
†Air Force Research Laboratory, Rome, NY 13441–4514
Email: {mark.zappavigna, jennifer.cassetti.1}@us.af.mil

***Abstract*—Natural language processing techniques have proven to be useful for analysis of technical specifications documents. One such technique, information extraction (IE), can help automate the analysis of software systems requirement specifications (SysRS) by extracting structured information from unstructured or semi-structured natural language data, allowing for requirements to be converted into formal logic. Current IE techniques are not designed for SysRS data, and often do not extract the information needed for requirements formalization. In this work, we introduce an IE method specifically designed for SysRS data. We provide a description of our approach, analysis of the technique on a set of real requirements, example of how information obtained using our technique can be converted into a formal logic representation, and discussion of our technique and its benefits in automated SysRS analysis tasks.**

***Index Terms*—natural language processing, information extraction, requirements formalization, requirements analysis**

I. Introduction

Natural language processing (NLP) has increasingly been used to solve software requirements engineering problems [8], [16]. Developing a successful software system starts with requirements engineering - eliciting, documenting and analyzing the needs of users and required features of the software to be built [14]. These requirements are typically written using natural language (NL), describing capabilities, functions and constraints of the software product. *Software system requirements specifications (SysRS)* contain information about how the software will work and interact with the user. SysRS are analyzed by requirements engineers, domain experts and business analysts to identify deficient, conflicting, or underspecified requirements before the software is developed. NLP empowers requirements engineers by automating cumbersome and repetitive tasks [5], reducing the time and costs associated with designing a software system. It can facilitate detecting SysRS inconsistencies [3], detecting SysRS ambiguities [9], and structuring SysRS [2], among other analysis tasks [16].

Automatically analyzing SysRS is challenging, as they are typically expressed using a textual NL representation rather than a formal representation. Many analysis tasks require SysRS to be expressed in formal representations such as predicate logic [10], temporal logic [3], [4], executable code [11], or other formal representations. To facilitate the formalization of requirements, NLP techniques obtain information such as their part-of-speech (PoS) tags and grammatical dependencies. Requirements engineers are then tasked with using this information to formalize SysRS, often by creating a set of rules based on patterns found within the information obtained.

Among NLP techniques, information extraction (IE) can extract structured information from unstructured requirements. This can help conduct SysRS formalization by automatically assigning meaningful high-level information, which can then be used instead of (or in conjunction with) PoS tags or grammatical dependencies to create formalization rules. While promising, IE tools were not designed with SysRS data in mind [13]. SysRS are technical with specific structure and terminology that differs from NL commonly used on the web. Existing IE techniques typically structure information into triples containing the subject, relation, and object of a sentence. SysRS often contain detailed information that an information triple cannot adequately express for formalization. For example, "*Hosting Service Provider will comply with HIPAA and HITECH security controls for relevant health care-based components*," requires the identification of details that describe not only what the security controls are for (components), but also what kind of components they are for (relevant, health care-based). Since neither of these details can be obtained using only a relation triple, an IE method that is better-suited towards SysRS data is needed.

In this paper, we introduce a novel IE method tailored to analyzing SysRS data. It uses six information categories and helps obtain the information needed to formalize SysRS. We describe the approach and its advantages for SysRS data over traditional IE tools. We analyze our approach on a set of real requirements used in the development of the Vermont Health Care Uniform Reporting and Evaluation System (*VHCURES*), a software program built by the Government of Vermont. We then demonstrate the practicality of our approach by showing

This project was partially supported by the Air Force Research Laboratory (AFRL), Rome, NY under Contract #: FA8750-19-C-0524.

how it can aid the task of formal requirements elicitation from a NL textual representation. Finally, we provide a discussion of our results, our technique's capabilities, and plans to improve and expand our technique.

II. Background

In order to conduct SysRS analysis, data is processed using NLP techniques such as IE. Typically, information extracted by IE is structured by categorizing it into a *relation tuple*, usually describing the *subject*, *relation*, and *object* of the sentence. For example, "*User account passwords shall include a minimum of two different types of characters*" may be expressed as the relation tuple "(*user account passwords*; *shall include*; *a minimum of two different types of characters*)." Relation tuples aim to express the meaning of the original NL statement at a high-level structured manner and can be used to formalize the data for further analysis such as automated reasoning.

IE tools use patterns, also known as *rules*, that are matched in sentences' grammatical dependencies and/or PoS tags obtained using a parser. Parsing, or *syntactic analysis*, relies on probabilistic methods to analyze words of text based on an underlying grammar. Parsers can describe how individual words of a sentence are related or group words into noun or verb phrases (see Fig. 2). Patterns used in IE are either handwritten for *rule-based IE* or learned using labeled training data for *learning-based IE* [13]. We briefly describe three well-known IE techniques and their application to SysRS analysis, as well as their limitations when applied to SysRS, in the following subsections.

• **Open IE** extracts information triples representing binary relations using a set of 14 hand-written patterns. To aid the extraction, a classifier traverses a sentence's parse tree and separates the sentence into shorter independent clauses to which the rules are applied, yielding the subject, relation, and object [1]. **Observation #1:** This technique lacks support for identifying and scoping *negative relations*. Because SysRS commonly have negative clauses ("*not* X") there is a significant amount of information that Open IE cannot extract from SysRS. **Observation #2:** Open IE lacks support for *conditional phrases* ("*if* Y, *then* Z"), which are commonly used in SysRS to describe expected behaviour of software. **Observation #3:** Open IE is also unable to correctly process *contractually binding* requirements ("X *shall be able*") which are common among SysRS.

• **ReVerb** is a rule-based IE approach that aims to reduce the number of incoherent relation tuples by using simple PoS-based regular expressions [7]. It uses a relation-based approach in which the relation is first identified, followed by its corresponding arguments. ReVerb can output *normalized* triples, which contain the lemmatized version of words. For example, "*user account passwords shall expire after 12 months*" would yield the triple "(*user account passwords*; *expire after*; *# months*)." **Observation #4:** While ReVerb extracts more information from our SysRS dataset than Open IE, it still is unable to identify clauses, and does not include negations in its normalized (lemmatized) extractions.

• **Ollie** is a learning-based successor of ReVerb, providing interpretations of more complex sentences that contain clauses describing information that may be only hypothetically or conditionally true [15]. Analyzing "*the system shall notify the administrator if malware is detected during a scan*," yields "(*The system*; *shall notify*; *the Administrator*)[*enabler=if malware is detected during a scan*]." The *enabler* field is a fourth (optional) extracted information element. Ollie uses a set of high-confidence extractions from ReVerb to bootstrap a training set by searching a web corpus for phrases with the words in the seed tuple. Pattern templates learned from the training set are applied to a sentence's parse tree for extraction. **Observation #5:** Since Ollie is trained on web data, it still has trouble correctly extracting information from common phrases in SysRS, such as *contractually binding* requirements or *interface* requirements captured using the "*able to <process>*" phrase, which states a functionality of the system in reaction to trigger events [2].

• **ClausIE** is a rule based approach that exploits linguistic knowledge about English grammar to detect and identify types of clauses in an input sentence [6]. ClausIE is able to generate high-precision n-tuple clause extractions, which it then concatenates to make *propositions* (relation triples). **Observation #6:** While ClausIE yields multiple relation triples, they do not capture context. For example, if a *conditional phrase* such as "*if* event X, *then* event Y" is present, ClausIE would capture both *event X* and *event Y*, but not their conditionality (an important component of many requirements). This results in capturing *non-factual propositions*, which do not accurately represent the information expressed. **Observation #7:** Because ClausIE's relation propositions often contain multiple concatenated clauses, the *subject*, *relation*, or *object* may be over-specified, meaning that the information within is too coarse-grained to convert to a formal representation without further processing or customization of the extraction tool.

III. Approach

Our custom approach is built using a Stanford CoreNLP pipeline [12]. Using the dependency parse patterns of a NL sentence, we extract information into *six categories*. As shown in Fig. 1, categories are the *subject descriptor*, *subject*, *relation*, *case* (such as prepositions and postpositions), *object descriptor*, and *object*. Each of these categories are optional; only relevant categories are filled. We include **six categories** instead of three because SysRS data often contains important details that need to be formalized for analysis.

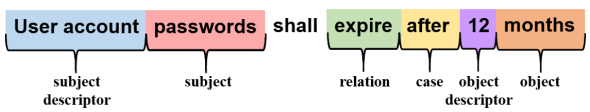

Fig. 1. Information extraction on a security-related requirement is conducted by categorizing the sentence based on patterns within its dependency parse. Common relation modifiers used in SysRS such as "shall" are not explicitly categorized.

In order to detect the correct words for each category, we first input each SysRS as a string with one sentence per line.

TABLE I

PATTERNS DEVELOPED TO EXTRACT OUR SIX MAIN INFORMATION CATEGORIES. PSEUDO-EXPRESSION SYNTAX IS DEFINED AS FOLLOWS: {R}, {S}, AND {O} DEFINE THE MAIN RELATION (ROOT), SUBJECT, AND OBJECT (AS OPPOSED TO THE GRAMMATICAL DEPENDENCY "*obj*"), RESPECTIVELY. IN CURLY BRACKETS, LOWERCASE TERMS DEFINE GRAMMATICAL DEPENDENCIES AND UPPERCASE TERMS DEFINE PARTS-OF-SPEECH. AN ARROW CONNECTING TWO TERMS DEFINES A {PARENT NODE} -> {CHILD NODE} RELATIONSHIP.

Rule Category	Pseudo-Expression	Descriptive Summary
Subject	{R}->{nsubj} OR {R}->{nsubjpass}	Child of root and (NOMINAL_SUBJECT or NOMINAL_PASSIVE_SUBJECT relation)
Relation	{R}	Root
Object	{R}->{nmod} OR {R}->{obj}	Child of root and (NOMINAL_MODIFIER or DIRECT_OBJECT relation)
Descriptor	{S}->{amod} OR {O}->{amod} OR {S}->{nummod} OR {O}->{nummod} OR {S}->{compound} OR {O}->{compound}	(Child of subject or object) and (ADJECTIVAL_MODIFIER, COMPOUND_MODIFIER, or NUMERIC_MODIFIER relation)
Case	{O}->{case}	Child of object and CASE_MARKER relation
Negation	{VERB}->{neg}	Child of VERB and NEGATION_MODIFIER relation

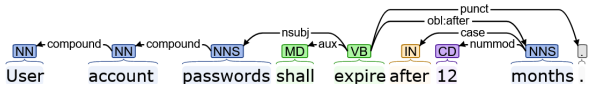

Fig. 2. Grammatical dependencies and part-of-speech tags for a sentence. For example, plural noun (NNS) "passwords" is the nominal subject (nsubj) of the verb (VB) "expire."

For each sentence, we obtain the parse tree, which contains all of the grammatical dependencies and PoS tags of a sentence, as shown in Fig. 2. Finally, we use our custom rules and conduct a pattern search based on the words' dependencies. Our current list of patterns are shown in Table I. Each tree is traversed by starting from a chosen node and checking all of that node's children for specified dependencies. One rule exists for each category (or attribute) we are looking to fill. Once a sentence in SysRS is checked, our technique automatically moves on to the next, until IE is conducted on all of the SysRS.

Descriptors allow us to separate a subject's or object's description from other parts of the requirement. This reduces formalization ambiguities by ensuring the same type of information is kept in the same category. Furthermore, it allows us to give formalized elements measurable properties. For example, we can differentiate types of *attempts* by formalizing them as an element with possible states (in this case *unsuccessful* or *successful*) or types (such as *login* or *transaction*).

Cases allow us to more accurately describe what is happening in a system. For example, we can formally model whether passwords expire *after* 12 months or *for* 12 months. Again, this allows us to use simple relations such as *expire* rather than having separate relations for *expire after*, *expire on*, and *expire for*, which risks missing the case's relation to the object.

Information Extraction from Enablers in SysRS: Other than six information categories, we are able to extract certain clauses, such as the conditional clause (*if*), to-infinitive clause (*to*), and result clause (*so*), to recognize their function in a sentence. Similar to Ollie's *enabler*, this can describe features such as causality. Unlike Ollie, which keeps the whole *enabler* phrase as one "chunk" (without categorizing information within), our clausal phrase is split into its corresponding categories just like the "main" part of the sentence.

Support of Negations in SysRS: We have implemented the detection of verb and noun negation, an essential component of SysRS data. At its core, negations are detected by finding words with a *negation modifier* relation. However, we have strengthened our technique's negation detection abilities by detecting multiple negated words in a sentence using certain *conjunct* relations. For example, in the phrase, "*the system shall be deployed with no viruses or malware*," we detect that both the words "*viruses*" and "*malware*" are negated. Our negation detection functionality extends to lists of negated items, which we will elaborate upon in the next paragraph.

Support of Listed items in SysRS: While most well-written requirements are simple in structure, it is common for requirements to contain lists. Often, lists are expressed using a series of bullet points or a series of items following an independent clause and preceded by a colon (e.g., "*log entries capture the following information: date, time, ...*"). Many parsers are designed to work with full sentences, and as a result misinterpret each bullet point as a complete sentence. When items follow an independent clause, the list is often missed completely. One way to process bullet points and other lists is to convert them into a set of listed items related directly to a verb or preposition, as in the example: "*The system shall encrypt credit card information, SSNs, PINs, passwords, ID photos, financial data, health records, geographic location, contact details, and personal messages.*" We found that while parsers can often handle such a sentence, IE techniques tend to overlook such cases and lose data contained in the sentence. For example, Ollie's extraction for the above example is "(*The system*; *shall encrypt*; *credit card information*)." Not only does our technique extract each listed item, but it also extracts and correctly assigns descriptors (and negations, if they apply) to each item. Our technique detects **subjects**, **objects**, and **descriptors** in a listed format. While we have successfully tested up to 20 listed items, we are confident in our technique's ability to correctly identify even more items within a requirement, although this occurrence is uncommon in a real-world scenario.

IV. CASE STUDY OF HEALTH CARE UNIFORM REPORTING AND EVALUATION SYSTEM

The proposed approach has been evaluated on a real SysRS document. We used NL requirements from the VHCURES

3.0 Software—Government of Vermont's Health Care Uniform Reporting and Evaluation System[1]. We first created a manually-labeled ground truth dataset made up of 33 well-written functional and nonfunctional requirements from the VHCURES 3.0 document. This was done by indicating which of our six categories each word should map to, as well as whether or not a word is negated or part of the enabler clause. A *well-written* requirement is one that is direct, unambiguous, and not overly complex. While requirements based on templates would likely yield better results [2], they are also more restrictive and not always used. We chose well-written requirements for our dataset based on the following criteria:

- Requirements must have a situation (subject), action (verb), corresponding action ("*must*," "*shall*," etc.), and object, unless it is of the form "X *is* Y."
- Requirements must avoid passive phrases that modify the verb. For example, phrases such as "*be able to* X," "*will have* X *done to it*," and "*agree to do* X" are avoided.
- Requirements must have a maximum of two clauses that contain their own subject(s), relation, and object(s).
- Listed items must use direct language (not bullet points or an independent clause with a colon).
- Requirements must be both grammatically correct and unambiguous. For example, "X *and* Y *or* Z" is an ambiguous phrase.
- Requirements must avoid second-person phrases.
- Requirements must avoid overly complex subjects. For example, "*logs with relevant information shall*" contains extra information about logs that neither belongs to the subject nor the object category.

After creating the ground truth dataset, we obtained the categorized information elements from each of these 33 requirements using our custom IE technique. Our analysis is based on the percentage of words in each requirement that is correctly categorized into the six information categories and properly identified as an enabler or negated term (if applicable) according to our ground truth data. Words that are not relevant or directly extracted into our categories are removed and not counted towards our analysis. These words include determiners (such as "*the*" and "*a*"), acronyms of phrases (in parentheses), as well as modal verbs which are standard language used in requirements and not directly translated into logic (such as "*shall*" and "*will*"). Remaining words are known as *relevant information*. An example of our analysis is shown in Fig. 3.

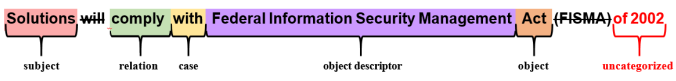

Fig. 3. Finding the percent of correctly identified words in a security requirement. Two irrelevant words are removed. Since two words ("of 2002") are not classified, this sentence contains 80% (8/10 words) words correctly identified.

After analyzing our results by comparing our method's output to the manually-labeled ground truth data, we found that our technique is able to correctly extract and categorize 85.75% of the information contained in each requirement on average. A breakdown of our results is shown in Table II.

TABLE II
A MORE DETAILED BREAKDOWN OF OUR IE TECHNIQUE'S RESULTS FOR OUR 33-REQUIREMENT DATASET FROM THE VHCURES 3.0 PROJECT.

Percent (%) relevant information extracted	Number of requirements
100.00	17
90.00-99.99	1
80.00-89.99	6
70.00-79.99	1
60.00-69.99	4
50.00-59.99	3
40.00-49.99	1

Out of the 16 results that did not achieve 100% correctly extracted relevant information, we determined that nine of these extractions had missing information, four had miscategorized information, and three (not included in the previous two categories) had both missing and miscategorized information. Each of these 16 requirements fell into one or more of four classes describing what they missed/miscategorized. We describe and provide examples of these classes as follows:

Miscategorized noun descriptor: When a noun acts as a descriptor, it can be miscategorized as an object. This is more likely to occur when the noun is being used in a conjunction ("*and*" or "*or*") with another descriptor. For example, in the phrase "*dictionary and brute force attacks*," the noun "*dictionary*" should be a descriptor of the object "*attacks*" (like "*brute force*") but is instead categorized as its own object. Additional factors that may contribute to this miscategorization are the noun being a proper noun or an acronym. There were six requirements in this category.

Missing case and object: Cases may be expressed differently in a parse tree, depending on the sentence's structure. Because of this, the case follows a different pattern than what we search for. This may be mitigated by further investigation and potentially implementing additional rules to find the case. In our dataset, missing cases would answer the questions *within where?*, *of what?* (e.g., "*protection of data*"), *for the purpose of doing what?*, and *to whom?*. There were eight requirements in this category.

Missing determiner descriptor: Since most determiners such as "*a*," "*an*," and "*the*" are irrelevant, we do not categorize determiners. Descriptive determiners such as "*some*" or "*all*" are also not categorized. Depending on the use case, these determiners may or may not be important to categorize, which is why we considered them to be missing information. There were two requirements in this category.

Proper noun miscategorization: When a proper noun phrase contains a word that may normally function as a different category (such as a case or a relation), this word may be miscategorized into that category. For example, "*single sign on components*" may mistakenly categorize "*on*" as a case rather than a descriptor, resulting in the incorrect meaning, "*[a] single sign on [top of] components*." This may be mitigated

[1]https://gmcboard.vermont.gov/content/RFP-Attachment1-functional-and-nonfunctional-requirements-vhcures-30-rfp

by defining technical terms before parsing. We only have one requirement in this category.

V. Generating Formal Representations from Textually-Represented Requirements

In order to automatically conduct formal reasoning on SysRS documents, processed requirements from the documents need to be formalized. Requirements are formalized in representations such as predicate logic [10], temporal logic [3], [4], and even executable code [11]. In this section, we provide a walkthrough and case study of how our IE results can be used to formalize processed requirements into logic formulas. Our walkthrough and case study is based on converting NL requirements to formal logic similar to that in [10], which contains requirements for a computer-aided dispatch system of the London Ambulance Service (LAS).

Logic formulas similar to those in [10] contain operators such as (but not limited to) `AND`, `OR`, `NOT`, and `IMPLICATION`, as well as *predicates* describing the system, such as "have(ambulance, crew)" or "available(ambulance)." Using our six information categories and two additional elements we detect, we demonstrate the advantages of using our information categories to formalize processed requirements into formally represented predicates and logic operators. Although they are described in Section III, we list each of our categories and additional elements below for convenience:

- Subject – one or more assigned to the relation
- Subject descriptor – one or more assigned to each subject
- Relation -– the "main verb" of the sentence or clause
- Object — one or more assigned to the relation
- Object descriptor — one or more assigned to each object
- Case — assigned to the object
- Enabler (adverbial clause modifier) – detected in sentences with two clauses; each clause can have the main six categories
- Negation – assigned to a subject, object, or relation

Logic formulas are obtained by applying a set of rules to processed requirements. Instead of creating rules based on patterns directly in a sentence's PoS tags and grammatical relations [10], we demonstrate example rules based on entities obtained using our IE technique in Table III. This non-comprehensive list of rules shows that rules vary in complexity, and that multiple rules may yield the same logic formula.

A. Walkthrough Example

We provide step-by-step examples of converting NL requirements from a textual representation into a formal representation using our extracted information elements and example rules from Table III. Our examples are based on requirements for a computer-aided dispatch system of the LAS [10] and show how using our extracted information elements is beneficial in formalizing NL requirements.

Example 1: "*If an ambulance does not have a crew, the ambulance is not available*."

TABLE III
Example rules for converting our extracted information categories into formal logic formulas.

Binary relation: P(x, y)	
Rule (a)	[if no case present] relation(subject, object)
Rule (b)	relation_case(subject, object)
Unary relation: P(x)	
Rule (c)	[if no object present] relation(subject)
Rule (d)	[if no subject present] relation_case(object)
Rule (e)	subject(subject_descriptor)
Rule (f)	object(object_descriptor)
Negated relation: !P( )	
Rule (g)	[if relation negated] !relation( )
Rule (h)	[if object negated] !relation( )
Rule (i)	[if subject negated] !relation( )
Conditional relation: P( ) –> Q( )	
Rule (j)	[if enabler present] {enabler clause}–>{regular clause}

1) We input this requirement into our custom IE tool, yielding the following output (categories with no extractions are not listed to conserve space):
   Relation: *available* [*negated*]
   Subject: *ambulance*
   Enabler (adverbial clause modifier): *if*
   Enabler relation: *have* [*negated*]
   Enabler subject: *ambulance*
   Enabler object: *crew*
2) Because an enabler is present, we apply Rule (j) and split our sentence into two clauses:
   `{enabler clause} -> {regular clause}`
3) The enabler clause contains a subject, object, and relation, yielding a binary relation according to Rule (a):
   `have(ambulance, crew) -> {regular clause}`
4) Since the relation is negated, Rule (g) results in:
   `!have(ambulance, crew) -> {regular clause}`
5) Since there are no more rules to apply to the enabler clause, the main clause is investigated. Rule (c) is applied since no object is present:
   `!have(ambulance, crew) -> available(ambulance)`
6) Since the relation is negated, Rule (g) results in:
   `!have(ambulance, crew) -> !available(ambulance)`

Our final logic formula is !have(ambulance, crew) –> !available(ambulance). This formal representation is the same as the logic formula provided for this requirement in [10]. This example has shows how just a few simple rules need to be applied based on our extracted information categories to automatically create formal logic statements from NL requirements.

Example 2: "*If an ambulance was not serviced during the last year, then the ambulance is not available*."

1) Inputting this sentence into our IE technique yields the following output:
   Relation: *available* [*negated*]
   Subject: *ambulance*
   Enabler (adverbial clause modifier): *if*
   Enabler relation: *serviced* [*negated*]
   Enabler subject: *ambulance*
   Enabler object descriptor: *last*

Enabler case: *during*
Enabler object: *year*

2) Because an enabler is present, we apply Rule (j) and split our sentence into two clauses:
`{enabler clause} -> {regular clause}`
3) The enabler clause contains a subject, object, case, and relation, so we apply Rule (b):
`service_during(ambulance, year) -> {regular clause}`
4) Because the object has a descriptor, Rule (f) is applied:
`service_during(ambulance, year(last)) -> {regular clause}`
5) Since the relation is negated, Rule (g) results in:
`!service_during(ambulance, year(last)) -> {regular clause}`
6) Since there are no more rules to apply to the enabler clause, the main clause is investigated. Rule (c) is applied since no object is present:
`!service_during(ambulance, year(last)) -> available(ambulance)`
7) Since the relation is negated, rule (g) is applied:
`!service_during(ambulance, year(last)) -> !available(ambulance)`

Our logic formula, `!service_during(ambulance, year(last)) -> !available(ambulance)`, differs from that in [10], `!last(year) -> (!revise(ambulance, year) -> !available(ambulance))`. We argue that our formal representation of this requirement becomes more clear with the use of our extracted *case* element. Besides obtaining an intuitive and descriptive relation (*service_during*), our use of cases eliminates the need for creating a relation not described in the NL text (*revise*) to explain a more complex concept such as time. Additionally, through our effective use of descriptors, we obtain the predicate *year(last)* rather than *last(year)*. While at first glance this may appear insignificant, it is useful in conducting formal reasoning. With our representation, we are able to easily find each instance of *year* and see what properties apply to it in each instance (such as: *last* year, *one* year, *next* year, etc.). Most importantly, we avoid confusion arising from expressing similar relations that may represent different concepts. For example, something may occur *last year*, something may *last a year*, or something may have occurred *the last few years*. Expressing the first concept as *last(year)* may contain ambiguity as to which idea is being expressed, especially if multiple ideas are present. In our case, the representation for each of these ideas is simple and intuitive: *year(last)*, *last(something, year)*, and *occur(something, year(last few))*, respectively.

B. Case Study

We provide a case study by applying our formalization rules to the extracted information elements from our real SysRS dataset of 33 requirements from the VHCURES 3.0 document. Table IV shows examples of our IE categories and formal logic representations automatically obtained from NL text requirements. These examples show the advantages of fine-grained information categories such as descriptors and cases when applied to a set of real requirements. We focus on these two information entities, as they are especially unique compared to other works. We also describe the advantages of all of our information entities as a whole in section VI.

The importance of extracting the *case* within a sentence or clause is shown in all three examples, but especially in example 1. In this example, there are two separate cases extracted within the sentence, both of which apply to the same relation. This yields two relations, *respond_to* and *respond_within*, rather than a single relation *respond*. The main advantage to this is that our relations are unambiguous, since *respond* can refer to concepts such as *responding to someone*, *responding within a timeframe*, or *responding with an answer*. We also automatically generate these formalizations, rather than making separate formalization rules based on keywords such as *within*, *to*, or *with*.

The advantage of descriptors is shown in example 2. In this example, it is clear that both the *infrastructure* and *software* in belong to the *hosting service provider*. We automatically assign these descriptors to subjects/objects efficiently and unambiguously. Within requirements, it is often important to differentiate between same objects of different types (such as *service provider software* vs *third party software*) or within different objects of the same type (such as *service provider infrastructure* vs *service provider software*). Automatically formalizing these types of differences using our *descriptor* category not only helps express ideas precisely for automatic reasoning, but it also helps requirements engineers to have an unambiguous representation when these statements are manually reviewed.

We provide additional results describing the performance of the formalization technique. Out of the 33 requirements within the dataset, 23 of the requirements resulted in a correct formal representation that can be used for reasoning, even though only 17 requirements perfectly extracted all information into the correct categories. 3 requirements successfully formalized one or more logic statements but did not formally represent all of the intended information, while 7 incorrectly expressed the logic formulas. All of the requirements that had missed or incorrectly formalized any information were ones on which our custom IE technique missed or miscategorized information, and we are confident that our results will improve as we continue to extend our IE technique. While these results are preliminary, they show promise in the use of our technique for requirements formalization.

VI. Discussion

In this work, we presented an IE technique that obtains information that can be used to formalize NL SysRS data. Using our IE technique, as opposed to using direct grammatical dependencies or PoS tags, to write formalization rules allows requirements engineers to automatically formalize information-rich NL requirements without needing to write a great amount of highly-specific, and possibly redundant, rules. An advantage of our technique is that it is possible to generate

TABLE IV

EXAMPLES OF REAL REQUIREMENTS CONVERTED INTO A FORMAL LOGIC REPRESENTATION BY USING OUR CUSTOM INFORMATION CATEGORIES. EACH WORD IS TAGGED WITH ZERO OR MORE TAGS REPRESENTING THE INFORMATION ELEMENT AS FOLLOWS: SUBJECT DESCRIPTOR [SD], SUBJECT [S], RELATION [R], OBJECT DESCRIPTOR [OD], OBJECT [O], CASE [C], AND NEGATION [N]. THE LOGIC REPRESENTATION IS OBTAINED THROUGH OUR TECHNIQUE THAT AUTOMATICALLY APPLIES A SET OF RULES (SUCH AS THOSE IN TABLE III) TO THE INFORMATION ELEMENTS.

#	Requirement Tagged with Extracted Information Categories	Formalized Logic Representation
1	Vendor[S] shall respond[R] to[C] all requests/questions[O] within[C] 24[OD] hours[O].	respond_to(Vendor, requests/questions) `AND` respond_within(Vendor, hours(24))
2	Hosting[SD] Service[SD] Provider[SD] infrastructure[S] and Hosting[SD] Service[SD] Provider[SD] software[S] will comply[R] with HIPAA[OD] standards[O].	comply_with(infrastructure(Hosting Service Provider), standards(HIPAA)) `AND` comply_with(software(Hosting Service Provider), standards(HIPAA))
3	System[S] shall be delivered[R] with no viruses[O][N] or malware[O][N].	!deliver_with(System, viruses) `AND` !deliver_with(System, malware)

logic formulas from NL requirements using our information elements without an in-depth knowledge of syntactic analysis, PoS tags, or grammatical dependencies. Those with this additional knowledge are able to further expand and customize the technique (for example, by adding additional categories) to better fit the task at hand.

Our technique provides some robustness to missing information. Even if one of the six categories is missing its information element, we may still use the information retained within the other categories. For example, if a requirement states that "X *is doing an action to* Y," but subject *X* is missed by the parser, we still retain sufficient information to know that an action is being done to object *Y*. Using PoS tags directly, we would need to have separate rules for both cases to retain this information.

Our generalized information elements allow for less repetitive rules to be explicitly written out, while providing specific and unambiguous formal representations. For example, our *negation* extraction covers cases such as: "*shall not* X," "*shall* X *no* Y," "*does not* X," "*is not an* X," and more. We are able to handle multiple subjects, objects, and descriptors, without setting a limit to the number of items as long as they are written using direct language. Descriptors may be included or excluded as needed.

Our IE technique shows promising results, capturing, on average, close to 86% of the information contained in a given requirement. We have analyzed cases in which some information is missing or miscategorized, and have found that these cases fall under four specific situations. We plan to target those four cases to further improve our technique, and are confident that such improvement will also improve our automatic requirements formalization. Compared to other IE tools, our technique extracts finer-grained information that makes it easy to convert NL requirements into formalizations such as logic formulas, which can be used for tasks such as finding inconsistencies or making inferences about requirements. We aim to further expand our technique to create even more formalization possibilities.

REFERENCES

[1] Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D Manning. Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 344–354, 2015.

[2] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Automated checking of conformance to requirements templates using natural language processing. *IEEE transactions on Software Engineering*, 41(10):944–968, 2015.

[3] Julia Badger, David Throop, and Charles Claunch. Vared: verification and analysis of requirements and early designs. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 325–326. IEEE, 2014.

[4] Igor Buzhinsky. Formalization of natural language requirements into temporal logics: a survey. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 400–406. IEEE, 2019.

[5] Fabiano Dalpiaz, Alessio Ferrari, Xavier Franch, and Cristina Palomares. Natural language processing for requirements engineering: The best is yet to come. *IEEE software*, 35(5):115–119, 2018.

[6] Luciano Del Corro and Rainer Gemulla. Clausie: clause-based open information extraction. In *Proceedings of the 22nd international conference on World Wide Web*, pages 355–366, 2013.

[7] Anthony Fader, Stephen Soderland, and Oren Etzioni. Identifying relations for open information extraction. In *Proceedings of the 2011 conference on empirical methods in natural language processing*, pages 1535–1545, 2011.

[8] Alessio Ferrari. Natural language requirements processing: from research to practice. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 536–537. IEEE, 2018.

[9] Alessio Ferrari and Andrea Esuli. An nlp approach for cross-domain ambiguity detection in requirements engineering. *Automated Software Engineering*, 26(3):559–598, 2019.

[10] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):277–330, 2005.

[11] Xuan Phu Mai, Fabrizio Pastore, Arda Göknil, and Lionel Briand. A natural language programming approach for requirements-based security testing. In *29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*. IEEE, 2018.

[12] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[13] Christina Niklaus, Matthias Cetto, André Freitas, and Siegfried Handschuh. A survey on open information extraction. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3866–3878, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics.

[14] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[15] Michael Schmitz, Stephen Soderland, Robert Bart, Oren Etzioni, et al. Open language learning for information extraction. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 523–534, 2012.

[16] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J Letsholo, Muideen A Ajagbe, Erol-Valeriu Chioasca, and Riza T Batista-Navarro. Natural language processing for requirements engineering: A systematic mapping study. *ACM Computing Surveys (CSUR)*, 54(3):1–41, 2021.