9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD2021), 8-10 Feb 2021, online

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

Aya Zaki-Ismail[1][a], Mohamed Osama[1][b], Mohamed Abdelrazek[1][c], John Grundy[2][d] and Amani Ibrahim[1][e]

[1]*Information Technology Institute, Deakin University, 3125 Burwood Hwy, VIC, Australia*

[2]*Information Technology Institute, Monash University, 3800 Wellington Rd, VIC, Australia*

{*amohamedzakiism, mdarweish, mohamed.abdelrazek, amani.ibrahim*}*@deakin.edu.au, john.grundy@monash.edu*

Keywords: Natural-Language Extraction, Requirements Formalization

Abstract: Formal verification requires system requirements to be specified in formal notations. Formalisation of system requirements manually is a time-consuming and error-prone process, and requires engineers to have strong mathematical and domain expertise. Most existing requirements formalisation techniques assume requirements to be specified in pre-defined templates and these techniques employ pre-defined transformation rules to transform requirements specified in the predefined templates to formal notations. These techniques tend to have limited expressiveness and more importantly require system engineers to re-write their system requirements following these templates. In this paper, we introduces an automated extraction technique (RCM-Extractor) to extract the key constructs of a comprehensive and formalisable semi-formal representation model from textual requirements. We have evaluated our RCM-Extractor on a dataset of 162 requirements curated from the literature. RCM-Extractor achieved 95% precision, 79% recall, 86% F-measure and 75% accuracy.

1 INTRODUCTION

Formal verification techniques - e.g. model checking - are usually highly recommended, and in many cases mandatory, when proving the correctness of a given mission critical system or system component [Buzhinsky, 2019a]. To benefit from these formal verification techniques, system requirements need to be specified in suitable formal notations, such as temporal logic (TL) [Buzhinsky, 2019a].

According to the recently conducted reviews [Brunello et al., 2019] and [Buzhinsky, 2019b], there is still a need for an approach that can automatically extract and transform existing textual requirements written using different structures and formats – i.e. no pre-defined templates – into formal notations. To achieve this goal, we transform NL-requirements to a well-defined semi-formal representation model that is mappable (using well-defined transformation rules) into formal notations.

Our previous work in [Zaki-Ismail et al., 2020] proposes a comprehensive intermediate representation for critical system requirements - RCM: Requirement Capturing Model. The model defines a comprehensive list of key requirement properties. In addition, the model is transferable to temporal logic using transformation rules. We also provide the mapping from RCM into Metric Temporal Logic (MTL) and Computational Tree Logic (CTL).

In this paper, we introduce our requirement extraction technique (RCM-Extractor) that can process textual requirements and produce RCM representation of these requirements. Then using the rules defined by [Zaki-Ismail et al., 2020], we can generate formal notation. Our approach does not require textual requirements to follow specific structure, format, style, order or length. We evaluated our technique on a dataset of 162 requirements sentences curated from papers in the literature and online sources as well.

Fig.1 presents the entire process of transforming a set of textual requirements into TL notations. Fig.1 (a) shows a set of 5 input textual requirements. Fig.1 (b) shows a simplified flow of our RCM-Extractor to transform example one requirement (R1) into RCM representation in Fig.1 (c). Fig.1 (d) provides the TL formula to be produced by the formalisation rules ap-

[a] https://orcid.org/0000-0000-0000-0000
[b] https://orcid.org/0000-0000-0000-0000
[c] https://orcid.org/0000-0000-0000-0000
[d] https://orcid.org/0000-0000-0000-0000
[e] https://orcid.org/0000-0000-0000-0000

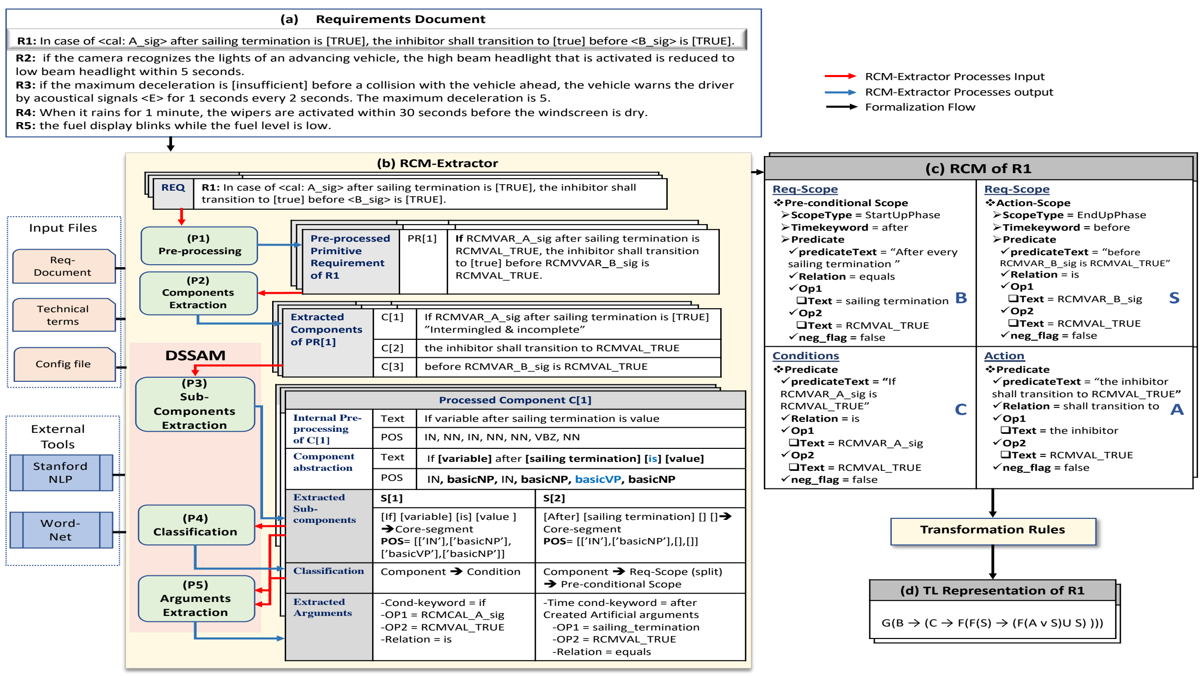

(a) Requirements Document
R1: In case of <cal: A_sig> after sailing termination is [TRUE], the inhibitor shall transition to [true] before <B_sig> is [TRUE].
R2: if the camera recognizes the lights of an advancing vehicle, the high beam headlight that is activated is reduced to low beam headlight within 5 seconds.
R3: if the maximum deceleration is [insufficient] before a collision with the vehicle ahead, the vehicle warns the driver by acoustical signals <E> for 1 seconds every 2 seconds. The maximum deceleration is 5.
R4: When it rains for 1 minute, the wipers are activated within 30 seconds before the windscreen is dry.
R5: the fuel display blinks while the fuel level is low.
RCM-Extractor Processes Input
RCM-Extractor Processes output
Formalization Flow
Input Files
Req-Document
Technical terms
Config file
External Tools
Stanford NLP
Word-Net
(b) RCM-Extractor
REQ
R1: In case of <cal: A_sig> after sailing termination is [TRUE], the inhibitor shall transition to [true] before <B_sig> is [TRUE].
(P1) Pre-processing
(P2) Components Extraction
DSSAM
(P3) Sub-Components Extraction
(P4) Classification
(P5) Arguments Extraction
Pre-processed Primitive Requirement of R1
PR[1]
If RCMVAR_A_sig after sailing termination is RCMVAL_TRUE, the inhibitor shall transition to [true] before RCMVVAR_B_sig is RCMVAL_TRUE.
Extracted Components of PR[1]
C[1] If RCMVAR_A_sig after sailing termination is [TRUE] ”Intermingled & incomplete”
C[2] the inhibitor shall transition to RCMVAL_TRUE
C[3] before RCMVAR_B_sig is RCMVAL_TRUE
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]
[If] [variable] [is] [value] ➔Core-segment POS= [['IN'],['basicNP'], ['basicVP'],['basicNP']]
S[2]
[After] [sailing termination] [] []➔ Core-segment POS= [['IN'],['basicNP'],[],[]]
Classification
Component ➔ Condition
Component ➔ Req-Scope (split) ➔ Pre-conditional Scope
Extracted Arguments
-Cond-keyword = if
-OP1 = RCMCAL_A_sig
-OP2 = RCMVAL_TRUE
-Relation = is
-Time cond-keyword = after
Created Artificial arguments
-OP1 = sailing_termination
-OP2 = RCMVAL_TRUE
-Relation = equals
(c) RCM of R1
Req-Scope
❖Pre-conditional Scope
➢ScopeType = StartUpPhase
➢Timekeyword = after
➢Predicate
✓predicateText = “After every sailing termination ”
✓Relation = equals
✓Op1
❑Text = sailing termination
✓Op2
❑Text = RCMVAL_TRUE
✓neg_flag = false
B
Req-Scope
❖Action-Scope
➢ScopeType = EndUpPhase
➢Timekeyword = before
➢Predicate
✓predicateText = “before RCMVAR_B_sig is RCMVAL_TRUE”
✓Relation = is
✓Op1
❑Text = RCMVAR_B_sig
✓Op2
❑Text = RCMVAL_TRUE
✓neg_flag = false
S
Conditions
❖Predicate
✓predicateText = “If RCMVAR_A_sig is RCMVAL_TRUE”
✓Relation = is
✓Op1
❑Text = RCMVAR_A_sig
✓Op2
❑Text = RCMVAL_TRUE
✓neg_flag = false
C
Action
❖Predicate
✓predicateText = “the inhibitor shall transition to RCMVAL_TRUE”
✓Relation = shall transition to
✓Op1
❑Text = the inhibitor
✓Op2
❑Text = RCMVAL_TRUE
✓neg_flag = false
A
Transformation Rules
(d) TL Representation of R1
G(B → (C → F(F(S) → (F(A v S)U S))))

Figure 1: Textual Requirement Extraction Example

plied on the extracted RCM of R1, where B, C, S, and A are proposition variables corresponding to ”sailing termination being happened”, ”<cal: A_sig> is [True]”, ”$< B\_sig >$ is [TRUE]”, and ”the inhibitor shall transition to [true]”, respectively.

2 REQUIREMENTS CAPTURING MODEL (RCM)

In RCM [Zaki-Ismail et al., 2020], each system requirement R_i, may have one or more primitive requirements PR where $\{R_i = < PR_n >$ and n>0$\}$. Each PR_j represents only one sentence. Each primitive requirement contains zero or more conditions, zero or more triggers, zero or more Req-scope, and one or more actions. Figure 2 shows a simplified representation of RCM. Below we provide a description of each of these components using the example requirements in Figure 1:

- **Trigger**; holds 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.1).
- **Condition**; represents the constraints that should be explicitly checked by the system before executing one or more actions (e.g., if the maximum deceleration is [insufficient] in R3 Fig. 1).
- **Req-scope**; determines the operational context under which (i) ”condition(s) and trigger(s)” can be valid – called a pre-conditional scope; or (ii) ”action(s)” can occur – called an action scope. The Scope may define a start boundary (e.g., ”after sailing termination”), end boundary (e.g., ”before $< B\_sig >$ is [TRUE]” in R1 Fig.1), or both (e.g., ”while R is true” can be expressed by after and until as ”after R is true” and ”until not R”).

A component can be broken into sub-components. The RCM uses 5 types of sub-components:

- **Core-Segment**: Each requirement component must have one Core-Segment. It expresses the core part of the component including: the operands, the operator and 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**: an optional sub-component 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 hold for 1 second length of time).
- **Hidden constraint**: holds an explicit 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 ac-*

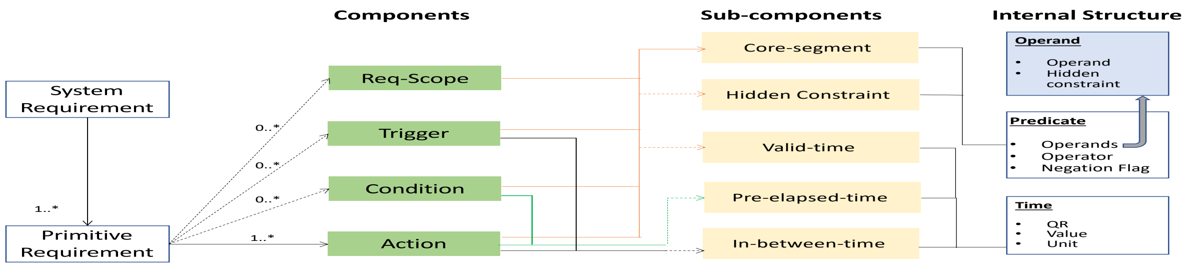


Figure 2: Compact Meta-model of the RCM

Dashed arrow: optional, solid arrow: mandatory, line-segment: internal representation, green box: component and yellow box: sub-components

tivated is a constraint defined for the operand *the high beam headlight*. RCM stores the constraint inside its related operand object.

- **Pre-elapsed-time:** can be found for action and condition components. This indicates the length of time from an offset point before the action to start or the condition to be checked (e.g., ”the wipers are activated **within 30 seconds**” in R4).
- **In-between-time**: associated with action and trigger components and used to reflect the elapsed time between consecutive occurrences of the same event (e.g., ”the vehicle warns the driver by acoustical signal for 1 seconds **every 2 seconds**” in R3).

3 RCM-EXTRACTOR APPROACH

RCM-Extractor consists of five main phases as in Figure 1. We use StanfordNLP to extract Parts-of-Speech (POS), Typed-Dependencies (TD) and parsing tree. TDs are set of mentions representing syntactic relations among the sentences words (e.g., in ”nsubj(equals-14, X-13)”, ”X” is subject to the verb ”equals”). POS provide the syntactic type of each word (e.g., noun, verb, .etc). We use WordNet to confirm the Stanford POS correction (i.e., the word tagged as a verb by Stanford is actual verb in WordNet). The approach accepts the following as input: (1) Requirements document – a text file containing system requirements in natural language, and (2) Technical terms – a file containing domain-specific terms expressed in English.

3.1 Phase 1: Pre-processing

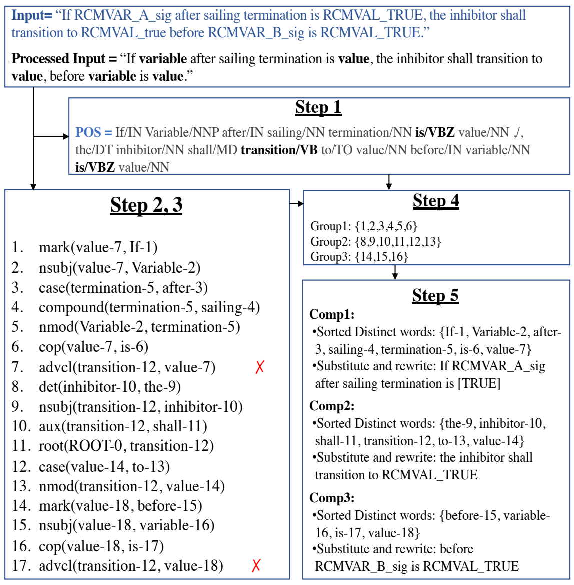
The requirement pre-processing process aims to improve the accuracy of the extraction as follows:

1. **Requirements cleaning:** multiple spaces and defected spaces at the beginning or at the end of the sentence are removed. Other styling formats are also considered (e.g., ”-” replaced with ”_”).
2. **Closed words unification:** English has closed word classes [Leech et al., 1982]. Each class contains a finite set of words with a defined function. In this step, all English words holding the same function (*e.g.*, Subordinating conditional keyword, timing keyword, instant timing keyword) 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} replaced with ”when”). For example, in Fig.1.P1 ”In case of” is replaced with ”if”.
3. **Foreign words substitution:** the extraction approach is supported by the Stanford Parser, which is specified and trained on specific dataset and has a percentage of error. When the Stanford parser is feed with non-English words, it usually produces incorrect results. To avoid this problem, we replace any identified non-English words (i.e., technical variables, technical terms and technical values) with English words representing them (e.g., variable, term and value).

3.2 Phase 2: Components Extraction

In this process, we extract the RCM requirement components – action, Req-scope, trigger or condition – from each primitive natural language requirement.

We base our component extraction algorithm - ESSAG - on SSGA [Das et al., 2018]. Figure 3 outlines the steps used to extract requirement R1 in figure 1. In **step1**, the elements identifying the clauses -verbs in our case- are marked. First, we get the POS of the given sentence using Stanford and then, highlight the main verbs. We confirm the correction of the highlighted verbs using WordNet –defected cases are removed. In **step2**, we compute the typed-dependency of the input sentence using SnafordNLP. Then, in **step3**, we break the connection between clauses be removing the same mentions connecting clause used by SSGA, but we excluded mentions reflecting important domain relations (e.g., "mark", "CComp", "ref", "dep"), removed mentions are marked with "" in figure 1. Afterwards, in **step4**, all the mentions having direct or indirect connections with each identified element are grouped. Finally, in **step5**, distinct words of each group are sorted according to their occurrence indices in the original sentence formulating one component.



Input= "If RCMVAR_A_sig after sailing termination is RCMVAL_TRUE, the inhibitor shall transition to RCMVAL_true before RCMVAR_B_sig is RCMVAL_TRUE."

Processed Input = "If **variable** after sailing termination is **value**, the inhibitor shall transition to **value**, before **variable** is **value**."

Step 1

POS = If/IN Variable/NNP after/IN sailing/NN termination/NN **is/VBZ** value/NN ,/, the/DT inhibitor/NN shall/MD **transition/VB** to/TO value/NN before/IN variable/NN **is/VBZ** value/NN

Step 2, 3

1. mark(value-7, If-1)
2. nsubj(value-7, Variable-2)
3. case(termination-5, after-3)
4. compound(termination-5, sailing-4)
5. nmod(Variable-2, termination-5)
6. cop(value-7, is-6)
7. advcl(transition-12, value-7) ✗
8. det(inhibitor-10, the-9)
9. nsubj(transition-12, inhibitor-10)
10. aux(transition-12, shall-11)
11. root(ROOT-0, transition-12)
12. case(value-14, to-13)
13. nmod(transition-12, value-14)
14. mark(value-18, before-15)
15. nsubj(value-18, variable-16)
16. cop(value-18, is-17)
17. advcl(transition-12, value-18) ✗

Step 4

Group1: {1,2,3,4,5,6}
Group2: {8,9,10,11,12,13}
Group3: {14,15,16}

Step 5

Comp1:
•Sorted Distinct words: {If-1, Variable-2, after-3, sailing-4, termination-5, is-6, value-7}
•Substitute and rewrite: If RCMVAR_A_sig after sailing termination is [TRUE]

Comp2:
•Sorted Distinct words: {the-9, inhibitor-10, shall-11, transition-12, to-13, value-14}
•Substitute and rewrite: the inhibitor shall transition to RCMVAL_TRUE

Comp3:
•Sorted Distinct words: {before-15, variable-16, is-17, value-18}
•Substitute and rewrite: before RCMVAR_B_sig is RCMVAL_TRUE

Figure 3: Components Extraction Example

Requirements expressed in NL causes several complex challenges. Requirement components may (1) exist in any order in the sentence (e.g., R3.S1 and R4 in Fig.1 show alternative orders); (2) may be intermingled (e.g., "In case of $< cal : A_sig >$after sailing termination is [TRUE]" in R1); (3) exist in alternative structures (e.g., simple sentence and complex sentence) that could be represented by alternative types (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 one on its own, step2 and step4. Such isolation makes the approach insensitive to intermingling components, their order, and sentence complexity.

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 "the sailing termination happens"). This case is only eligible to adverbal clauses [Huddleston and Pullum, 2005], where the clause involves a time conditional keyword and that corresponds to the Req-scope component type. Such incompleteness may cause components merge especially if the missed part is the verb (e.g., before termination).

3.3 Deep Syntactic and Semantic Analysis (DSSAM)

DSSAM is responsible for deeply understanding each component to complete the extraction. It consists of the remaining three processes: sub-components extraction, classification, and arguments extraction.

3.3.1 Phase 3: Sub-components Extraction

In Phase 3, the first part of DSSAM, each component is processed to extract the sub-components that comprise it. Figure 1 shows two extracted sub-components (S[1] and S[2]) along with their POS tags.

This process 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 indicated in the example in Fig.1 ("basicNP" for noun phrases and "basicVP for verb phrases). This is carried out via regular expressions that match the POS tags of the tokens within a component against a hand crafted set of 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 (i.e., the succeeding set of words to fulfil a correct grammatical structure for the sub-component) as in Fig.4. Some sub-components (e.g., the core-segment reflecting condition, trigger, or Req-scope) have a keyword head as a result of the preprocessing unification step (e.g., the head of a trigger is "when"). The heads of other sub-components (e.g., 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").

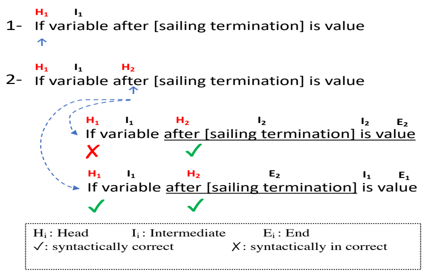

H₁ I₁
1- If variable after [sailing termination] is value
H₁ I₁ H₂
2- If variable after [sailing termination] is value
H₁ I₁ H₂ I₂ I₂ E₂
If variable after [sailing termination] is value
✗ ✓
H₁ I₁ H₂ E₂ I₁ E₁
If variable after [sailing termination] is value
✓ ✓
Hᵢ : Head
Iᵢ : Intermediate
Eᵢ : End
✓: syntactically correct
✗: syntactically in correct

Figure 4: Best suitable sub-components decomposition

However, identifying the body of a sub-component is more challenging because it can have different structures in English clauses. To overcome this, we created a set of hand crafted reasoning rules to reflect the possible structures of each sub-component. We also developed a recursive technique to figure out 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.4 starting at "H_2" conforms to two possible structures, but only one of them will prevent syntactic defects to the other sub-component. The reasoning rules are developed on Prolog to benefit from the inference backtracking nature.

3.4 Phase 4: Classification

The aim of this sub-process is assigning label/type (e.g., Trigger, condition, action, Req-scope, Factual Rule) to each of the extracted components and (pre-elapsed-time, valid-time, in-between-time, hidden constraints) to sub-components. The classifier assigns types by applying two-level checking on the obtained sub-components. The first level, as indicated in Fig.5, identifies types based on four attributes: (1) the head of the given sub-component, (2) comp count: is the total count of the extracted components of the current prim-requirement, (3) the count of the extracted core-segment sub-components. It worth noting that, sub-components heads are unified through step3:*closed words unification* in the pre-processing phase. In addition, the classification is done in comply to the types of clauses [Huddleston and Pullum, 2005], where (1) independent clauses identified through "No Head", (2) subordinating clause identified through: "conditional, instant-conditional, and time-conditional heads", and (3) relative clause identified through "Relative head". Regarding the coordinating clause, first, it is 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 is ready to undergo the classification process.

In the second level, Req-scope component type is classified further to either "action scope" or "pre-conditional scope". The type is identified based on the surrounding components. If Req-scope is found in a merged component, its type will be identified based on the component it is merged with (i.e., if action => action scope, else ==> Pre-conditional scope). Otherwise, we rely on the nearest non-Req-scope component within the primitive requirement as in Table 1.

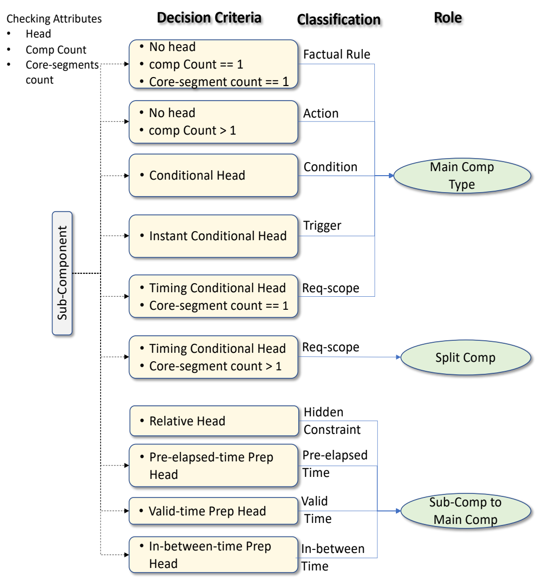


Figure 5: Classification Checking

Table 1: Req-Scope Classification examples

	Req Example	Req-scope type
(a)	1. **Before termination**, if X is [*True*], Y shall be set to [*True*] 2. **Before termination**, Y shall be set to [*True*], if X is [*True*]	1. Pre-conditional Scope 2. Action Scope
(b)	1. If X is [*True*], Y shall be set to [*True*] **before termination** 2. Y shall be set to [*True*], if X is [*True*] **before termination**	1. Action Scope 2. Pre-Conditional Scope
(c)	1. If X is [*True*] **before termination**, Y shall be set to [*True*] 2. Y shall be set to [*True*] **before termination**, if X is [*True*]	1. Pre-Conditional Scope 2. Action Scope
(d)	1. If X is [*True*], **before termination**, Y shall be set to [*True*]	1. Pre-Conditional Scope

3.5 Phase 5: Arguments Extraction

This process identifies the complete arguments set of a given sub-component. To achieve this, we benefit from the sub-component extraction process, to construct 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.1 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: time-conditional keyword, subj, verb, obj (e.g., [After][X][transitions to][True])

Hidden constraint: relative-noun, relative-pronoun, subj, verb, obj (e.g., [X][whose][index][exceeds][2])

Time: time head, quantifying relation, value, unit (e.g., [for] [at most] [2] [seconds]). This depicts 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 element(s) are left empty to maintain the proper ordering and roles of the expected elements within the list.

4 EVALUATION

We evaluated the performance of our RCM-Extractor technique on 162 behavioral requirements sentences (found in[1]) of critical systems, curated from existing case studies in the literature. In which, 89 of these requirements used for expressing proposed CNLs, templates and defined formats for representing requirements in different domains considering different writing styles in [Justice, 2013] [Jeannet and Gaucher, 2016] [Thyssen and Hummel, 2013], [Fifarek et al., 2017], [Lúcio et al., 2017a], [Dick et al., 2017], [Bitsch, 2001], [Teige et al., 2016], [Lúcio et al., 2017b], [Mavin et al., 2009], [R. S. Fuchs, 1996]. Additional 28 requirements were used for evaluating formalization approaches in [Ghosh et al., 2016, Yan et al., 2015]. Further, the remaining 45 requirements were extracted from an online available critical-system requirements that are not tweaked for any special use in [Houdek, 2013]. These requirements do not contain coordinating relations (i.e, and/or) as we currently do not support coordination in the RCM-extractor. However, they cover the entire components and sub-components types –proposed by the RCM– with different writing styles.

We first processed the 162 requirements with the RCM-Extractor (i.e., the extraction output found in[1]). Then, we assess the performance of each process/step of the RCM-Extractor as well as the final results against the expected outcomes of the manual extraction (conducted and agreed by the authors, where each requirement sentence has a unique ground truth of extraction –the manual assessment found in[1]). Table 2 presents the manual evaluation measures and the computed measures for the extraction (i.e., recall, precision, f-measure and accuracy) on the dataset. The assessment criteria are:

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 respectively by the ESSGA Algorithm. In addition, the remaining 5 components (FP) are: 1)with incomplete text, 2)with excess text, or 3) composition of two components (i.e., un-illegibly merged). The main cause for the missed and the wrongly produced components is the miss-interpretation by Stanford. It also worth noting that, not all of the 14 components are missed, but each un-illegibly merged components in FP increases the FN (FN = the expected outcome - TP).

Final components: are the final components of the given requirement sentence obtained by our DSSAM process (after resolving merged cases). The RCM-Extractor succeeded in dissolving the merged cases (Sec.3.3.1) by increasing the initial components to 347. In addition, the decrease in the FP from 5 (in the initial-count) to 3 (in the final count), indicates that, the un-illegibly merged components due to Stanford interpretations are also resolved in this stage.

Rel(sub-components): are the sub-components extracted by Phase 3 in our DSSAM process, given the extracted components by ESSGA that are provided as input(i.e., missed components by ESSGA are excluded). The DSSAM correctly extracted 318 sub-component out of 407 one's derived from the correctly/partially extracted 347 component. The DSSAM produced 8 incomplete sub-components (FP) and failed to produce 82 ones.

Rel(Arguments): are the sub-components with correctly extracted arguments by Phase 5 of our DSSAM process, given the extracted sub-components by P3 and P4 (i.e, missed sub-components are excluded). The correctness of an extracted sub-component indicate the correctness of its initially extracted arguments according to the described process in Sec.3.5. Thus, the correctly extracted 318 sub-component correctly decomposed into arguments.

Entire Primitive Requirement: are the extracted primitive requirements reflecting the performance of the entire pipeline. The table shows that, 122 requirement sentence are correctly extracted in addition to 7 sentences (FP) are produced with missed arguments. On the other hand, 33 requirement sen-

[1]Dataset, RCM-Extractor output, and evaluation sheet: https://github.com/ABC-7/RCM-Extractor

Table 2: Measured performance of the RCM-Extractor Technique

Perspective	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	82	80%	98%	88%	78%
	Rel(Arguments)	326	318	8	0	100%	98%	99%	98%
Entire Prim Requirement		162	122	7	33	79%	95%	86%	75%

tences failed due to the failure of DSSAM at any phase. It is also worth noting that, the failed sentences contain (sub-)components process-able by the 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.

Overall the RCM-Extractor achieved 94% and 94% accuracy for extracting the initial and final requirements components, 78% accuracy in Rel(Sub-components), 98% accuracy in Rel(Arguments) and 75% accuracy in the entire primitive requirement extraction. The main causes of extraction failure are:

- the Stanford accuracy (e.g., the requirement ”the display elements glow”, failed in Phase 1 since it is wrongly interpreted as NP by StanfordNLP although it is a simple sentence).
- input with wrong grammar (e.g., ”if timer greater than timeout then heater_command equal to error.”, failed in Phase3, the first clause ”if timer greater than timeout” does not have verb).
- current extraction limitations (e.g., ”while moving the window up, the engine control system shall be essentially single fault tolerant with respect to loc event”, failed in Phase 3 due to the excess word ”essentially” –excesses words are words that do not affect semiformal/formal semantic of the input requirement (e.g., adverbs)).

These causes affect 9, 7,and 17 requirement sentence respectively of the missed 33. 76% of the requirements are correctly extracted and $\approx$ 4% are partially correct due to the ”missed”/”partially extracted” components through the RCM-Extractor phases.

5 RELATED WORK

There is a rich body of research for formalizing requirement specification into formal notations to eliminate errors in an early phase. The main adopted paradigm is feeding the approach with requirement sentences written in pre-defined format(s) [Konrad and Cheng, 2005, Pohl and Rupp, 2011, R. S. Fuchs, 1996, Mavin et al., 2009, Sládeková, 2007, Justice, 2013, Marko et al., 2015, Fu et al., 2017]. Then the defined structure of the template is utilized for parsing the NL requirements into formal notations. The extraction of each technique differs slightly upon the addressed elements in the defined template.

Ghosh et al., in [Ghosh et al., 2016] proposed a framework called ARSENAL for translating natural language requirements into LTL. In ARSENAL, a semantic analysis applies on dependency parsing and POS tags of a simplified sentence. NExt, grammatical relation with word types are aggregated into an intermediate representation ”IR”. Then, IR is enriched with formal information resulted from applying a set of hand crafted mapping rules on the typed dependency of the input sentence. Such rules are inevitably domain-specific and limited to restricted scenarios. The final version of IR can be eventually converted into several formalisms, including LTL. The approach achieved 78% and 95% on two different data-sets. The provided approach is order sensitive since it achieved 95% and 65% accuracy for the same data-set by perturbing ”If A then B” to ”B if A”.

Yan et al., in [Yan et al., 2015] presented NLP-based technique for formalizing NL-requirements into LTL. Similar to our approach, this technique identify clauses of a requirement mapping each to one proposition in LTL later. The provided approach allow coordination between clauses which is currently lacked by our approach. On the other hand, this approach is very limited to its very strictly defined clause structure. In which, a clause should contain (1) single word noun as a subject and a verb predicate with one of the following formats ”verb — be+(gerund—participle) — be+complement”, (2) the complement should be adjective or adverbial word, (2) prepositional phrases are not allowed except ”in + time point” at the end of the clause, etc. Moreover, more complex cases of natural language (e.g., relative clauses, imperative cases, and intermingled clauses) are not addressed. Time scope and repetition properties are not considered in the defined structure as well.

Conversely to this approach, our technique aims to process NL-requirements instead and thus reveal the overhead work of rewriting the requirements in addition to covering a wider range of requirements structure due to the insensitivity to number, order, or types of components constituting a requirements sentence.

6 CONCLUSION

The paper introduces RCM-Extractor - an automated approach to extract and transform textual requirements into intermediate representation - RCM. In which, RCM is expressive enough to be transformed into formal notation. The approach is domain independent and insensitive to structure and format of the input requirements. We evaluated the approach on 162 requirements sentences achieving 95% precision, 79% recall, 86% F-measure and 75% accuracy.

REFERENCES

Bitsch, F. (2001). Safety patterns—the key to formal specification of safety requirements. In *International Conference on Computer Safety, Reliability, and Security*, pages 176–189. Springer.

Brunello, A., Montanari, A., and Reynolds, M. (2019). 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.

Buzhinsky, I. (2019a). 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.

Buzhinsky, I. (2019b). Formalization of natural language requirements into temporal logics: a survey. pages 400–406.

Das, B., Majumder, M., and Phadikar, S. (2018). A novel system for generating simple sentences from complex and compound sentences. *International Journal of Modern Education and Computer Science*, 11(1):57.

Dick, J., Hull, E., and Jackson, K. (2017). *Requirements engineering*. Springer.

Fifarek, A. W., Wagner, L. G., Hoffman, J. A., Rodes, B. D., Aiello, M. A., and Davis, J. A. (2017). Spear v2. 0: Formalized past ltl specification and analysis of requirements. In *NASA Formal Methods Symposium*, pages 420–426. Springer.

Fu, R., Bao, X., and Zhao, T. (2017). Generic safety requirements description templates for the embedded software. In *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*, pages 1477–1481. IEEE.

Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., and Steiner, W. (2016). Arsenal: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pages 41–46. Springer.

Houdek, F. (2013). System requirements specification automotive system cluster(elc and acc). *Technical University of Munich*.

Huddleston, R. and Pullum, G. (2005). The cambridge grammar of the english language. *Zeitschrift für Anglistik und Amerikanistik*, 53(2):193–194.

Jeannet, B. and Gaucher, F. (2016). Debugging embedded systems requirements with stimulus: an automotive case-study.

Justice, B. (2013). Natural language specifications for safety-critical systems. Master's thesis, Carl von Ossietzky Universität.

Konrad, S. and Cheng, B. H. (2005). Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM.

Leech, G., Deuchar, M., and Hoogenraad, R. (1982). *English Grammar for Today*. Palgrave Macmillan UK.

Lúcio, L., Rahman, S., bin Abid, S., and Mavin, A. (2017a). Ears-ctrl: Generating controllers for dummies. In *MODELS (Satellite Events)*, pages 566–570.

Lúcio, L., Rahman, S., Cheng, C.-H., and Mavin, A. (2017b). Just formal enough? automated analysis of ears requirements. In *NASA Formal Methods Symposium*, pages 427–434. Springer.

Marko, N., Leitner, A., Herbst, B., and Wallner, A. (2015). Combining xtext and oslc for integrated model-based requirements engineering. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 143–150. IEEE.

Mavin, A., Wilkinson, P., Harwood, A., and Novak, M. (2009). Easy approach to requirements syntax (ears). In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 317–322. IEEE.

Pohl, K. and Rupp, C. (2011). *Requirements Engineering Fundamentals*. Rocky Nook.

R. S. Fuchs, N. E. (1996). Attempto controlled english (ace). In *CLAW 96, First International Workshop on Controlled Language Applications*, Katholieke Universiteit, Leuven.

Sládeková, V. (2007). Methods used for requirements engineering. Master's thesis, Univerzity Komenského.

Teige, T., Bienmüller, T., and Holberg, H. J. (2016). Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. pages 6–9. MBMV.

Thyssen, J. and Hummel, B. (2013). Behavioral specification of reactive systems using stream-based i/o tables. *Software & Systems Modeling*, 12(2):265–283.

Yan, R., Cheng, C.-H., and Chai, Y. (2015). Formal consistency checking over specifications in natural languages. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1677–1682. IEEE.

Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., and Ibrahim, A. (2020). Rcm: Requirement capturing model for automated requirements formalisation.