Noname manuscript No.
(will be inserted by the editor)

RCM-Extractor: Automated Extraction of a Semi Formal Representation Model from Natural Language Requirements

Aya Zaki-Ismail · Mohamed Osama · Mohamed Abdelrazek · John Grundy · Amani Ibrahim

the date of receipt and acceptance should be inserted later

Abstract Most existing (semi-)automated requirements formalisation techniques assume requirements to be specified in predefined templates. They also employ template-specific transformation rules to provide the corresponding formal representation. Hence, such techniques have limited expressiveness and more importantly require system engineers to re-write their system requirements following defined templates for maintenance and evolution. In this paper, we introduce an automated requirements extraction technique (RCM-Extractor) to automatically extract the key constructs of a comprehensive and formalisable semi-formal representation model from textual requirements. This avoids the expressiveness issues affecting the existing requirement specification templates, and eliminates the need to rewriting the requirements to match the structure of such templates. We evaluated RCM-Extractor on a dataset of 162 requirements curated from several papers in the literature. RCM-Extractor achieved 87% precision, 98% recall, 92% F-measure, and 86% accuracy. In addition, we evaluated the capabilities of RCM-Extractor to extract requirements on a dataset of 15,000 automatically synthesised requirements that are constructed specifically to evaluate our approach. This dataset has a complete coverage of the possible structures and arrangements of the properties that can exist in system requirements. Our approach achieved 57%, 92% and 100% accuracy for un-corrected, partially-corrected and fully-corrected Stanford typed-dependencies representations of the synthesised requirements, respectively.

Keywords Requirements Extraction · Requirements Formalization · Natural-Language Extraction

A. Zaki-Ismail · Mohamed Osama · Mohamed Abdelrazek · Amani Ibrahim
Deakin University, 221 Burwood Hwy, Burwood VIC 3125
E-mail: amohamedzakiism, mdarweish, mohamed.abdelrazek,amani.ibrahim@deakin.edu.au

J. Grundy
Monash University, Wellington Rd, Clayton VIC 3800
E-mail: john.grundy@monash.edu

1 Introduction

Formal verification techniques, such as model checking and theorem proving, are usually highly recommended – and in many cases mandatory – when proving the correctness of critical systems [3]. To benefit from these formal verification techniques, the systems under development need to be specified in suitable formal notations, such as temporal logic [3] (e.g. LTL, MTL and CTL). Nevertheless, most real-world systems are still specified in natural language (NL) [25]. Formalising NL-requirement specifications – i.e. mapping these requirements to formal notations – is usually done manually [22]. This requires strong expertise in mathematics and in the target system domain to correctly translate system requirements into formal notations while preserving correct system semantics [2,22]. This manual process is time consuming and highly susceptible to errors. Fully or partially automating the formalisation process would reduce the prospect of having errors, in addition to increasing the chances of using formal verification methods in a wider range of systems [3]. The work in [34] provides an overview about requirements representation in different levels of formality (i.e., informal, semi-formal and formal levels) while highlighting the strengths and weaknesses of each level.

Almost all existing requirements formalisation techniques assume that the source requirements are specified in some form of restrictive, predefined templates (*e.g.*, patterns [14], boilerplates [21]), or structured, controlled and limited English [25]. Enforcing a predefined structure on the input requirements has allowed researchers to develop parsing and transformation rules to map the key constructs (with specific format and position in the input requirement sentence) of a given well-formatted/well-structured requirement into formal notations. However, these approaches suffer from several limitations including: *First*, they add an overhead burden on the system engineers to re-write their requirements to conform to the used template(s) even if the requirements are well written (i.e., have no quality issues). *Second*, the user needs guidance to phrase the requirements in compliance with the defined format(s). *Third*, they reduce the expressiveness power of the writing. *Finally*, the format might be so restricted that it becomes irritating to use.

According to recent reviews [2] and [4], there is still a critical need for an approach that can automatically extract and transform existing textual requirements written using different structures and formats – i.e. without predefined templates – into formal notations. To achieve this goal, we developed a two-stage approach [32,33] to first extract the key requirement properties (we refer to them as requirement components and sub-components) from an input textual requirement to be represented in a well-defined semi-formal representation model - RCM. The RCMs are then mapped (using transformation rules) into formal notations.

In [33], we presented a comprehensive semi-formal representation - RCM: Requirement Capturing Model, based on reviewing: 1) several systems requirements, 2) existing templates, constrained natural languages (CNLs) and defined formats for expressing requirements, and 3) existing requirement for-

malisation techniques. RCM defines a comprehensive list of requirement properties. The underlying structure of these properties encapsulates both the semi-formal and formal semantics. In addition, RCM is automatically formalisable into temporal logic using transformation rules [33].

In [32], we introduced an automated requirements extraction technique (RCM-Extractor) that can process sentence-based textual requirements and produce the corresponding RCMs, along with the breakdowns of these requirements. We introduced: (1) ESSGA (Enhanced Simple Sentence Generation Algorithm) for extracting clauses from a sentence(each corresponding to one RCM component) and (2) DSSAM (Deep Syntactic and Semantic Analysis) that extracts the breakdowns of the components from each clause (i.e., sub-components and arguments of a given component) using StanfordNLP part-of-speech (POS) tagger. We evaluated our approach on a dataset of 162 requirements sentences curated from multiple papers in the literature and from several online sources. The evaluation results showed that RCM extractor achieved 75% accuracy. The approach is insensitive to the RCM (sub-)components count, type and order within the given requirement sentence. In addition, It does not mandate requirements to follow any defined format (e.g., CNL and patterns). However, DSSAM cannot extract clauses that do not match any of the underlying POS analysis rules (a rule is a possible sequence of the words types).

In this paper, we significantly extend our work in [32] by:

- Improving the performance of the RCM-Extractor by developing two modified extraction versions (Enhanced-DSSAM and Hybrid-DSSAM). We developed Enhanced-DSSAM (discussed in section 4.4) based on Typed-dependencies (TDs) (i.e., the syntactic relations between the words in the sentence) to overcome the mismatch of the POS rules in the original DSSAM. Hybrid-DSSAM (discussed in section 4.5) avoids the drawbacks of both DSSAM and Enhanced-DSSAM and achieves the best performance (by combining both POS and TDs rules).
- Developing a new RCM-refinement process (discussed in section 5) that associates formal-semantics to the extracted semi-formal semantics. This results in producing a refined RCM (encapsulating both semi-formal and formal semantics) for the given requirement. The refined RCMs allow the generation of multiple formal notations from using our formalisation approach presented in [33].
- Extending the evaluation to cover all the mentioned techniques (comparison of the original and modified versions) on the same dataset of 162 requirements [32]. Additionally, we also utilised a new dataset of 15,000 automatically synthesised requirements covering all the possible structures of the requirement properties to assess the robustness of the developed approaches. The extended evaluation and comparison of the developed versions are discussed in section 6.2). Furthermore, we measured the average time for extraction given the length of the input sentence (in section 6.1.2).

The remainder of this paper is organised as follows. Section 2 covers the key related work. Section 3 provides an overview of the requirement capturing

model - RCM. Section 4 introduces the details of the RCM-Extractor. Section 5 provides the details of the supplementary RCM-refinement process. Section 6 presents the evaluation experiments we conducted to evaluate the performance of the RCM-Extractor. Section 7, discusses the key findings and limitations of the approach. Finally, Section 8 concludes the paper.

2 Related Work

There is a rich and diverse body of research for formalising textual requirements into formal notations. The main paradigm is feeding a tool with requirement sentences written in predefined format(s) [16, 24, 25, 21, 26, 14, 20, 9]. The structure of the template is utilised to support the parsing of the input NL requirements. The extraction of each technique differs slightly according to the supported elements in the template. In contrast to this approach, our technique aims to process NL requirements instead and thus eliminate the overhead work of rewriting the requirements. In addition, it covers a wider range of requirements structures (i.e., our approach is insensitive to the number, order, and types of the components constituting a requirement sentence).

Sturla [27] provides a two-phase approach to parse natural language sentences into a logical format. The first phase parses English utterance, only in the form of simple sentences, into a logical structure. The second phase assigns values to the discovered variables of functions and constants in phase one. The approach achieved 38.16% accuracy on 76 sentences for both phases combined, and 46% and 78% accuracy for the first and second phase, respectively. Our RCM-Extractor approach processes more complex forms of requirements sentences, and achieves far better accuracy.

Ghosh et al., in [10], proposed a framework called ARSENAL for translating natural language requirements into Linear Temporal Logic (LTL). ARSENAL first reduces the complexity of the input sentence through term replacement. Next, it creates an intermediate representation (IR) enriched with formal information resulting from applying a set of hand crafted mapping rules on the typed dependency of the input sentence. These rules are domain-specific and limited to restricted scenarios. The constructed IR can be later converted into LTL. The approach achieved 78% and 95% accuracy on two different datasets. However, the accuracy of the second dataset is reduced from 95% to 65% by perturbing "If A then B" to "B if A" (i.e., indicating that the approach is order sensitive). In addition, it does not support requirements with temporal information. In contrast, RCM-Extractor is insensitive to the components order and is able to process time information missed by ARSENAL. However, we still do not support coordinating relations. Nevertheless, we are aiming to extend our extraction approach to address this issue.

Nelken et al., in [22], proposed a formalisation method that converts constrained English utterances ("verb to be" based) into Action Computation Tree Logic (ACTL). First, the approach converts the English utterances into an intermediate representation called discourse representation structure (DRS)

based on Kamp's Discourse Representation Theory [15]. Then, the DRSs are transformed into ACTL logical formulas. Although the requirements engineers are required to specify the requirements in a precise and concrete language, some tolerance is permitted in the use of multi-sentence in a requirement (e.g., pronominal anaphora and inter-sentential links).

Yan et al., in [30], presented an NLP-based technique for formalising requirements into LTL. Similar to our approach, this technique identifies clauses of a requirement and maps them to propositions in LTL. Although, this approach allows coordination between clauses, it is strictly limited to a predefined clause structure only. The clause must contain a single-word noun as the subject and a "verb to be" based predicate. Moreover, complex cases of NL (e.g., relative clauses, imperative cases, and intermingled clauses) are not addressed. Time scope and repetition properties are not considered as well. These limitations are handled in our approach, in addition to covering a wider range of complex formats for clauses containing noun phrases, compound nouns, verb phrases with multiple complement and/or object, and multi-prepositional phrases.

3 Requirement Capturing Model (RCM)

This is an overview of our target reference model – the RCM model, presented in [33]. RCM is a semi-formal representation model that aggregates the key requirements properties – components and sub-components – existing in the literature and necessary to transform textual requirements into formal notations. It supports a wide range of requirements because the model adapts to any permutation of its (sub-)components.

Fig. 1 shows a simplified representation of RCM. Each system requirement may have one or more primitive requirements, each representing one sentence. A primitive requirement can contain zero or more conditions, triggers, and

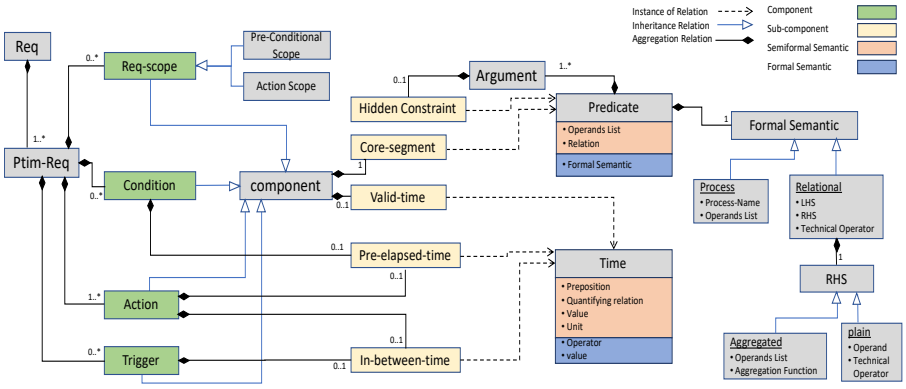


Fig. 1: Compact Meta-model of the RCM [33]

Req-scopes, in addition to one or more actions. These components are defined as follows:

- **Trigger:** is an event that automatically initiates/fires action(s) whenever it occurs within the system life-cycle (e.g., "When it rains for 1 minute" in R4 Fig. 2).
- **Condition:** represents the constraints that should be explicitly checked by the system before executing actions (e.g., if the maximum deceleration is [insufficient] in R3 Fig. 2).
- **Req-scope:** determines the operational context under which (i) conditions and triggers can be valid – called "a pre-conditional scope", or (ii) actions can occur – called "an action scope". The Scope may define a start boundary (e.g., "after sailing termination"), an end boundary (e.g., "before $< B\_sig >$ is [TRUE]" in R1 Fig. 2), or both (e.g., "while R is true" can be expressed by after and until as "after R is true" and "until not R").
- **Action:** represents the task that should be executed by the system in response to triggers and/or constrained by conditions (e.g., the inhibitor shall transition to [true]" in R1 Fig. 2).

A component can be broken into sub-components. RCM uses the following five types of sub-components:

- **Core-Segment**: expresses the core part of the component including: the operands, the operator and the negation flag/property (e.g., in "In case of $<: A\_sig >$ is [True]" the "$<: A\_sig >$" and "[True]" are the operands and "is" is the operator).
- **Valid-Time**: is an optional sub-component. It provides the valid period of time for the component of interest including: the quantifying relation (e.g., "=","<",">", etc.), the time length, and the unit (e.g., in "the vehicle warns the driver by acoustical signals $< E >$ for 1 seconds" the action is valid for 1 second length of time).
- **Hidden constraint**: is a constraint defined for a specific operand within a component. For example, in "the high beam headlight that is activated is reduced" in R2, the *that is activated* is a constraint defined for the operand *the high beam headlight*.
- **Pre-elapsed-time:** can be found for action and condition components. This indicates the length of time from an offset point before the action can begin or the condition can be checked (e.g., "the wipers are activated **within 30 seconds**" in R4).
- **In-between-time**: is attachable to action and trigger components. It is used to reflect the elapsed time between two consecutive occurrences of the event in case of repetition (e.g., "the vehicle warns the driver by acoustical signals <E> for 1 seconds **every 2 seconds**" in R3).

The underlying internal structures of components and sub-components are Predicate and Time structures that encapsulate both the semi-formal and formal semantics.

4 Approach: RCM-Extractor

RCM-Extractor is responsible for extracting the semi-formal breakdowns of components and sub-components of RCM from NL requirements. It consists of five main phases as shown in Fig. 2.(b). The set of five input requirements are shown in Fig. 2.(a) and the RCM representation of requirement "R1" is shown in Fig. 2.(c). Fig. 2.(d) provides the TL formula produced by the formalisation rules applied on the extracted RCM of R1. The proposition variables B, C, S, and A correspond to "sailing termination being happened", "<cal: A_sig> is [True]", "$< B\_sig >$ is [TRUE]", and "the inhibitor shall transition to [true]", respectively (i.e., <cal: A_sig> and $< B\_sig >$ are domain technical variables referring to signal A and signal B, respectively). RCM-Extractor utilises StanfordNLP, the most widely used NLP-tool [35], to get Part-of-Speech (POS) tags, Typed-Dependencies (TDs) relations, and parse tree (PT) of the input sentences. It also uses WordNet as the most widely used NLP-resource [35]. RCM-Extractor accepts the following as input:

- Requirements document: a file of system requirements (each requirement can be one or more sentences) written in natural language (i.e., English).
- Technical terms: a file containing domain-specific terms expressed in English. Such terms may introduce non-finite clauses to the requirement sentence (e.g., "if the acceleration requested by the driver exceeds 20", the underlined text is a domain term with non-finite format).
- Configuration: a file containing a set of configurations for the requirements (*e.g.*, (1) non-English words format like technical variables, (2) overloaded-English words like technical values, and (3) requirements separators).

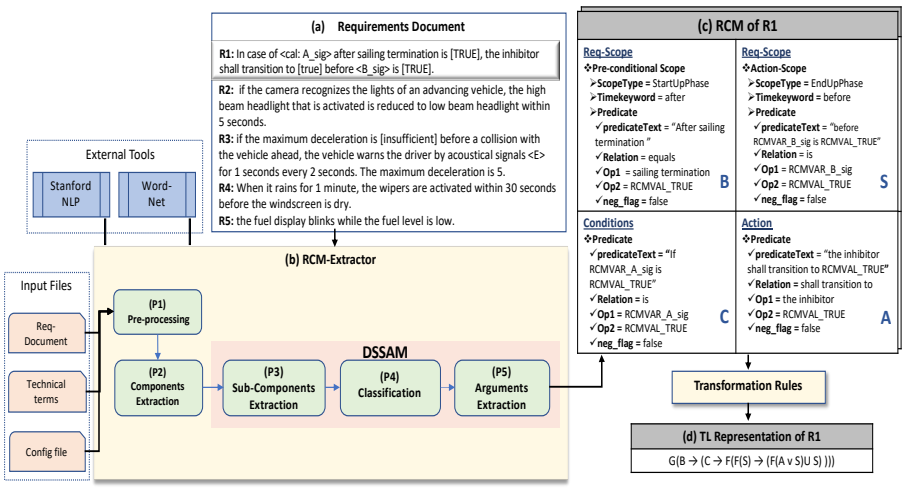


Fig. 2: Textual Requirement Extraction Flow

4.1 RCM-Extractor Phase 1: Requirements pre-processing

Requirements specifications are usually written in long sentences (e.g., R3 in Fig. 2) and comprise a more restricted vocabulary, where 62% of the used words do not appear in generic text [7] (e.g., the word "$< cal : A\_sig >$" in R1 in Fig. 2). Hence, the existing NLP tools (StanfordNLP in our case), that are trained on generic text, will not provide a reliable performance on requirements. To avoid this problem, we pre-process the requirements to eliminate the defective and unnecessary elements and adjust the incomprehensible and overloaded words in the input sentences. The steps for this process are as follows:

1. **Requirements cleaning:** multiple spaces and incorrect ones (i.e., at the beginning or at the end of the sentence) are removed. Other styling formats are also considered (e.g., "-" is replaced with "_").
2. **Requirements conciseness:** because the requirements are written in NL, the users may use appositives to link a technical term to its representing technical variable (e.g., "If the fuel amount exceeds **F_min_thr** , the fuel minimum threshold, the fuel level is set to HIGH" – the technical term is underlined and the technical variable is in bold). In this step, appositive cases are extracted and stored aside to prevent loss of information. This makes requirement sentences shorter and simpler to analyse.
3. **Requirements configuration:** for each system, a specific format is utilised to write the technical variables and values (e.g., writing technical variables between "<>" and the values between "[]"). In most cases, such variables and values either have no meaning in English (non-English words) or have a different meaning from the intended one (overloaded English words). For example, in the action "turn $< btn\_1 >$ to [on]", the "$< btn\_1 >$" is a non-English word and "[on]" is an overloaded English word because it acts as a value in the action although it is commonly used as a preposition in English. RCM-Extractor detects such formats, according to the configuration file, and then transforms them into an internal format (by adding a suitable prefix "RCMVAR_...", "RCMVAL_..." – e.g., "$< cal : A\_sig >$" in Fig. 2 is replaced with "$RCMVAR\_A\_sig$" in Fig. 3). Similarly, the input technical terms are detected in the requirements and transformed to an internal format (e.g., the term is concatenated with "_" and the prefix "RCMTERM_" is added). Finally, the input requirements are separated based on the separation format in the configuration file.
4. **Foreign words substitution:** to overcome the issue of StanfordNLP handling restricted words in requirements (non-English words and domain-specific words), we replace the identified non-English/overloaded words in the *requirement configuration step*(e.g. technical variables, technical terms and technical values) with placeholder English words representing them (e.g., variable, term and value). The original words are restored again after the application of the StanfordNLP analysis.
5. **Closed words unification:** English has closed word classes [17]. Each class contains a finite set of words with a defined grammatical function

(e.g., conjunctions, subordinating conditional keyword, timing keyword, instant timing keyword). In this step, all English words holding the same function are unified and converted to a single unique word, selected from the class to be its representative within our approach (e.g., all subordinating instant-timing keywords {whenever, once, .etc} are replaced with "when").This simplifies the extraction while covering various alternatives of the text. For example, "In case of" in Fig. 2. is replaced with "if" in Fig. 3.

6. **Ambiguity Resolution:** to overcome ambiguity issues, we utilise our ambiguity detection and resolution approach, proposed in [23]. This approach provides feedback to the user when it detects multiple interpretations of the same sentence with a recommendation of the most likely correct interpretations to select one from them. After that, each requirement is attached with the selected interpretation and the corresponding TDs, PT and POS as well. We base our extraction approach on the provided unambiguous versions of requirements and their attached TDs, POS and PT.

4.2 RCM-Extractor Phase 2: Requirement Components Extraction

In this process, we extract the RCM components (action, Req-scope, trigger and condition) from each NL requirement sentence. We base our enhanced component extraction algorithm (ESSGA) on the original SSGA proposed in [5]. SSGA generates simple sentences, each representing one clause, from a complex/compound sentence through analysing the TDs of the input sentence. The TDs are a set of mentions, each representing a syntactic relation between two words (e.g., nsubj(equals-14, X-13) means "X" is subject to the verb "equals").

SSGA suffers from two major limitations. First, it fails to identify imperative clauses within sentences. This is because the algorithm is only able to generate declarative sentences by locating the existing subjects/passive-subjects (considered as the starting points of the simple sentences). Second, SSGA does not keep any information reflecting the semantic relations of the generated simple sentences. For example, in the sentence "if X is ON, Z shall be set to True", SSGA produces the sentences "X is on" and "Z shall be set to True" while neglecting the implication logical relation connecting them (loss of information).

ESSGA constructs clauses starting from the verb (because it is a mandatory component) instead of the subject in both the imperative and declarative types. Furthermore, ESSGA keeps the semantic relations because they represent important domain characteristics forming the type of each component (e.g., subordinating head "If" in the previous example reflects a Condition type). ESSGA is developed based on the automated NLP analysis types provided by StanfordNLP: (i) TDs (i.e., words relations), and (ii) POS (i.e., grammatical type of each word like noun, verb, adj, .etc), in addition to the manually annotated POS tags provided by WordNet.

Fig. 3 outlines the steps used to extract the clauses of requirement R1 in Fig. 2. In **step1**, the elements identifying the clauses -verbs in our case- are marked. First, we get the POS of the given sentence using StanfordNLP and then highlight the main verbs. We confirm the correctness of the highlighted verbs using the POS of WordNet (defective cases "not a verb" are removed). In **step2**, we get the TDs of the input sentence using StnafordNLP. In **step3**, we break the connection between the clauses by removing the same TDs mentions, used by SSGA, that connect them. However, we exclude the mentions reflecting important domain relations (e.g., "mark", "ccomp", "ref", "dep"). The removed mentions are marked with "✗" in Fig. 2. In **step4**, all the mentions having direct or indirect connections are grouped. Finally, in **step5**, the distinct words within each group are sorted according to their occurrence indices in the original sentence forming one component.

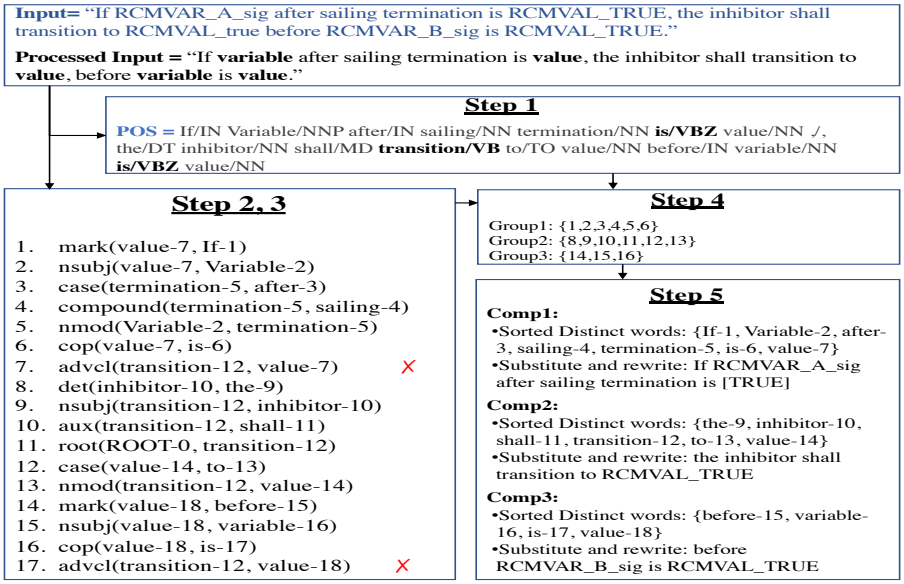


Fig. 3: Components Extraction Example

ESSGA handles several complex challenges for processing and extracting the requirements specified in NL. First, the requirement components may exist in any order in the sentence (e.g., R3.S1 and R4 in Fig. 2 show alternative orders). Second, the components may also be intermingled (e.g., "In case of $< cal : A\_sig >$ after sailing termination is [TRUE]" in R1, the two components "In case of $< cal : A\_sig >$ is [TRUE]" and "after sailing termination" are mixed up). Third, a requirement sentence may be expressed in different structures (e.g., simple sentence, compound sentence, complex sentence or compound complex sentence) with different types of clauses (e.g., imperative

and declarative) and different voice (e.g., active and passive). Our ESSGA algorithm overcomes these challenges by breaking the connections between the clauses and reforming each clause on its own as a component, in step3 and step4. This separation makes the approach insensitive to intermingling components, their order, and the sentence structure. In addition, step1 handles both the imperative and declarative types by considering the verb as the starting point.

A component may be expressed by an incomplete clause following a correct syntax and hold implicit meaning (e.g., "after sailing termination" in R1 implicitly means that the "sailing termination" has already happened). This case is only eligible to adverbal clauses [12], where the clause involves a time-related conditional keyword that corresponds to the Req-scope component type. This incompleteness may cause the components to be merged, especially if the missing part is the verb (e.g., before termination). However, this merge is resolved later in following phases.

4.3 Deep Syntactic and Semantic Analysis (DSSAM)

Our Deep Syntactic and Semantic Analysis (DSSAM) is responsible for deeply understanding each component to complete its extraction. It consists of the remaining three phases: sub-components extraction, classification, and arguments extraction.

4.3.1 RCM-Extractor Phase 3: Sub-components Extraction

In Phase 3, each component is processed to extract the sub-components constituting it. Fig. 4 shows two extracted sub-components (S[1] and S[2]) for the obtained component "C1" in Fig. 3.

Processed Component C[1]			
Internal Pre-processing of C[1]	Text	If variable after sailing termination is value	
	POS	IN, NN, IN, NN, NN, VBZ, NN	
Component abstraction	Text	If **[variable]** after **[sailing termination]** [is] **[value]**	
	POS	IN**, basicNP,** IN**, basicNP,** basicVP**, basicNP**	
Extracted Sub-components	**S[1]**		**S[2]**
	[If] [variable] [is] [value] ➔Core-segment **POS**= [['IN'],['basicNP'], ['basicVP'],['basicNP']]		[After] [sailing termination] [] [] ➔ Core-segment **POS**= [['IN'],['basicNP'],[],[]]

Fig. 4: Sub-component extraction example for component "C1" in Fig. 3

This phase consists of two steps: (1) abstracting the input component, and (2) identifying the boundaries of each sub-component. **First**, the initial POS tags of all the tokens within a component are scanned to identify noun and verb phrases. For each identified phrase, the POS tags of its tokens are replaced with a single tag as in the example in Fig. 4 ("basicNP" for noun phrases

and "basicVP" for verb phrases). This is carried out via regular expressions that match the POS tags of the component against a set of hand-crafted patterns covering the possible structures of noun and verb phrases in the English language.

Second, the boundaries of each sub-component are identified by locating the head (starting word) and the most suitable body (the next set of words to fulfil a correct grammatical structure for the sub-component) as in Fig. 5. Some sub-components (like the core-segment in condition, trigger, and Req-scope) have a known head keyword as a result of the unification step in pre-processing (e.g., the head of a trigger is "when"). The heads of other sub-components (i.e., pre-elapsed-time, valid-time, in-between-time, hidden constraint) have exclusive POS tags (e.g., the hidden constraint starts with a relative pronoun having the POS tag "WP$" or "WDT").

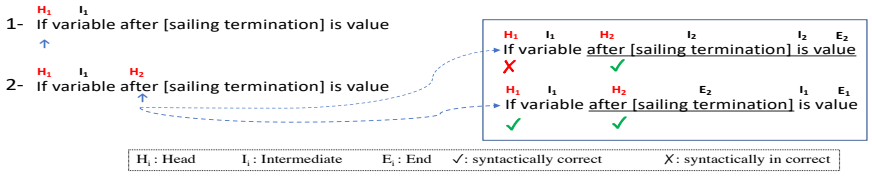


Fig. 5: Best suitable sub-components decomposition for "C1"

Identifying the body of a sub-component is more challenging than the head identification because of the various possible structures. To overcome this, we created a set of hand-crafted reasoning POS-based rules representing the possible structures of each sub-component (i.e., relative clauses, adverbal clauses, subordinating clauses, declarative clauses, and time prepositional phrase). Fig. 6 provides the alternative structures for adverbal clauses. We also developed a recursive technique to determine the most suitable body structure for an intended sub-component while taking into account the remaining sub-components. For example, the second sub-component in Fig. 5 starting at "H_2" conforms to two possible structures, but only one of them will prevent the other sub-component from being grammatically incorrect. The reasoning rules were developed using Prolog to benefit from its built-in matching and backtracking inference engine.

Adverbal Clause	Head	Head: Start with (while, After, Before)	-Before x transitions to true
	Body	✓ NP Verb (NP?)	-Before terminating the IDC
		✓ Verb (NP?)	-Before the window start moving
		✓ NP Verb	-Before IDC termination
		✓ NP	

Fig. 6: Alternative structures of Adverbal clause
Note: the verb itself has various structures conforming to the tense, person, voice, etc; the same applies on NP

4.3.2 RCM-Extractor Phase 4: Classification

The aim of this phase is assigning labels (e.g., Trigger, condition, action, Req-scope and Factual Rule) to each of the extracted components, and to the extracted sub-components as well (e.g., pre-elapsed-time, valid-time, in-between-time and hidden constraints). The classifier assigns types by applying two-levels of checking on the obtained sub-components. The first level, as indicated in Fig. 7, identifies types based on three attributes: (1) the head of the given sub-component, (2) comp count: the total count of the extracted components of the current primitive requirement, and (3) the count of the extracted core-segments sub-components from the given component. It is worth noting that the sub-components heads are unified in step3:*closed words unification* in the pre-processing phase. In addition, the classification is done in compliance with the types of clauses discussed in [12], where (1) independent clauses are identified through "No Head", (2) subordinating clauses are identified through: "conditional, instant-conditional, and time-conditional heads", and (3) relative clauses are identified through "Relative head". The coordinating clause is first adjusted to one of the other types (i.e., independent, subordinating or relative) based on its main attached clause (e.g., "if X is True or Y is True" $\longrightarrow$ "if X is True, or if Y is True"), then it becomes ready for the classification process. In this level, the merged components, in the components extraction phase, if any, are detected based on the obtained core-segments count as indicated in Fig. 7 and are separated as in Fig. 8.

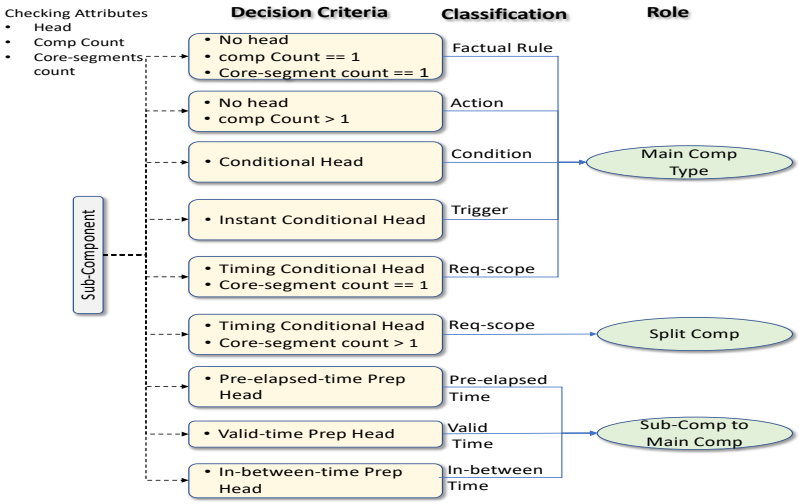


Fig. 7: Classification Checking

In the second level, Req-scope component type is further classified to either "action scope" or "pre-conditional scope". The type is identified based on the

Classification of the sub-components of C[1]		
	S[1]: core-segment conditional head	S[2]: core-segment Timing Conditional Head
Level-1	Component ➔ Condition	Component ➔ Req-Scope (split into separate component)
Level-2		Pre-conditional Scope

Fig. 8: Classification of the obtained sub-components in Fig. 4

surrounding components. If a Req-scope is found in a merged component, its type will be identified based on the component merged with it (i.e., if action => action scope, else ==> Pre-conditional scope). Otherwise, we rely on the nearest non-Req-scope component within the primitive requirement sentence.

4.3.3 RCM-Extractor Phase 5: Arguments Extraction

This process identifies the complete arguments of a given sub-component. To achieve this, we benefit from the sub-component extraction process by constructing an initial argument decomposition within each identified sub-component. Each sub-component contains two lists. The first list contains the text of the sub-component, initially broken down into separate arguments. The second list contains the corresponding POS tags of each argument in the first list. For example, the second sub-component in Fig. 4 is represented with: (1) text=[['After'],['sailing_termination'],[],[]] and (2) POS= [['IN'],['basicNP'],[],[]]. Both lists are constructed with the same static length that is determined based on the most complete possible version of the sub-component (i.e., the case where all the elements of the sub-component are present), as follows:

- **Core-segment:** the adopted structure consists of head keyword, subj, verb, complement. This sub-component type can represent:
  1. Condition case: conditional keyword, subj, verb, complement (e.g., [If] [X] [exceeds] [Y])
  2. Trigger case: conditional keyword, subj, verb, complement (e.g., [when] [X] [exceeds] [Y])
  3. Action case: empty-item, subj, verb, complement (e.g., [x] [shall be set to] [True)]
  4. Req-scope: timing conditional keyword, subj, verb, obj (e.g., [After] [X] [transitions to] [True])
- **Hidden constraint:** relative noun, relative pronoun, subj, verb, obj (e.g., [X] [that] [is] [larger than] [2])
- **Time:** time head, quantifying relation, value, unit (e.g., [for] [at most] [2] [seconds]). This applies to pre-elapsed-time, valid-time and in-between-time sub-components.

The role of each argument is defined based on the argument location in the list. In case of missing element(s), the corresponding location of such el-

ement(s) is(are) left empty to maintain the proper ordering and roles of the expected elements within the lists.

Further processing is applied on the initial arguments to: (1) identify more arguments, if possible, and (2) provide artificial arguments when needed. As discussed earlier, a sub-component may eventually be mapped to either time or predicate structure in RCM. For sub-components that are eventually mapped to predicate structure, they may have two or more operands (e.g., **X** exceeds **Y** with **2**). In this case, technically, the complement part of the sentence may be decomposed into a set of operands. The determinant factor in the decomposition is detecting the engendered nouns in the complement attached to the predicate verb. Algorithm 1 shows our technique for identifying these engendered nouns. Fig. 9 shows an example of applying this to a requirement.

Algorithm 1 Extracting Verb Operands

Input: Sub-Comp(text, POS-List)
Output: verb operands list
procedure
Step 1: get Typed-Dependency of the Sub-component text.
Step 2: extract the main verb of the Sub-component text.
Step 3: get all mentions with the functionality (object|noun-modification) that connect "the extracted main verb" with "a word of the type noun".
Step 4: extract all engendered nouns from the filtered mentions in Step 3.
Step 5: divide the complement into set of noun phrases each starts with an engendered noun obtained in Step 4.
end procedure

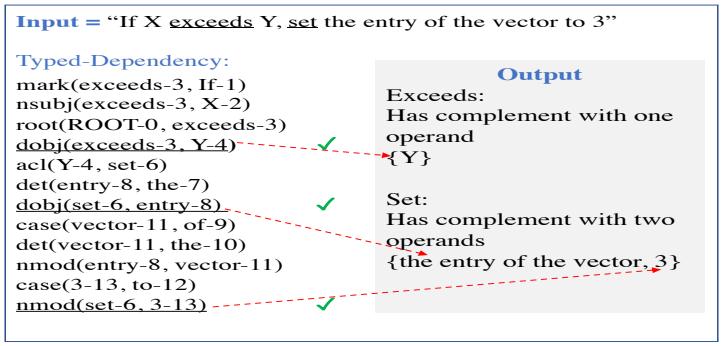


Fig. 9: Example illustrating Operands extraction of a verb

An RCM model may be populated with semantically incomplete but syntactically valid sub-components. For example, "After sailing termination" is a syntactically valid NL predicate but is incomplete because it has only one argument "termination". Thus it is corrected to "After sailing termination equals true" with the underlined artificial argument for completeness. We ad-

dress these incompleteness issues by attaching proper artificial arguments as in Fig. 10. Artificial arguments are provided to incomplete predicates that have one or two arguments at maximum (i.e., a complete predicate should have at least three arguments constituting the LHS, RHS and operator connecting them in the formal form of a predicate). In case of one argument, the argument is either noun or verb (e.g., before termination or after starting up). Arguments of the other case consist of noun and verb (e.g., subj+verb "If the fuel display blinks" or verb+subj "integrate X"). Table 1 illustrates how the proposed approach handles these cases.

Table 1: Handling Incomplete Predicates

Exist Args	**Taken Action**	**Added Args**	**Example**
0ne	(1) make it Op1 (2) add two argument (Rel and Op2)	Rel="equal" & Op2="True"	- old = "After termination" - new = "After termination equals True"
Two	(1) make the noun Op1 (2) Make the verb Op2 (as a status to Op1). (3) add one argument (Rel)	Rel="be"	- old = "integrate X" - new = "(X) (shall be) (integrated)"

Extracted Arguments	S[1]	S[2]
	-Cond-keyword = if -OP1 = RCMCAL_A_sig -OP2 = RCMVAL_TRUE -Relation = is	-Time cond-keyword = after -OP1 = sailing_termination Created Artificial arguments -OP2 = **RCMVAL_TRUE** -Relation = equals

Fig. 10: Arguments Extraction Example for Component "C1" in Fig. 3

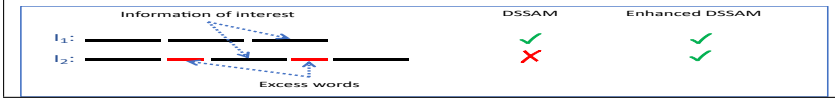
4.4 Enhanced-DSSAM

Although DSSAM shows very good performance when analysing components based on clause-structure, it is not efficient for handling clauses whose grammatical structures do not match any of the variations supported by the POS sequences in phase3 (sub-components extraction phase). We overcome this limitation in the Enhanced-DSSAM version by extracting any sub-component based on the syntactic relations (TDs) between the words instead of their POS sequences.

Instead of scanning and chunking the given POS sequence of the component text into information composing the corresponding sub-components and arguments, the sub-components are extracted from the given text using its internal connecting relation (TDs). This extraction approach shows better performance because it locates the relations of interest relevant to any sub-component without being affected by excess words (i.e., word(s) whose existence/disappearance does not affect the formal semantics of the requirement

but breaks the POS sequences in DSSAM). Table 2 highlights the differences and similarities between the DSSAM versions.

Table 2: DSSAM vs Enhanced-DSSAM

	DSSAM	Enhanced-DSSAM
Difference	The sub-components extraction is POS-Tagging analysis based	The sub-components extraction is Typed-dependencies analysis based
Similarity	Classification and Arguments extraction (Phase 4&5)	
Pros/Cons	-Affected by the excess words -An input may have no output	-Is not affected by excess word(s) -Every input has output
		

The modified phase (sub-components extraction) has two steps: (1) abstracting the input component, and (2) identifying each sub-component utilising TD-based extraction rules (available online [1]). In step1, the TDs of the input components are computed. Within these TDs, mentions forming phrases (noun phrases and verb phrases) are aggregated according to the rules illustrated in Table 3.

Table 3: Enhanced-DSSAM Entities and Relations Extraction rules

Process		
1- Identify TDs relations of interest 2- Extract and aggregate the corresponding elements 3- Update all typed dependencies to contain the aggregated elements 4- Remove unwanted TDs		
Element	**TD relations**	**Example**
Compound Noun	Determinants, compound, amod	IN: det(**frequency-2**,**the-1**) nsubj(consistent-13,frequency-2) OUT: nsubj(consistent-13, the-1 frequency-2)
Noun phrase	nmod, case	IN: $case$(**Engine-4 Control-5 System-6 inspection-7**, **of-3**) $nmod$(**the-1 frequency-2**, **Engine-4 Control-5 System-6 inspection-7**) $nsubj(consistent-13, the frequency-2)$ OUT: nsubj(consistent-13, **the frequency-2 of-3 Engine-4 Control-5 System-6 inspection-7**)
Verb Phrase	Aux, auxpass, cop	IN: aux(consistent-13, **shall-11**) cop(**consistent-13**, **be-12**) nsubj(consistent-13, the frequency-2 of-3 Engine-4 Control-5 System-6 inspection-7) OUT: nsubj(**shall-11 be-12 consistent-13**, the frequency-2 of-3 Engine-4 Control-5 System-6 inspection-7)

In Step2, the abstracted TDs are scanned to extract the sub-components in type-based order (i.e., time, hidden-constraint, core-segment). The TDs are iteratively scanned, producing one sub-component at a time until no further

[1] Enhanced-DSSAM TDs Rules: `https://github.com/ABC-7/RequirementsExtraction/blob/main/EnDSSAMRules.pdf`

matching is found. The sub-component, whose breakdown relations exist, is extracted and its corresponding TDs are removed.

We modified the first component of R1 to "If RCMVAR_A_sig just after sailing termination equals [TRUE]" by adding an excess word "just". Then, we fed the component to both DSSAM versions. Fig. 11 shows the step by step execution of both version highlighting the capability of Enhanced-DSSAM in overcoming excess words.

<table>
<tr><th colspan="4">Processing C[1] by Enhanced-DSSAM</th><th colspan="3">Processing C[1] by DSSAM</th></tr>
<tr><td>Original text</td><td colspan="3">If RCMVAR_A_sig just after sailing termination equals [TRUE]</td><td>Original text</td><td colspan="2">If RCMVAR_A_sig just after sailing termination equals [TRUE]</td></tr>
<tr><td rowspan="2">Internal Pre-processing of C[1]</td><td>Text</td><td colspan="2">If variable after sailing termination equals value</td><td rowspan="2">Internal Pre-processing of C[1]</td><td>Text</td><td>If variable after sailing termination equals value</td></tr>
<tr><td>TD</td><td colspan="2">1. mark(equals-7, If-1)
2. nsubj(equals-7, variable-2)
3. advmod(variable-2, just-3)
4. case(termination-6, after-4)
5. compound(termination-6, sailing-5)
6. nmod(equals-7, termination-6)
7. root(ROOT-0, equals-7)
8. obj(equals-7, value-8)</td><td>POS</td><td>IN, NN, IN, NN, NN, VBZ, NN</td></tr>
<tr><td rowspan="2">Component abstraction</td><td>Text</td><td colspan="2">If variable just after [sailing termination] equals value</td><td rowspan="2">Component abstraction</td><td>Text</td><td>If [variable] just after [sailing termination] [equals] [value]</td></tr>
<tr><td>TD</td><td colspan="2">1. mark(equals-7, If-1)
2. nsubj(equals-7, variable-2)
3. advmod(variable-2, just-3)
4. case(sailing termination-6, after-4)
5. nmod (equals-7, sailing termination-6)
6. root(ROOT-0, equals-7)
7. obj(equals-7, value-8)</td><td>POS</td><td>IN, basicNP, RB, IN, basicNP, basicVP, basicNP</td></tr>
<tr><td>Extracted Sub-components</td><td>S[1]
[If] [variable] [equals] [value] ➔Core-segment
Matched TD: [1,2,7]
• mark(equals-7, If-1)
• nsubj(equals-7, variable-2)
• obj(equals-7, value-8)</td><td>S[2]
[After] [sailing termination] [] []
➔ Core-segment
Matched TD: [4,5]
• case(sailing termination-6, after-4)
• nmod (equals-7, sailing termination-6)</td><td>Extracted Sub-components</td><td colspan="2">No output (no decomposition matches syntactic variations)</td></tr>
<tr><td>Classification</td><td>Component ➔ Condition</td><td>Component ➔ Req-Scope (split)
➔ Pre-conditional Scope</td><td>Classification</td><td colspan="2" rowspan="2">Suspended phases</td></tr>
<tr><td>Extracted Arguments</td><td>-Cond-keyword = if
-OP1 = RCMCAL_A_sig
-OP2 = RCMVAL_TRUE
-Relation = equals</td><td>-Time cond-keyword = after
Created Artificial arguments
-OP1 = sailing_termination
-OP2 = RCMVAL_TRUE
-Relation = equals</td><td>Extracted Arguments</td></tr>
</table>

Fig. 11: DSSAM vs Enhanced-DSSAM execution of Modified C[1]

4.5 Hybrid DSSAM

DSSAM may show better performance than Enhanced-DSSAM in "requirements with no excess words", because Stanford-POS has better accuracy than Stanford-TDs. However, Enhanced-DSSAM is capable of producing (partially-)correct output for requirements with excess words (DSSAM fails to extract these requirements). To achieve even better extraction performance, we aggregated both version as one extraction process. A given component is fed to phase 3 in DSSAM, and in case of extraction failure, it is re-fed into phase 3

in Enhanced-DSSAM. And then phase 4 and phase 5 are applied as indicated in Fig. 12.

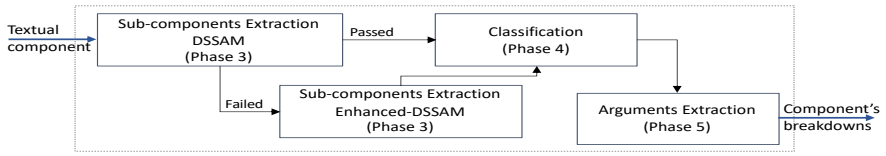


Fig. 12: Hybrid-DSSAM Flow

5 RCM Refinement

RCM refinement is responsible for associating the RCM's formal semantics to the extracted semi-formal one, in addition to unifying the requirements. It consists of two phases: breaking dependencies and formal-semantics mapping.

5.1 Breaking Dependencies

There are two types of dependencies that we aim to resolve in this phase: (1) local dependencies and (2) global dependencies. The former relates to dependencies within the same requirement. Local dependency may be within the same sentence (primitive requirement) or among multiple sentences of the same requirement (primitive requirements in one RCM) (e.g., referenced pronouns). The Later type of dependency relates to dependencies within multiple requirements, each represented with one RCM (e.g., multiple English terms, each in a different requirement, referring to the same system entity).

5.1.1 Global Dependency:

In this type of dependency, we **first** build an entities map (EM) to aggregate the entities within all the requirements (EM is used later in breaking global dependencies). This map is easier to be built after extraction because all the arguments are identified. Second, we utilize EM in breaking global dependencies. The map groups synonyms of each entity together and assigns a unique code to non-correlated entities as indicated in Fig. 13. Synonym entities come from: (1) appositives in the prepossessing conciseness step, and (2) factual rules (acting as descriptive requirements) incorporating entities only (e.g., "R1: $< EM\_State >$ indicates the elevator motion status", where, (i) "indicates" is a descriptive verb, and (ii) $< EM\_State >$ and "the elevator motion status" are entities). **Second**, we iterate over the RCMs to unify all the fuzzy/descriptive entities with the technical ones (i.e., un-mapped descriptive entities can be listed for the users for optional mapping according to their preference).

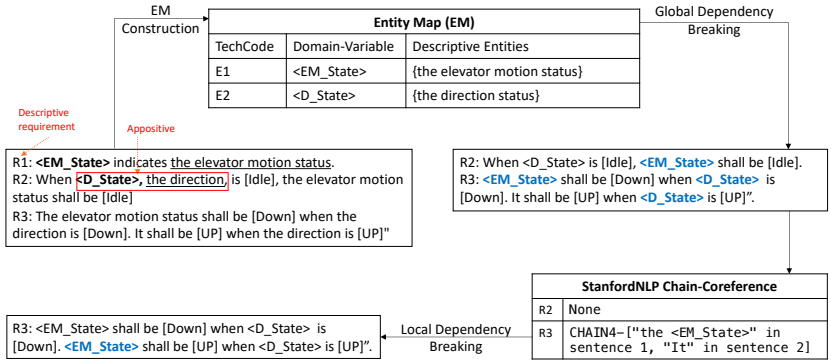


Fig. 13: Global and Local Dependency Breaking Example

5.1.2 Local dependencies

We resolve local dependencies with the support of StanfordNLP as in Algorithm 2. We get the candidate terms referencing the same entity in a given requirement by StanfordNLP co-reference chain. It provides us with the list of candidates (each is a series of co-references for one entity as in Fig. 13). For each candidate (if any), we get –for all co-references– the corresponding component and operand in RCM. Finally, we unify all co-references with the technical entity (i.e., the operand that have a technical code in EM as in Fig. 13) in case of having one, otherwise with the first co-reference in the series.

Algorithm 2 Breaking Local Dependency

```
L1, Correlated entities candidates
EM, Unified Entities Map
procedure
    for all R ∈ RCMLis do
        RText ← R.Text
        L1 ← Stanford.corefrenceChain(RText)
        for all Cand ∈ L1 do
            opList ← Locate the operands for all references in Cand
            UnifiedEntity ← OP[1]
            for all OP ∈ opList do
                if OP.TechCode ≠ φ then
                    UnifiedEntity ← EM[OP.TechCode].TechEntity
                    break;
                end if
            end for
            for all OP ∈ opList do
                OP ← UnifiedEntity
            end for
        end for
    end for
end procedure
```

5.2 Formal-semantics Mapping

This phase formulates the formal semantics at the predicate level by locating the left hand side (LHS), the right hand side (RHS) and the operator [33]. Mapping a grammatical English role to a static formal semantics is not always correct (e.g., assuming the subject is always placed in the LHS of the formula) because it depends on the semantics of the used verb. Fig. 14 illustrates how LHS matches different grammatical roles.

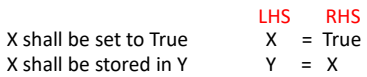


Fig. 14: Same grammatical role "X" have different formal semantic role

A rigorous repository is needed to be responsible for playing the role of formal semantics assignment. We provide a database-based repository (called VerbFrame-DB) for mapping any frame into the standard formal semantics "<LHS, RHS, OP>". VerbFrame is easy to extend with non-mathematical-expert users on the run or pre-execution. This allows the requirements writer to use the language that is most appropriate to the domain, while at the same time declare in precise terms how the writing should be interpreted. Verb-frame has the structure "frameName(Arg_1,Arg_2,..Arg_n)" – each argument has a defined formal semantics based on its position in the list. VerbFrame-DB supports three forms of formal semantics proposed in RCM [33] as indicated in Table 4.

Table 4: VerbFrame to RCM Mapping

Input-Frame Signature	**VerbFrame to RCM Mapping**	**RCM formal-semantic output**
frameName(OP_1,..OP_n)	Process: – frameName ==> processName – frameOperands ==> ProcessOperands	ProcessName(OP_1,OP_2,..OP_n)
	Relational with Plain RHS: – LHS ==> OP_i – RHS ==> OP_j – rel $\{=, \neq, >, <, \leq, \geq\}$	LHS rel RHS
	Relational with Aggregated RHS: – LHS ==> OP_i – RHS. fun ==> user_defined – RHS. ArgList ==> $\{OP_1,..OP_n\}$ - $\{OP_i\}$ – rel $\{=, \neq, >, <, \leq, \geq\}$	LHS rel fun(ArgList)

Fig. 15 shows the RCM components of R1, each enriched with formal semantics after applying Verb-Frame mapping.

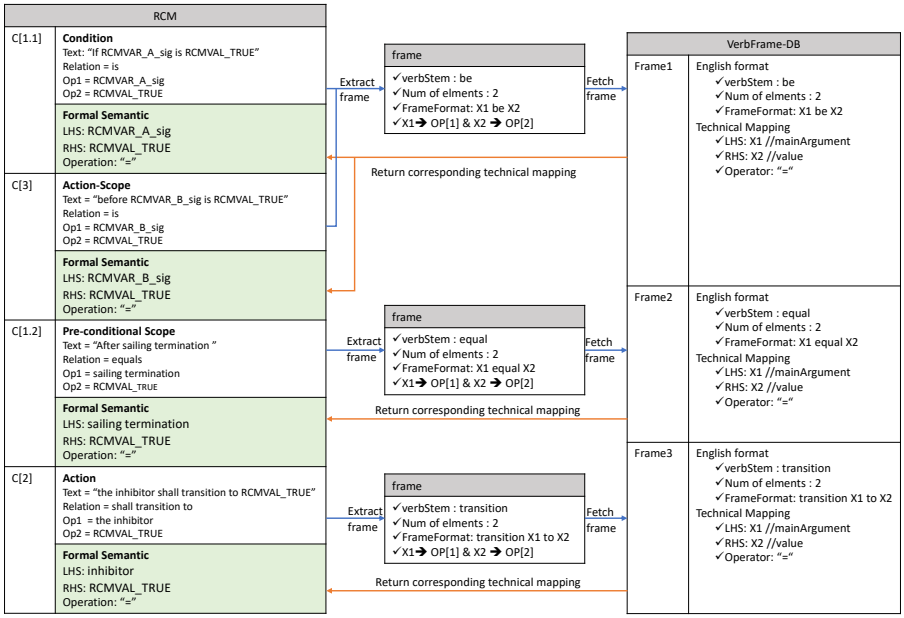


Fig. 15: Formal Semantics association for the RCM output of R1 in Fig. 2.c

6 Evaluation

We evaluated the performance of our RCM-Extraction technique from two perspectives. First, we measure the performance on real requirements (162 requirements sentences). Second, we evaluate the capabilities of the proposed extraction approach on 15,000 natural language requirements sentences automatically synthesised in a controlled manner to cover the complete possible combinations and permutations of the components and sub-components. The key advantages of the complete coverage evaluation are: (1) considering many cases that are not included in the curated requirements, and (2) robust measurements (i.e., the data used for the evaluation is not biased towards a specific pattern causing unreal measurements).

6.1 Experiment-1 Benchmark Dataset

We evaluated the performance of our RCM-Extractor technique in addition to the processing time on 162 behavioral requirements sentences (found in[2]) of critical systems, curated from existing case studies in the literature. 89 of these requirements are used for expressing CNLs, templates and defined formats for representing requirements in different domains considering different writing

[2]Datasets of 162 requirements and the corresponding RCM-Extractor output `https://github.com/ABC-7/RequirementsExtraction/tree/main/RealRequirement-Experiment`

styles in [14] [13][29], [8], [18], [6], [1], [28], [19], [21], [25]. An additional 28 requirements were used for evaluating formalisation approaches in [10,30]. The remaining 45 requirements were extracted from an online available critical-system requirements that are not tweaked for any special use in [11]. These requirements do not contain coordinating relations (i.e, and/or) as we currently do not support coordination in RCM-extractor. However, they cover the entire components and sub-components types –proposed by RCM– with different writing styles and structures. The detailed description of these requirements are presented in Table 5 showing that the source natural language requirements exhibit a wide range of grammatical structures.

Table 5: Dataset Description

Syntactic Characteristics	**Count**	**RCM Properties**	**Count**
Requirements	149	Actions	162
requirement sentences	162	Triggers	44
clauses	337	Conditions	68
-Relative clauses	6	Req-Scope	69
-Subordinating clauses	169	- Pre-conditional Scope	19
-Independent clauses	162	-Action Scope	50
passive voice clauses	105	Valid time	37
imperative clauses	16	-Action Valid time	16
Merged clauses	21	-Condition Valid time	10
-Interleaved clauses	3	-Trigger Valid time	3
-Type-Merged clauses	18	-Action scope Valid time	4
Transitive verbs with one object	114	-Preconditional scope Valid time	4
Transitive verbs with multiple object	16	Condition-Pre-elapsed time	2
Intransitive verbs	107	Action-Pre-elapsed time	16
Time phrases	60	Action-In-between time	5
Fed Technical terms	31	Trigger-In-between time	2
Noun entities	283	Hidden Constraint	6

6.1.1 RCM-Extractor performance

We first processed the 162 requirements with RCM-Extractor (the datasets, DSSAM extraction output, and EnhancedDSSAM extraction output attached with the formal semantics and the automatically generated Metric temporal logic (MTL) formal notation can be found in[3]). Then, we assessed the performance of each process/step of RCM-Extractor as well as the final results against the expected outcomes of the manual extraction (conducted and validated by the authors, where each requirement sentence has a unique extraction ground truth –the manual assessment can be found in[3]). Table 6 presents the manual evaluation measures (Manual-Ev) and the computed measures for the extraction (Recall, Precision, F-measure and Accuracy) on the dataset. We also report the number of true positives (TP), false positives (FP), and false negatives (FN).

[3]DSSAM, EnhancedDSSAM, and HybridDSSAM conducted evaluation sheet `https://github.com/ABC-7/RequirementsExtraction/blob/main/RCM-ExtractorEvaluation-OnRealDatasets.xlsx`

Table 6: Measured performance of the RCM-Extractor technique

Technique	Criteria	Manual-Ev	TP	FP	FN	Recall	Precision	F-measure	Accuracy
ESSGA	Initial components	331	317	5	14	96%	98%	97%	94%
DSSAM	Final components	366	347	3	19	95%	99%	97%	94%
	Rel(sub-components)	407	318	8	89	78%	98%	87%	77%
	Rel(Arguments)	326	318	8	0	100%	98%	99%	98%
	Entire Prim Requirement	162	122	7	33	79%	95%	86%	75%
Enhanced-DSSAM	Final components	366	355	4	11	97%	99%	98%	96%
	Rel(sub-components)	407	374	27	33	92%	93%	93%	86%
	Rel(Arguments)	401	374	27	0	100%	93%	97%	93%
	Entire Prim Requirement	162	135	22	5	96%	86%	91%	83%
Hybrid-DSSAM	Final components	366	359	5	7	98%	99%	98%	97%
	Rel(sub-components)	407	382	22	25	94%	95%	94%	89%
	Rel(Arguments)	404	382	22	0	100%	95%	97%	95%
	Entire Prim Requirement	162	139	20	3	98%	87%	92%	86%

For each requirement sentence, we compute the following:

- Initial components: are the initial set of components extracted by our ESSGA algorithm. As seen in the table, 317 and 14 components out of the expected 331 are correctly extracted and missed by the ESSGA algorithm, respectively. In addition, the five FP components have: 1) incomplete text, 2) excess text, or 3) composition of two components (i.e., incorrectly merged). The main cause for the missed and the wrongly produced components is the incorrect interpretations of StanfordNLP. It is also worth noting that the 14 missed components (FN) also include the five incorrectly extracted ones (i.e., some components are counted as both FP and FN because the extracted component parts are incorrect and the actual component is not extracted).
- Final components: are the final components of the given requirement sentence obtained by our DSSAM versions (after resolving merged cases). RCM-Extractor succeeded in splitting the merged cases (discussed in Sec.4.3.1) and hence increased the initial components to 347, 355, and 359 final components by DSSAM, Enhanced-DSSAM and Hybrid-DSSAM, respectively.
- Rel(sub-components): are the sub-components extracted by Phase 3 in the DSSAM versions, given the extracted components by ESSGA that are provided as input (i.e., the missed components by ESSGA are excluded). DSSAM correctly extracted 318 sub-components out of the 407 ones identified within the correctly extracted 347 components. DSSAM produced eight incomplete sub-components (FP) and failed to produce 82 ones. Enhanced-DSSAM correctly extracted 374 sub-components and missed 33 ones. However, it produced more incomplete sub-components (27 FP). Hybrid-DSSAM has the best performance by extracting 382 correct sub-components, 22 incomplete and just 25 missed ones.
- Rel(Arguments): are the sub-components with correctly extracted arguments by Phase 5 of our DSSAM versions, given the extracted sub-components by Phase 3 (i.e, the missed sub-components are excluded). The correctness of an extracted sub-component indicates the correctness of its initially extracted arguments according to the described process in Sec.4.3.3. Thus, the correctly extracted 318, 374, and 382 sub-components in DSSAM,

Enhanced-DSSAM and Hybrid-DSSAM, respectively, are correctly decomposed into arguments. Nine within the 318 sub-components, containing transitive verbs with multiple objects, in addition to seven sub-components, provided with artificial arguments through the supplementary process, are all decomposed correctly.

- Entire Primitive Requirement: are the extracted primitive requirements reflecting the performance of the entire pipeline (i.e., ESSGA and one DSSAM version). In DSSAM, 122 requirement sentences are correctly extracted, in addition to seven sentences (FP) produced with missing arguments. Another 33 requirement sentences were missed because of the failure of DSSAM at one of its phases. It is also worth noting that the failed sentences contain components and/or sub-components processable by RCM-Extractor. However, the failure in decomposing a given component in the entire requirement into (partially-)correct sub-components by DSSAM causes failure to the entire requirement sentence. Enhanced-DSSAM achieved a better performance by correctly extracting 135 requirement sentences and missing just five sentences. However, it suffers from a larger number of the partially extracted requirements (22 FP). Hybrid-DSSAM achieved the best performance, as expected, by correctly extracting 139 requirements sentences, partially extracting 20 requirements, and missing only three.

Overall, RCM-Extractor achieved 94% accuracy for extracting the initial requirements components by ESSGA and 94%, 96% and 97% for the final components extraction by DSSAM, Enhanced-DSSAM and Hybrid-DSSAM, respectively. In addition, DSSAM achieved 77% accuracy in Rel(Sub-components), 98% accuracy in Rel(Arguments) and 75% accuracy for the entire primitive requirement extraction. **Enhanced-DSSAM achieved better accuracy than DSSAM with 86% Rel(Sub-components) and 83% in the entire primitive requirement extraction. Hybrid-DSSAM showed the best performance achieving 89% Rel(Sub-components) and 86% in the entire primitive requirement extraction.**

DSSAM missed 17 primitive requirements because of the approach limitation in handling excess words. It also partially extracted eight primitive requirements and an additional 16 primitive requirements are also missed because of the StanfordNLP accuracy for POS. In the Enhanced-DSSAM and Hybrid-DSSAM approaches, the failure is because of the accuracy of the Stanford parser. $\approx 5\%$, $\approx 14\%$, and $\approx 12\%$ of the extracted requirement sentences by DSSAM, Enhanced-DSSAM, and Hybrid-DSSAM, respectively, are partially correct.

6.1.2 RCM-Extractor Processing-Time:

RCM-Extractor consumes on average half a second for processing one requirement sentence (on a machine with 2.6 GHz Intel Core i7 8th generation, and 16GB of 2400 MHz DDR4 RAM). Fig. 16 shows the consumed time for processing the 162 requirements sentences. The X-axis and the Y-axis represent

the word count per requirement sentence and the consumed time by the RCM-Extractor in seconds, respectively. The figure shows the time range for processing requirements sentences with the same count of words. In addition, the black line indicates the mean value of the consumed extraction time.

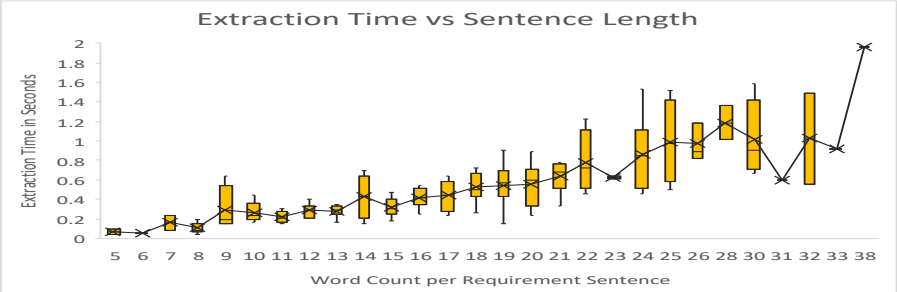


Fig. 16: Extraction time Performance

6.2 Experiment-2 Robust Evaluation

In this experiment, we aim to evaluate the performance of the proposed extraction approach including the different versions of DSSAM to ensure: (1) insensitivity to the count/types of the contributing components and sub-components, and (2) insensitivity to the order (of the sub-components within the same component and the components within the requirement sentence. We fed RCM-Extractor with 15,000 synthesised requirements, representing the complete set of automatically generated requirements based on three parameters: (1) the count of the contributing components and sub-components, (2) their types, and (3) the order (of the sub-components within the same component and the components within the primitive requirement). To compute the minimum required number of requirements sentences covering all the valid cases considering the count, the type and the order we applied two steps. First, we computed the possible combinations of the components and sub-components using the combination equation ${}^nC_0 + {}^nC_1 + {}^nC_2 + ... + {}^nC_n = 2^n$. Second, we computed the permutations of each combination using ${}^nP_r = \frac{n!}{(n-r)!} = n!$, where n = r. Using these computations, we concluded that 14288 requirement sentences can cover the possible cases. We generated these requirements using the synthesised requirements generator proposed in [31], where it provides the generated requirements along with their breakdowns (e.g., components, sub-components and arguments).

6.2.1 Experiment Setup

We fed RCM-Extractor with these synthesised requirements files to assess the extraction performance of ESSGA, DSSAM, Enhanced-DSSAM and Hybrid-DSSAM (i.e., the generated requirements in addition to the corresponding extraction output of DSSAM, EnhancedDSSAM and HybridDSSAM are found in [4]). First, the synthesised requirements along with their breakdowns are populated in RCMs. Then, the text of each synthesised requirement sentence is passed as input to the RCM-Extractor. Next, the constructed RCMs resulting from the different DSSAM extraction versions are automatically compared to the populated RCM from the synthesised breakdowns. In the automatic comparison, the text of the automatically extracted components, sub-components and arguments are checked for equality against the generated ones.

To assess the capabilities of our approach independent of the StanfordNLP limitations, we fed the extraction approach with the normal Stanford typed-dependencies, partially-corrected ones, and fully-corrected ones, each in a separate trial (i.e., the evaluation results are available online in[5]:-

- First trial: the extraction approach is directly fed with the obtained TDs by StanfordNLP.
- Second Trial: a partial correction is applied to the Stanford Typed dependency. Instead of feeding the entire sentence to StanfordNLP, the text slots of the generated components of a sentence are fed in sequence to Stanford, then their corresponding typed dependencies are aggregated to construct a better interpretation (i.e., the accuracy of StanfordNLP increases by decreasing the length of the input text).
- Third Trial: a full correction is obtained for the typed dependency by feeding StanfordNLP with the sub-components' text slots of the generated sentence in sequence, then their corresponding typed dependencies are aggregated to construct correct interpretations.

6.2.2 Experiment Result

Table 7 shows the performance of the ESSGA, DSSAM, Enhanced-DSSAM and Hybrid-DSSAM on the three trials each with our 15,000 synthesised requirements. The table shows that by feeding the extraction approach with correct analysis (typed-dependency), it achieves 100% accuracy which means that the approach is insensitive to the count, types and order of the contributing components and/or sub-components. It can also be seen that the overall performance improved by enhancing the quality of the input analysis. **In the second trial, DSSAM, Enhanced-DSSAM and Hybrid-DSSAM achieved 83%, 84%, and 92%, respectively, on the entire requirement**

[4]Synthesised requirements and the corresponding RCM-Extraction output: `https://github.com/ABC-7/RequirementsExtraction/tree/main/Synthesised-Experiment`

[5]Synthesised requirements results: `https://github.com/ABC-7/RequirementsExtraction/blob/main/RCM-ExtractorEvaluation-OnSynthesiedDatasets.xlsx`

extraction which is better than the first trial (where they achieved 54%, 54%, and 57%, respectively). The third trial is the best where a score of 100% is achieved for all the versions.

RCM-Extractor still performs well with the direct StanfordNLP analysis by achieving 70% in the initial components extraction by ESSGA and 64%, 69% and 72% in the sub-components extraction by DSSAM, Enhanced-DSSAM and Hybrid-DSSAM, respectively. The DSSAM versions also achieved 54%, 54% and 57% for the entire requirements sentences extraction, respectively. In which, **Enhanced-DSSAM extracts 36 more requirement sentences than DSSAM**. In addition, by merging both techniques, **Hybrid-DSSAM shows the best results with difference around 3% indicating that DSSAM is not fully replaceable by EnhancedDSSAM, as each may outperform the other in some scenarios, but both together can achieve the best results.**

Table 7: Measured performance of the RCM-Extractor technique

	Perspective	**Criteria**	**Total**	**T**	**F**	**Accuracy**
No Correction	ESSGA	Extracted Components	46269	32366	13903	70%
	DSSAM	Abs(sub-components)	111961	71722	40239	64%
		Abs(Arguments)	424149	269228	154921	63%
		Entire Prim Requirement	15000	8086	6914	54%
	Enhanced-DSSAM	Abs(sub-components)	111961	77160	34801	69%
		Abs(Arguments)	424149	289316	134833	68%
		Entire Prim Requirement	15000	8122	6878	54%
	Hybrid-DSSAM	Abs(sub-components)	111961	80227	31734	72%
		Abs(Arguments)	424149	299563	124586	71%
		Entire Prim Requirement	15000	8493	6507	57%
Partial Correction	ESSGA	Extracted Components	46269	43659	2610	94%
	DSSAM	Abs(sub-components)	111961	103117	8844	92%
		Abs(Arguments)	424149	390640	33509	92%
		Entire Prim Requirement	15000	12521	2479	83%
	Enhanced-DSSAM	Abs(sub-components)	111961	107112	4849	96%
		Abs(Arguments)	424149	406396	17753	96%
		Entire Prim Requirement	15000	12527	2473	84%
	Hybrid-DSSAM	Abs(sub-components)	111961	108971	2990	97%
		Abs(Arguments)	424149	413525	10624	97%
		Entire Prim Requirement	15000	13833	1167	92%
Full Correction	ESSGA	Initial components	46269	46269	0	100%
	DSSAM	Abs(sub-components)	111961	111961	0	100%
		Abs(Arguments)	424149	424149	0	100%
		Entire Prim Requirement	15000	15000	0	100%
	Enhanced-DSSAM	Abs(sub-components)	111961	111961	0	100%
		Abs(Arguments)	424149	424149	0	100%
		Entire Prim Requirement	15000	15000	0	100%
	Hybrid-DSSAM	Abs(sub-components)	111961	111961	0	100%
		Abs(Arguments)	424149	424149	0	100%
		Entire Prim Requirement	15000	15000	0	100%

6.3 Discussion

Efficient requirements formalisation: our RCM-Extractor approach can reduce the overhead effort and time needed to formalise NL requirements. It is not affected by the ordering of the key elements of interest nor their count. The approach can also handle a comprehensive range of various requirements structures compared to the other existing approaches. As it is also domain independent, it can be augmented with any domain knowledge (i.e., additional domain words).

Extraction limitation: RCM-Extractor can only support the timing specified as prepositional phrases with closed structures of quantifying relations. In addition, it currently does not support the coordination relation (e.g., "If the Status attribute of the Lower Desired Temperature **or** the Upper Desired Temperature equals Invalid, the Regulator Interface Failure shall be set to True" [10]). We are planning on extending the approach to support this relation.

Extraction accuracy depends on the quality of StanfordNLP: StanfordNLP is a statistical-based approach. A common issue with a statistical NLP approach is the prospect of producing incorrect results (e.g., the sentence "the display elements glow" is analysed by StanfordNLP as NP). Consequently, it is possible to have a sentence incorrectly parsed and analysed in the proposed approach.

Within the conducted experiments, we discovered that, StanfordNLP is not only sensitive to the length of the input text being analysed (a widely known problem), but it is also sensitive to the order of the clauses in case of a complex sentence type. For example, StanfordNLP correctly analysed the requirement sentence "when the sailing request whose index exceeds 9 is above the standstill request flag, the standstill request flag shall be sent to the engine error within 25 seconds every 4 seconds". However, it provides incorrect analysis for the same sentence just by reordering the clauses as "the standstill request flag shall be sent to the engine error within 25 seconds every 4 seconds when the sailing request whose index exceeds 9 is above the standstill request flag".

7 Conclusion

In this paper, we introduced RCM-Extractor - an automated approach to extract and transform textual requirements into intermediate representation, RCM, which is expressive enough to be transformed into formal notations. Our approach is domain independent and insensitive to the structure and format of the input textual requirements. We evaluated our approach on 162 real requirements sentences achieving rates of 87% precision, 98% recall, 92% F-measure and 86% accuracy. Additionally, we utilised 15,000 synthesised requirements comprising the possible combinations and arrangements of the requirements properties and the approach achieved 57% accuracy (the drop in accuracy is

mainly because of the StanfordNLP analysis issues as indicated in the evaluation section). However, RCM-Extractor achieved 100% accuracy when the StanfordNLP analysis are fully-corrected showing the capabilities of the approach in processing requirements with any count/type of the properties in any of the possible arrangements.

Acknowledgements

Grundy is supported by ARC Laureate Fellowship FL190100035. Ismail and Osama are supported by Deakin University School of IT PhD scholarships.

References

1. Bitsch, F.: Safety patterns—the key to formal specification of safety requirements. In: International Conference on Computer Safety, Reliability, and Security, pp. 176–189. Springer (2001)
2. Brunello, A., Montanari, A., Reynolds, M.: Synthesis of ltl formulas from natural language texts: State of the art and research directions. In: 26th International Symposium on Temporal Representation and Reasoning (TIME 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
3. Buzhinsky, I.: Formalization of natural language requirements into temporal logics: a survey. In: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), vol. 1, pp. 400–406. IEEE (2019)
4. Buzhinsky, I.: Formalization of natural language requirements into temporal logics: a survey pp. 400–406 (2019)
5. Das, B., Majumder, M., Phadikar, S.: A novel system for generating simple sentences from complex and compound sentences. International Journal of Modern Education and Computer Science **11**(1), 57 (2018)
6. Dick, J., Hull, E., Jackson, K.: Requirements engineering. Springer (2017)
7. Ferrari, A., Spagnolo, G.O., Gnesi, S.: Pure: A dataset of public requirements documents. In: 2017 IEEE 25th International Requirements Engineering Conference (RE), pp. 502–505. IEEE (2017)
8. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: Spear v2. 0: Formalized past ltl specification and analysis of requirements. In: NASA Formal Methods Symposium, pp. 420–426. Springer (2017)
9. Fu, R., Bao, X., Zhao, T.: Generic safety requirements description templates for the embedded software. In: 2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN), pp. 1477–1481. IEEE (2017)
10. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: Arsenal: automatic requirements specification extraction from natural language. In: NASA Formal Methods Symposium, pp. 41–46. Springer (2016)
11. Houdek, F.: System requirements specification automotive system cluster(elc and acc). Technical University of Munich (2013)
12. Huddleston, R., Pullum, G.: The cambridge grammar of the english language. Zeitschrift für Anglistik und Amerikanistik **53**(2), 193–194 (2005)
13. Jeannet, B., Gaucher, F.: Debugging embedded systems requirements with stimulus: an automotive case-study (2016)
14. Justice, B.: Natural language specifications for safety-critical systems. Master's thesis, Carl von Ossietzky Universität (2013)
15. Kamp, H.: A theory of truth and semantic representation. Formal semantics-the essential readings pp. 189–222 (1981)
16. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering, pp. 372–381. ACM (2005)

17. Leech, G., Deuchar, M., Hoogenraad, R.: English Grammar for Today. Palgrave Macmillan UK (1982). DOI 10.1007/978-1-349-16878-1. URL https://doi.org/10.1007%2F978-1-349-16878-1
18. Lúcio, L., Rahman, S., bin Abid, S., Mavin, A.: Ears-ctrl: Generating controllers for dummies. In: MODELS (Satellite Events), pp. 566–570 (2017)
19. Lúcio, L., Rahman, S., Cheng, C.H., Mavin, A.: Just formal enough? automated analysis of ears requirements. In: NASA Formal Methods Symposium, pp. 427–434. Springer (2017)
20. Marko, N., Leitner, A., Herbst, B., Wallner, A.: Combining xtext and oslc for integrated model-based requirements engineering. In: 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, pp. 143–150. IEEE (2015)
21. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (ears). In: Requirements Engineering Conference, 2009. RE'09. 17th IEEE International, pp. 317–322. IEEE (2009)
22. Nelken, R., Francez, N.: Automatic translation of natural language system specifications into temporal logic. In: International Conference on Computer Aided Verification, pp. 360–371. Springer (1996)
23. Osama, M., Zaki-Ismail, A., Abdelrazek, M., Grundy, J., Ibrahim, A.: Score-based automatic detection and resolution of syntactic ambiguity in natural language requirements. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 651–661. IEEE (2020)
24. Pohl, K., Rupp, C.: Requirements Engineering Fundamentals. Rocky Nook (2011)
25. R. S. Fuchs, N.E.: Attempto controlled english (ace). In: CLAW 96, First International Workshop on Controlled Language Applications. Katholieke Universiteit, Leuven (1996)
26. Sládeková, V.: Methods used for requirements engineering. Master's thesis, Univerzity Komenského (2007)
27. Sturla, G.: A two-phased approach for natural language parsing into formal logic. Ph.D. thesis, Massachusetts Institute of Technology (2017)
28. Teige, T., Bienmüller, T., Holberg, H.J.: Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. pp. 6–9. MBMV (2016)
29. Thyssen, J., Hummel, B.: Behavioral specification of reactive systems using stream-based i/o tables. Software & Systems Modeling **12**(2), 265–283 (2013)
30. Yan, R., Cheng, C.H., Chai, Y.: Formal consistency checking over specifications in natural languages. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1677–1682. IEEE (2015)
31. Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., Ibrahim, A.: Corg: A component-oriented synthetic textual requirements generator. In: Requirements Engineering: Foundation for Software Quality: 27th International Working Conference, REFSQ 2021, Essen, Germany, April 12–15, 2021, Proceedings. Springer Nature
32. Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., Ibrahim, A.: Rcm-extractor: Automated extraction of a semi formal representation model from natural language requirements. In: MODELSWARD 2021-9th International Conference on Model-Driven Engineering and Software Development
33. Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., Ibrahim, A.: Rcm: Requirement capturing model for automated requirements formalisation (2020)
34. Zaki-Ismail., A., Osama., M., Abdelrazek., M., Grundy., J., Ibrahim., A.: Requirements formality levels analysis and transformation of formal notations into semi-formal and informal notations. In: SEKE (2021)
35. Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K.J., Ajagbe, M.A., Chioasca, E.V., Batista-Navarro, R.T.: Natural language processing (nlp) for requirements engineering: A systematic mapping study. arXiv preprint arXiv:2004.01099 (2020)