SN Computer Science (2023) 4:57
https://doi.org/10.1007/s42979-022-01449-7

ORIGINAL RESEARCH

# A Comprehensive Requirement Capturing Model Enabling the Automated Formalisation of NL Requirements

**Mohamed Osama[1] · Aya Zaki-Ismail[1] · Mohamed Abdelrazek[1] · John Grundy[2] · Amani Ibrahim[1]**

Received: 15 September 2021 / Accepted: 10 October 2022
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2022

**Abstract**
Formalising natural language (NL) requirements is essential to have formal specifications that enable formal checking and improve the quality of requirements. However, the existing formalisation techniques require engineers to (re)write the system requirements using a set of requirements templates with predefined and limited structure and semantics. The main drawback of using such templates, usually with a fixed format, is the inability to capture diverse requirements outside the scope of the template structure. To address this limitation, a comprehensive reference model is needed to enable capturing key requirement properties regardless of their format, order, or structure. NLP technique can then be used to convert unrestricted NL requirements into the reference model. Using a set of transformation rules, the reference model representing the requirements can be transformed into the target formal notation. In this paper, we introduce requirement capturing model (RCM) to represent NL requirements by adapting to their key properties and without imposing constraints on how the requirements are written. We also implemented a requirements formalisation approach that supports transforming RCM into temporal logic (TL). In addition, we developed an automated similarity checking approach to check the correctness of the constructed RCM structures against the source NL requirements. We carried out extensive evaluation of RCM by comparing it against 15 existing requirements representation approaches on a dataset of 162 requirement sentences. The results show that RCM supports a much wider range of requirements formats compared to any of the existing approaches.

**Keywords** Requirement representation · Requirement modelling · Requirement engineering · Requirement formalisation

## Introduction

Formal verification techniques require system requirements to be expressed in formal notations [1]. However, the majority of critical system requirements are still predominantly written in informal notations—textual or natural languages (NL)—which are inherently ambiguous and have incomplete syntax and semantics [2, 3]. To automate the formalisation process, several bodies of work within the literature focussed on proposing predefined requirement templates, patterns [4], boilerplates [5], and structured control English [6], to express one system requirement sentence while eliminating the ambiguities.

These templates need to have a complete syntax to ensure the feasibility of transforming textual requirements into formal notations using a suite of manually crafted, template-specific transformation rules (e.g. [7]). However, some of the predefined templates are domain dependent and are hard to generalise [8], or can only capture limited subsets of requirements structures [6]. In addition, most existing formalisation algorithms are customised for transforming system requirements to one target formal language. Thus, a need to transform the same requirements into different formal languages mandates significant rework of the formalisation algorithm. Requirements engineers are limited when

✉ Mohamed Osama
mdarweish@deakin.edu.au

Aya Zaki-Ismail
amohamedzakiism@deakin.edu.au

Mohamed Abdelrazek
mohamed.abdelrazek@deakin.edu.au

John Grundy
john.grundy@monash.edu

Amani Ibrahim
amani.ibrahim@deakin.edu.au

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

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

Published online: 16 November 2022

using these pre-existing requirements templates and need to learn to use them and use them accurately for requirements formalisation tools based upon them to work.

Instead of considering introducing new sentence-based templates covering a wider range of requirements and complicating the requirements specification process, in [9] we introduced a requirement capturing model (RCM), as a reference model that defines the key properties that make up a system behavioural requirement sentence, *regardless of the syntactic structure of these properties, lexical words, or their order*. RCM separates the writing styles (format and structure) from the abstract requirement properties and the formal notations. Our new RCM model enables us to: (1) represent a much wider range of requirements that have differing count, order or types of properties, by identifying the specific properties in the input requirement sentence to generic RCM defined properties; (2) specify requirements in a wide variety of different formats, extremely useful to avoid re-writing existing requirements; (3) formalise requirements into different formal notations through mapping RCM properties to those of the target formal notation; and (4) enable use of NLP-based requirements extraction techniques to transform textual requirements into the RCM-based requirements model. with the key elements to be extracted now clearly defined and known.

The correctness of the generated formal notations mainly depends on the correctness of the RCMs representing the system requirements. Manually confirming the correctness of RCMs against the corresponding system requirements consumes a considerable amount of time and effort. In this paper, we extend our previous work [9] by introducing an automated approach to automatically check the correctness of RCM structures according to the corresponding NL requirements. We also add new experiments to the evaluation section assessing the performance of the proposed approach in checking the correctness of the automatically constructed RCMs in [10] against their corresponding source requirements. Key contributions of this work are:

- Present RCM as a new reference model and intermediate representation between informal and formal notations that can be automatically validated.
- Describe a set of transformation rules from RCM to Metric Temporal Logic (MTL), to demonstrate how an RCM-based requirements can be transformed into a formal notation.
- Introduce an RCM checking approach that automatically assesses the correctness of RCM structures against their source system requirements.
- Evaluate the expressiveness power of RCM by comparing it to 15 other existing approaches using 162 behavioural requirement sentences for critical systems. In addition, we evaluated the proposed RCM checking approach on the same dataset.

The rest of this paper is organised as follows. "Motivation" provides a motivating example. "Related Work" covers the key related work. "Requirement Capturing Model" presents the details of RCM. "RCM Correctness Checking Approach" discusses the automatic RCM checking approach. "Evaluation" evaluates the expressiveness power of RCM and the performance of the proposed checking approach. "Summary" concludes the paper.

## Motivation

Consider Jen who is a systems engineer working for an automotive company. She wants to specify the requirements of one of the system modules—a small excerpt is shown in Table 1—while making sure that these requirements can be easily transformed into formal notations as a mandatory compliance requirement. Jen decided to check the existing requirement specification techniques in the literature to choose which one covers most of her requirements. Jen researched the existing requirements formalisation techniques, see the related work section for these techniques, and outlined her trials to use these techniques to model her requirements after rephrasing some of her requirements to suit the existing templates.

Jen discovered that none of the existing techniques she investigated could be used to cover all her requirements. She then had to learn and use all these templates and have these tools all running. Furthermore, Jen found that the majority

**Table 1** Examples of critical system requirements and approaches representing them

| |
|---|
| **RQ1:** R_STATUS shall indicate the rain sensor. It shall be ON, when the external environment is raining |
| **Techniques:** Universal pattern [11], Structured English [12], Rup's boilerplates [8], ACE [6], EARS [5], CFG [3] and BTC [4] |
| **RQ2:** When the external environment rains for 1 min, the wipers shall be activated within 30 s until the rain sensor equals OFF |
| **Techniques:** Universal pattern [11] and BTC [4] |
| **RQ3:** While the wipers are active, the wipers speed shall be readjusted every 20 s |
| **Techniques:** Structured English [12] |

of these solutions rely on predefined formats and structure of requirements boilerplates. This mandates: (1) a fixed order of requirement components/sub-components, (2) a fixed English syntax for a specific component/sub-component, and (3) a fixed/small set of English verbs or other lexical words. Hence, Jen needs to rewrite her requirements to conform to the predefined format which puts more overhead on her, especially if such formats are limited and cannot be extended to new scenarios.

Taking into consideration all the combinations of styling, ordering, and omission/existence of different requirements model properties increases the size of the defined formats. Consequently, this increases the complexity of using them by system engineers and the complexity of the parsing algorithms needed to transform them into formal models.

Furthermore, most existing formalisation techniques apply on-the-fly transformation on the given structured requirement sentences to generate the corresponding formal notation. These transformations are hard-coded or tightly customised according to the target formal notation properties and formats. This type of transformation is customised according to the target formal notation—mostly a version of TL (e.g. CTL, MTL). Different versions of TL share some notations (e.g. propositions/predicates). Thus, it is better to compute the common parts only once and store them for later usage (generating different versions of TL). Having a rich model that supports storing the common parts, and enables computing the distinct parts independent from the common ones, can improve the formalisation performance. It also enables the transformation into multiple TL versions with minimal adjustments to the developed transformation technique.

## Related Work

Many requirements formalisation approaches assume the requirements to be specified in a constrained natural language (CNL) with specific style, format and structure to be able to transform into formal notations (e.g. [13–19]). These CNL approaches are meant to avoid natural-language-related quality problems (e.g. ambiguity, and inconsistency) and increase the viability of automating the formalisation process.

CNL is a restricted form of NL especially created for writing technical documents as defined in [20] with the aim of reducing/avoiding NL problems (e.g. ambiguity, inconsistency). CNL typically has a defined subset of NL grammar, lexicon and/or sentence structure [21]. Different forms of CNL are also provided as a reliable solution for requirements representation. Schwitter and Fuchs [6] proposed Attempto Controlled English (ACE) with a restricted list of verbs, nouns and adjectives for the requirements set, in addition to other restrictions on the structure of the sentence. ACE can be transformed into Prolog. It can handle requirements with Condition and Action components. Inspired by ACE, multiple CNLs have been proposed (e.g. Atomate language [22], and PENG [23]) for formal generation purposes and for other purposes (e.g. BioQuery-CNL [24]).

Similarly to ACE, Scott and Cook [25] presented Context Free Grammars (CFGs) for requirements specification. Although the formats of the requirement components is more limited than ACE with additional restrictions on words, it covers a broader range of requirement properties (e.g. Valid-time for Action). Yan et al. [7] presented a more flexible CNL with constraints on the word set such that 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 an adjective or an adverbial word, and (3) prepositional phrases are not allowed except "in + time point" at the end of the clause. This CNL does not consider time information except for a Pre-elapsed-time.

Boilerplates are also widely used. They provide a fixed syntax and lexical words with replaceable attributes. Boilerplates are more limited than CNL and require adaptation to different domains. In [8], a constrained RUP's boilerplate is provided to handle a limited range of requirements. EARS [5] boilerplates are less restricted and can support a wider range of requirements. Esser and Struss [26] proposed a suite of requirement templates (TBNLS) with mapping support to propositional logic with temporal relations. For validating the conformity of the written requirement and the boilerplate, checking techniques were introduced in [27, 28].

Requirement patterns provide a more flexible solution. However, when a new requirement structure is added, a new pattern must be created for it. This leads to a continuous increase in the size of patterns. In [11], a universal pattern is presented to support many requirements formats (Trigger then Action). Additional time-based kernel patterns were further introduced in [4]. Although these patterns cover many requirement properties, they still do not cover all the possible combinations of the supported properties eligible to a single requirement specification. In addition, the approach lacks complex time properties (e.g. In-between-time, and Pre-elapsed-time properties). Dwyer et al. [29] proposed several patterns applicable for non-real-time requirements specification. These patterns support scopes (e.g. globally, before R, after R), and are categorised into two major groups: (1) occurrence patterns, and (2) order

**Table 2** The supported properties and formats in the existing approaches

| Approach | | Requirement properties | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Source | A | A-vt | A-rt | A-pt | C | C-vt | C-pt | T | T-vt | T-rt | SP | SP-vt | EP | EP-vt | SA | SA-vt | EA | EA-vt | Hid |
| A1 | BTC [4, 11] | 1 | 1 | | 1 | 1 | 1 | | | | | 1 | | | | 1 | 1 | 1 | 1 | |
| | | 1 | 1 | | 1 | | | | 1 | 1 | | 1 | | | | 1 | 1 | 1 | 1 | |
| A2 | EARS [5] | 1 | | | | 1 | | | 1 | | | | | | | | | | | |
| | | 1 | | | | 1 | | | | | | 1 | | | | | | | | |
| | | 1 | | | | | | | 1 | | | 1 | | | | | | | | |
| A3 | EARS-CTRL [2] | 1 | | | | | | | 1 | | | 1 | | | | | | 1 | | |
| A4 | ECA [22] | 1 | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | |
| A5 | boilerplates [8] | 1 | | | | 1 | | | | | | | | | | | | | | |
| | | 1 | | | | | | | 1 | | | | | | | | | | | |
| A6 | Safety templates [39] | 1 | | 1 | 1 | | | | | | | | | | | 1 | | 1 | | |
| | | 1 | | 1 | 1 | 1 | | | | | | | | | | 1 | | 1 | | |
| | | 1 | | | | | | | 1 | | 1 | 1 | | 1 | | | | | | |
| A7 | Req Lang [40] | 1 | 1 | | | 1 | | | 1 | | | 1 | | 1 | | | | | | |
| A8 | CFG [3, 25] | 1 | | | | 1 | | | | | | | | | | | | | | 1 |
| | | 1 | | | | | | | 1 | | | | | | | | | | | 1 |
| | | 1 | | | | | | | | | | | | | | 1 | | | | 1 |
| | | 1 | | | | | | | | | | | | | | | | 1 | | 1 |
| A9 | ACE [6] | 1 | | | | 1 | | | | | | | | | | | | | | 1 |
| A10 | PENG [23] | 1 | | | | 1 | | | 1 | | | 1 | | 1 | | 1 | | 1 | | 1 |
| A11 | Struct.English [7] | 1 | | | 1 | 1 | | | 1 | | | 1 | | 1 | | 1 | | 1 | | |
| A12 | TBNLS [26] | 1 | 1 | | | 1 | 1 | | | | | 1 | | 1 | | | | 1 | | |
| A13 | Real-time [12] | 1 | | | 1 | | | | | | | | | | | 1 | | 1 | | |
| | | 1 | | 1 | | | | | | | | | | | | 1 | | 1 | | |
| | | 1 | 1 | | | | | | | | | | | | | 1 | | 1 | | |
| | | 1 | | 1 | | | 1 | | | | | 1 | | 1 | | | | | | |
| | | 1 | 1 | | | | 1 | | | | | 1 | | 1 | | | | | | |
| | | 1 | | | | 1 | | | | | | 1 | | 1 | | | | 1 | | |
| A14 | Dwyer [29] | 1 | | | | 1 | | | | | | 1 | | 1 | | | | 1 | | |
| A15 | Pattern_based [41] | 1 | 1 | | 1 | 1 | 1 | | | | | | | | | | | | | |

Property symbols → *A*:Action, *C:* Condition, *T:* Trigger, *Hid:* Hidden-constraint, *SP:* Precond StartUP-phase, *EP:* Precond EndUP-phase, *SA:* Action StartUP-phase, *EA:* Action EndUP-phase, *vt:* Valid-time, *pt:* Pre-elapsed-time, *rt:* In-between-time

patterns. The work was later extended in [12] to cope with real-time requirements specification. The real-time patterns consider versions of the Pre-elapsed-time, In-between-time and Valid-time information for the Action component.

Event-Condition-Action (ECA) was initially proposed in active databases area to express behavioural requirements. ECA became widely used by several researchers in diffident areas. An ECA rule assumes that when an Event E occurs, the Condition C is evaluated, and if it is true, the Action A is executed. ECA notations have been extended to capture time information in [30]. However, ECA rules do not support invariants (rules without preconditions), do not consider Scopes for Action, and the defined time notations only apply for the Events part.

## Requirement Capturing Model

We first explain the process we followed to develop RCM as a new comprehensive reference model. Then we describe the RCM meta-model in details. Finally, we introduce an RCM to formal notations transformation procedure.

### RCM Development Process

To identify the key requirement properties needed to be supported in a generic reference model for safety-critical requirements, we examined a large number of natural language-based critical system requirements curated from

several sources [2, 5, 6, 11, 31–38], in addition to 15 requirement representation approaches listed in Table 2.

We identified 19 distinct properties (17 from the existing approaches and 2 from the analysis of the requirements). Table 2 maps these properties to the investigated approaches (outlined in the related work section of this paper). In the table, the approaches are encoded as A1 to A15 and are listed in the source column. The remaining columns represent the identified requirement properties. Each approach is represented by one or more rows in the table depending on the number of template or pattern variations of the approach. This reflects that such approaches support multiple properties, but these properties cannot be used in the same requirement (i.e. each variation of the approach supports only a certain combination of properties in one requirement). A cell is filled with "1" if the property is supported in the template/pattern represented by the row containing the cell.

The table shows that some properties are supported by all or most of the investigated approaches, while some other properties are not even supported by a single approach. For example, the Action property "A" is supported by all the investigated approaches, but the Valid-times of both the Precond StartUP-phase "SP-vt" and the Precond EndUP-phase "EP-vt" are not supported by any of the approaches despite being present in the analysed requirements. An example of "SP-vt" is highlighted in bold in the following requirement *"After the switch is set to AUTO* ***for 2s****, if the headlights are OFF and the light intensity falls below 60%, then the lights should be turned ON."*.

Our analysis of this table shows that: (1) no approach covers all the identified requirement properties (possibly because this makes the approach too complex to use), (2) all the approaches support the Action property, (3) the approaches "A1" and "A11" are the most expressive approaches as they cover the majority of the properties, and (4) both the StartUP and EndUP Valid-time properties are not supported by any of the approaches (despite their existence in the analysed requirements). In addition, this table does not reflect the limitations or restrictions that these approaches apply on the formatting and/or order of a given property (e.g. a Condition must come before an Action, or a Req-scope comes before a Condition). All These limitations reflect the restricted focus of the investigated approaches.

To capture the roles and relations between the identified properties in a requirement sentence, we grouped them the properties into eight abstract property types (4 components and 4 sub-components). The properties that can independently exist in a requirement are considered to be components, while the properties that must be attached to another property or can only be encapsulated within another property are considered to be sub-components. For example, the Action property (indicating a task to be executed by the system) is considered to be a component. However, the Valid-time property (indicating the time period for the execution) is considered to be a sub-component. The eight abstract property types are listed and described in Tables 3 and 4. The manually crafted requirement (containing most of the components and sub-components), used as an example in the table, is shown in Fig. 1.

## RCM Meta Model

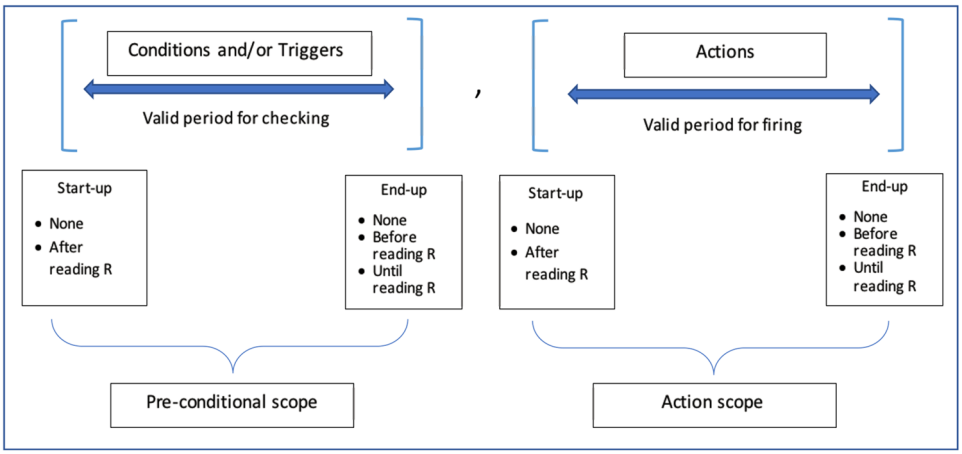
RCM is designed to capture the requirements properties listed in Tables 3 and 4 while relaxing the ordering and formatting restrictions imposed by the existing approaches. Consider a system represented as a set of requirements $R$. Each requirement $R_i$ is represented by one RCM and may have one or more primitive requirements PR where $\{R_i = \langle \mathrm{PR}_n \rangle \text{ and } n > 0\}$. Each $\mathrm{PR}_j$ represents only one requirement sentence, and may include Condition(s), Trigger(s), Action(s) and Requirement-scope(s) (i.e. the four components in RCM).

Figure 2 presents the detailed meta-model of RCM for one requirement $R_i$. The components are highlighted in green. While the sub-components are highlighted in yellow. The figure shows the relations between the components and sub-components, where some sub-components can be encapsulated in all the components while others can only be linked to specific components (e.g. Pre-elapsed-time and In-between-time). In addition, the Hidden-constraint sub-component is directly linked to its relevant operand. As shown in the figure, all the components and sub-components are eventually represented as predicates and time structures. These structures encapsulate the semi-formal semantics of the requirement (highlighted in orange), in addition to some formal semantics as well (highlighted in blue). The formal semantics of a predicate has two different formats that are dependant on the semi-formal semantics of the component.

The figure shows that a primitive *requirement* can be composed of four requirement *component* types: *Condition*, *Trigger*, *Action* and *Requirement-scope*. Except for Action, the existence of each of these components is optional in a primitive requirement (i.e. a requirement sentence must contain at least one Action). Each requirement component has a *Core-segment* that expresses the main portion of the component, and can optionally also have a *Valid-time* (the time length of the component being valid). The *Pre-elapsed-time* sub-component can only appear with a Condition or an Action component (based on the analysis of requirements and the conceptual meaning of the properties). An *In-between-time* sub-component can only appear with Trigger or Action components (as per the reviewed requirements and representation formats). A *Hidden-constraint* is an optional sub-component for an operand. To store this information without loss, RCM stores the Hidden-constraint inside the relevant operand object as indicated in Fig. 2. This

**Table 3** A list of identified abstract component

| Property | Description |
|---|---|
| *Component* | |
| Trigger | Is an event that initiates Action(s). For example, consider the following part of the requirement shown in Fig. 1: "**When the acoustical signals $\langle E \rangle$ turns to TRUE every 1 seconds**, M shall transition to FALSE before $\langle B_s ig \rangle$ is TRUE". The Trigger (highlighted in bold) automatically fires the Action (underlined text) to be executed. This component is ubiquitous throughout the requirements of most critical systems |
| Condition | Is a constraint that should be satisfied to allow a specific system Action(s) to happen. For example, "After sailing termination, **if X is ON for 1 second or (Y is ON and Z is ON)**, M shall transition to TRUE after less than 2 seconds" in Fig. 1. The Condition is highlighted in bold text in the requirement sentence. In contrast to Triggers, the satisfaction of the Condition should be checked explicitly by the system. The system is not concerned with "when the constraint is satisfied" but with "is the constraint satisfied or not at the checking time" to execute the Action (e.g. in the previous example "X" might remain "ON" for a while and have no effect on the system until its value is checked) |
| Action | Is a task that should be accomplished by the system in response to Triggers and/or constrained by Conditions. For example, "When the acoustical signals $\langle E \rangle$ turns to TRUE every 1 seconds, **M shall transition to FALSE** before $\langle B_s ig \rangle$ is TRUE" in Fig. 1. The Action component is represented by the text in bold in the requirement. In case that a primitive requirement consists of an Action component only, it is marked as **a factual rule** expressing factual information about the system (e.g. *The duration of a flashing cycle is 1 s* [42]) |
| Req-scope | Determines the context under which (i) "Condition(s) and Trigger(s)" can be valid (called a Preconditional-Scope in this case as it is linked to the Condition or Trigger), and (ii) "Action(s)" can occur (called an Action-Scope as it is applied only on the Action). The bold text in "**After sailing termination**, if X is ON for 1 second or (Y is ON and Z is ON), M shall transition to TRUE after less than 2 seconds." in Fig. 1 shows an example of Preconditional-Scope. In this case, the Conditions should be checked after the "sailing termination" event happens. In contrast, the bold text in "When the acoustical signals $\langle E \rangle$ turns to TRUE every 1 seconds, M shall transition to FALSE **before $\langle B_s ig \rangle$ is TRUE**" in Fig. 1 shows an Action-Scope example where the Action should occur before "$\langle B_s ig \rangle$ is TRUE".. The scope may define the starting or the ending boundaries (e.g. "after sailing termination", "before <B_sig> is True" in Fig. 1). The following figure presents the main variations for starting/ending a context: None, after operational constraint is true, until operational constraint becomes true, and before operational constraint becomes true. Other alternatives can be expressed by the main variations. For example, "while R is true" can be expressed by after and until as "after R is true" and "until not R". It is worth noting that "Before" and "Until" define the same end of the valid period which is "R is true". "Until" mandates the Precondition(s)/Action(s) to hold till "R is true", but "Before" is not concerned with their status |




breakdown is carefully designed to allow nested Hidden-constraints. For example, consider the following requirement sentence "*the entry of A1* ***whose index is larger than the first value in A2*** *that is larger than S1 shall be set to 0"*. In this example, the text in bold is a Hidden-constraint "H1" specified for the argument "the entry of A1" and is stored inside of it. In addition, the underlined text is a Hidden-constraint "H2" specified for the argument "the first value in A2" and is stored inside of it. Since the argument encapsulating H2 is part of H1, H2 is stored (nested) inside H1.

All sub-components are instances of either *Predicate* or *Time* structures. The *Predicate* structure consists of operands, an operator and a negation flag/property (e.g. in "if X exceeds 1", "X" and "1" are the operands and "exceeds" is the operator in the semi-formal semantics and ">" is the operator in the formal semantics). The *Time* structure stores

**Table 4** A list of identified abstract sub-components

| Property | Description |
|---|---|
| *Sub-component* | |
| Valid-time | Represents the valid time period of a given component (e.g. in the following requirement "the vehicle warns the driver by acoustical signals ⟨*E*⟩ for 1 second", the Action continues for 1 s [42]). Valid-time can be a part of any component |
| Pre-elapsed-time | Is the consumed time length from an offset point—before an Action occurs or a Condition is checked (e.g. in the requirement sentence "After sailing termination, if X is ON for 1 second or (Y is ON and Z is ON), M shall transition to TRUE **after less than 2 seconds"** in Fig. 1, the Action should happen after at most 2 s). This type is only eligible for Action and Condition components |
| In-between-time | Expresses the length of time between two consecutive occurrences in case of repetition. For example, in the sentence "When the acoustical signals ⟨*E*⟩ turns to TRUE **every 1 seconds**, M shall transition to FALSE before ⟨$B_sig$⟩ is TRUE" in Fig. 1, the text in bold represents an In-between-time sub-component for the repeated occurrence of the Trigger "the acoustical signals ⟨*E*⟩ turns to TRUE". This sub-component is eligible for Action and Trigger components as indicated in Fig. 2 |
| Hidden-constraint | Allows an explicit constraint to be defined on a specific operand within a component. For example, "if the camera recognises the lights of an advancing vehicle, the high beam headlight **that is activated** is reduced to low beam headlight within 5 second" [42]. The text in bold (**that is activated**) is a constraint defined on the operand (*the high beam headlight*) |

the unit, value and quantifying relation (e.g. in "for less than 2 seconds", "2" and "seconds" are the unit and value, respectively, "less than" is the semi-formal quantifying relation whose formal semantics is "<"). Since the *Predicate* and *Time* structures are the infrastructure of all the properties, they are designed to encapsulate the semi-formal and formal semantics allowing mappability to multiple TL. The details of formal semantics are described in "RCM and Formal Semantics".

In a primitive requirement, multiple components with the same type (e.g. multiple conditions in the first sentence

**Fig. 1** Crafted multi-sentence requirement "REQ"

**REQ:** After sailing termination, if X is ON for 1 second or (Y is ON and Z is ON), M shall transition to TRUE after less than 2 seconds. When the acoustical signals <E> turns to TRUE every 1 seconds, M shall transition to FALSE before <B_sig> is TRUE.

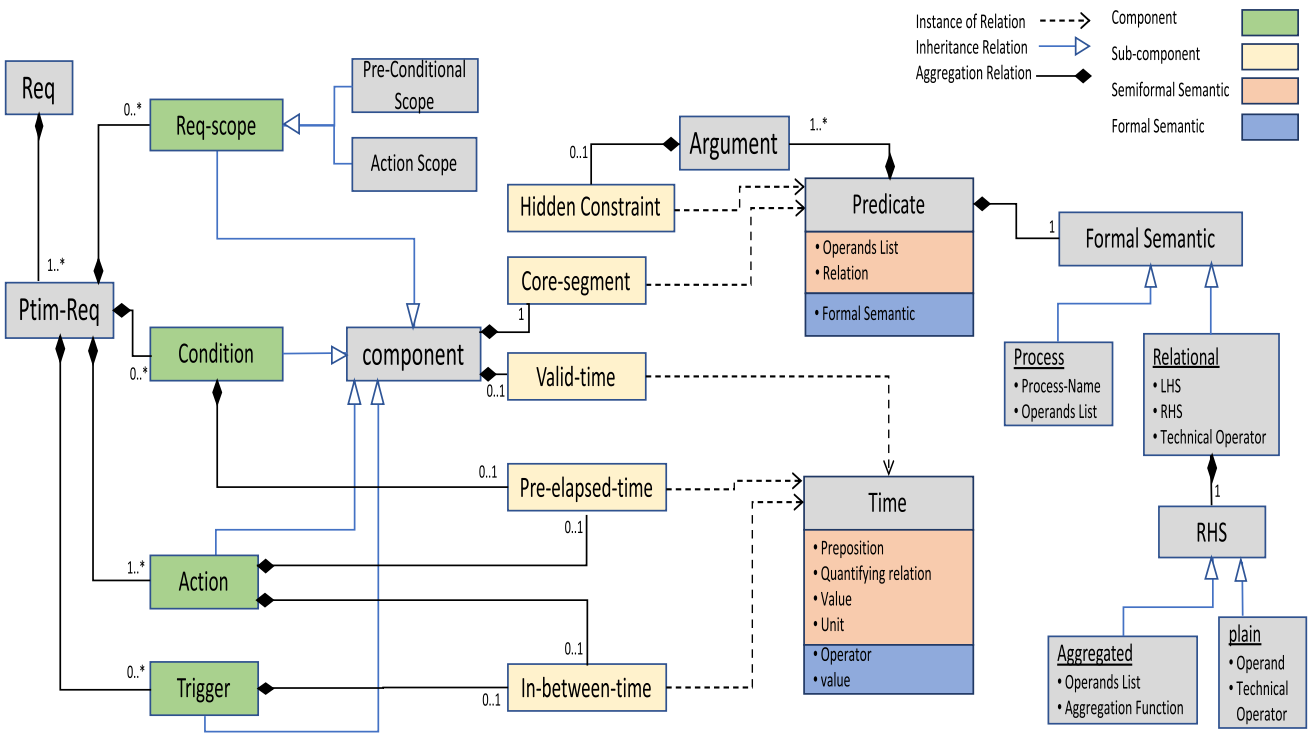



**Fig. 2** RCM meta-model (simplified)

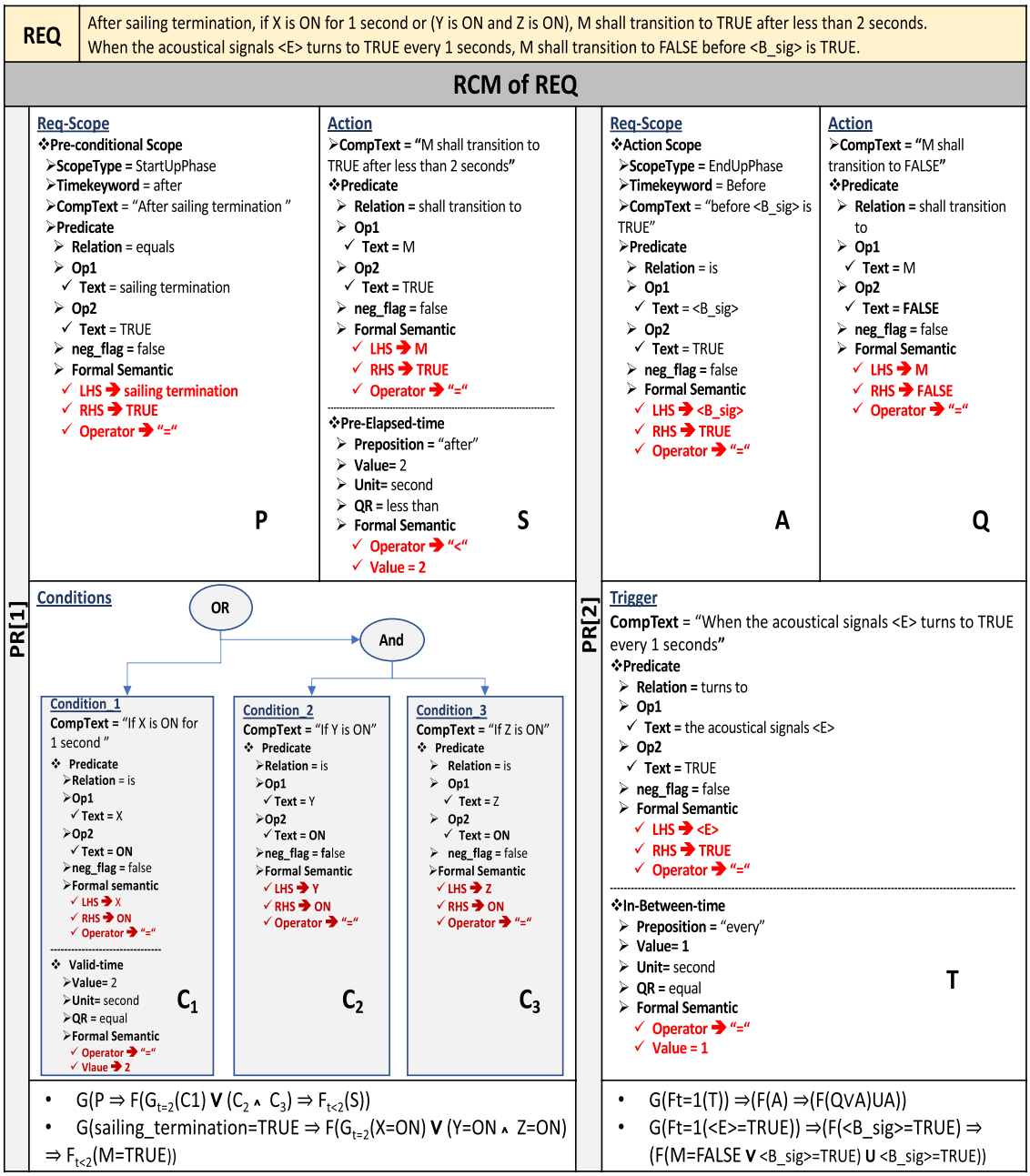



**Fig. 3** An example presenting a multi-sentence requirement "REQ" and the corresponding RCM representation

in Fig. 1) are linked through (nested-)coordination relation (i.e. and/or). In order to preserve this relation in RCM without loss, we store these component in a tree data structure (a suitable data structure to keep nested relations). In this structure, the components are stored as leaf nodes and the (nested-)coordination relations are stored in the interior nodes of the tree. Figure 3 shows an example that visualises the nesting relation in the tree representation for the Conditions components existing in PR[1].

Figure 3 shows the RCM representation of the REQ example. It has two primitive requirements: PR[1] and PR[2]. The components of each primitive requirement are presented in separate blocks in the figure. In each block, the sub-components are separated by a horizontal line. Figure 3 highlights the encapsulation of semi-formal semantics (in black) and formal semantics (in red). The figure also provides the corresponding MTL representation (discussed in details in "RCM Transformation Algorithm") of the requirement.

The following is a step-by-step analysis of the RCM representation of PR[1] "(1) After sailing termination, (2) if X is ON for 1 second or ((3) Y is ON and (4) Z is ON), (5) M shall transition to TRUE after less than 2 seconds". This requirement sentence has five components as numbered in the underlined text: one Preconditional-Scope, three Conditions, and an Action. Only two of these components has an attached sub-component: (1) the first Condition "if X is ON for 1 second" (contains a Valid-time sub-component "for 1 second"), and (2) the Action "M shall transition to TRUE after less than 2 seconds" (contains a Pre-elapsed-time "after less than 2 seconds"). Since the Conditions in the sentence (part 2, 3, and 4) are connected according to coordination relations, they are represented in RCM using a tree data structure preserving the correct connection between them. In Fig. 3, the black attributes show the semi-formal details while the red ones show the corresponding formal semantics. Our works in [10, 43, 44] discuss the details of automatically transforming NL requirements into RCM.

## RCM Transformation

In this subsection, we illustrate the transformation of RCM into TL. We first illustrate: (1) the mapping from RCM to TL, and (2) the formalisation of the RCM infrastructure (i.e. Predicate and Time structures). Then, we discuss the transformation process.

### RCM and TL

In order to formally model a given requirement represented by RCM in TL, we need to define a set of transformation rules. A TL formula $F_i$ is built from a finite set (AP) of proposition letters by making use of Boolean operators (e.g. "AND", "OR") and temporal modalities (e.g. U (until)) [45, 46]. Within such formula, each true/false statement is expressed by a proposition letter and may be attached with time notation in some versions of temporal logic (e.g. MTL). Consider the following sentence: "After the button is pressed, the light will turn red until the elevator arrives at the floor and the doors open[46]". This sentence can be captured by the following TL formula:

$$p \implies (qU(s \wedge v))$$

where $p$, $q$, $s$, and $v$ are proposition letter corresponding to "the button being pressed", "the light turning red", "the elevator arriving", and "the doors opening", respectively.

We apply the mapping between RCM and TL as follows:

1. Propositions and time notations: RCM components and sub-components are expressed as predicates or time structures as indicated in Fig. 2. These structures are mapped to proposition and time notations in the corresponding TL formula (e.g. the Action component "**M shall transition to TRUE** after less than 2 seconds" is mapped to "$F_{t<2}(S)$", where $S$ and "$t < 2$" represent the predicate in bold and the underlined time phrase).
2. Coordination relations: The logical operators connecting propositions can be obtained from the coordination relations connecting multiple components of the same type. Such relations are represented by tree structure in RCM, for each component type (as discussed before in "RCM Meta Model"). For example, the Condition components "X is ON for 1 second or (Y is ON and Z is ON)" are mapped to "$(G_{t=2}(C1) \vee (C2 \wedge C3))$".
3. Temporal modality: The temporal modalities can be identified based on the component type (e.g. the type of the component "After sailing termination" is "Preconditional-Scope StartUP-phase" and is mapped to "$\Longrightarrow$"

To demonstrate the flexibility of RCM and show that it can be transformed into different formal notations, we provide the mapping into two versions of TL (MTL [47] and CTL [48]), as shown in Table 5. We targeted these notations because they are widely used in model checking as indicated in [49] and [45], respectively. We base our temporal-modality and time notation mapping on the mapping done in [12].

The first column in Table 5 shows the RCM properties (components and sub-components), each attached with alternatives, if any (e.g. The Preconditions may be Conditions, Triggers, or both, based on the given requirement). The possible structures corresponding to each property version are listed in the third column (the utilised keywords, like "When", are just placeholders, and can be replaced by other keywords). The fourth column indicates which components can be linked to each property type. The MTL and CTL representations of each property are presented in the fifth and sixth columns, respectively, where these notations are grouped based on their formal types in the last column.

### RCM and Formal Semantics

TL has multiple versions exhibiting slight differences. In order to support the transformation to multiple versions with minimal adjustments in the transformation technique, RCM encapsulates formal semantics along with semi-formal semantics. Design-wise, RCM stores the formal semantics in basic units (predicate and time structures) as shown in Fig. 2. Such units are mappable to TL as indicated in Table 5. The formal semantics of a predicate covers three formats:

- Process format: Is suitable to predicates expressing functions or process (e.g. "the monitor sends a request

**Table 5** RCM mapping to MTL and CTL

| RCM | | | | | TL mapping | |
|---|---|---|---|---|---|---|
| Properties (component/sub-components) | | Versions | | Applicable on | MTL | CTL |
| *Temporal modality* | | | | | | |
| Action | | 1 | A: do something | | A | A |
| Precondition | Condition | 2 | If S | Action *(P in mapping)* | $G(S \Longrightarrow P)$ | $AG(S \Longrightarrow P)$ |
| | Trigger | 3 | When S | | $G(S \Longrightarrow P)$ | $AG(S \Longrightarrow P)$ |
| | Conditions and Triggers | 4 | When S, IF Q | | $G((S \wedge Q) \Longrightarrow P)$ | $AG((S \wedge Q) \Longrightarrow P)$ |
| Req-Scope: (Preconditional-Scope / Action-Scope) | StartUP | 5 | After S | Precondition/ Action *(P in mapping)* | $G(S \Longrightarrow F(P))$ | $AG(S \Longrightarrow AG(AF(P)))$ |
| | EndUP | 6 | Before S | | $F(S) \Longrightarrow (F(P \vee S)\mathbf{U}S)$ | $A[((AF(P \vee S)) \vee AG(\neg S))\mathbf{W}\, S]$ |
| | | 7 | Until S | | $F(P)US$ | $AF(P)US$ |
| | StartUP and EndUP | 8 | -After Q & Before S -Between Q and S | | $G((Q \wedge \neg S \wedge F(S)) \Longrightarrow F(P \vee S)\mathbf{U}S))$ | $AG((Q \wedge \neg S) \implies A[((AF(P \vee S) \vee AG(\neg S))\mathbf{W}S])$ |
| | | 9 | -After Q & Until S - While Z {Q=Z& S=¬ Z} | | $G((Q \wedge \neg S) \Longrightarrow F(P\mathbf{U}S))$ | $AG((Q \wedge \neg S) \implies A[(AF(P \vee S)\mathbf{W}S])$ |
| *Time notation* | | | | | | |
| Pre-elapsed-time | | 10 | After c time | Condition/Action *(P in mapping)* | $F_{t=c}(P)$ | |
| | | 11 | After at most c time | | $F_{t\leq c}(P)$ | |
| | | 12 | After at least c time | | $F_{t\geq c}(P)$ | |
| | | 13 | After less than c time | | $F_{t<c}(P)$ | |
| | | 14 | After greater than c | | $F_{t>c}(P)$ | |
| Valid-time | | 15 | For c time | Condition/Trigger/ Action *(P in mapping)* | $G_{t=c}(P)$ | |
| | | 16 | For at most c time | | $G_{t\leq c}(P)$ | |
| | | 17 | For at least c time | | $G_{t\geq c}(P)$ | |
| | | 18 | For less than c time | | $G_{t<c}(P)$ | |
| | | 19 | For greater than c | | $G_{t>c}(P)$ | |
| In-between-time | | 20 | Every c time | Action/Trigger *(P in mapping)* | $G(F_{t=c}(P))$ | |
| | | 21 | Every at most c time | | $G(F_{t\leq c}(P))$ | |
| | | 22 | Every at least c time | | $G(F_{t\geq c}(P))$ | |
| | | 23 | Every less than c time | | $G(F_{t<c}(P))$ | |
| | | 24 | Every greater than c | | $G(F_{t>c}(P))$ | |
| *Branch* | | | | | | |
| Hidden-constraint | | 25 | Whose S | P is Any component | | $AG(\exists S \Longrightarrow P)$ |

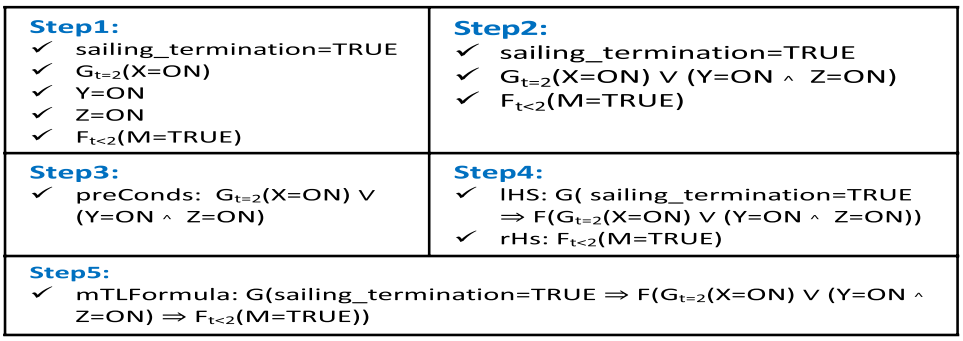



**Fig. 4** Step-by-step generation of PR[1] in Fig. 3

*REQ_Sig* to the station" ⟶ "send(the_monitor, the_station,*REQ_Sig*)").

- Relational format with plain RHS: This type is suitable for assignment predicates (e.g. "set X to True" ⟶"X = True"), comparison predicates (e.g. "If X exceeds Y" ⟶"X > Y") and changing state predicates (e.g. "the window shall be moving up" ⟶"the_window = moving-UP").
- Relational format with aggregated RHS: This format is similar to the previous one but the RHS is expressed with an aggregation function (e.g. "If the fuel level is less than the min value of Thr1 and Thr2" ⟶"the_fuel_level < "min(Thr1, Thr2)").

Similarly, the formal semantics is added to the time structure in which the technical time operator (e.g. $\{>, <, =, \leqslant, \geqslant\}$) is identified (e.g. "for at least 2 seconds" ⟶ "$t \geqslant 2$").

### RCM Transformation Algorithm

To accomplish the automatic transformation from RCM to MTL, we use the mapping rules provided in Table 5 on the obtained formal semantics of the given primitive requirements. Algorithm 1 shows the automatic transformation pseudo-code annotated in Fig. 4 with the output of each step for PR[1] in Fig. 3.

**Algorithm 1** RCM-to-MTL Transformation

```
Input:
    R: RCM-to-MTL indexed Mapping Rules
    PrimReq: primitive requirement of interest
procedure
    Step 1: Prepare each component
    for all comp ∈ PrimReq do
        Comp.Formal← Comp.CoreSegment.getFormalSemantic()
        for all timeInfo ∈ comp do
            Comp.Formal← comp.AttachTimeSemantic(timeInfo, R{10:24})
        end for
    end for
    Step 2: Aggregate components of the same type
    for all compTree ∈ CompTypeTree do
        aggVal ← aggRel(compTree)
        procedure AGGREL(Tree compTree)
            if compTree is leaf then
                return compTree.data.Formal;
            else
                return "(" + aggRel(compTree.Left) + compTree.data.LR()
                    + aggRel(CompTree.Right) ")"
            end if
        end procedure
        map.put(compTree.Type, aggVal)
    end for
    Step 3: Prepare Preconditions
    preConds ← preparePrecond(map[Triggers], map[Conditions], R{2:4})
    Step 4: Prepare LHS and RHS
    lHS ← prepareSide(preConds, R{5:9})
    rHS ← prepareSide(Actions, R{5:9})
    Step 5: Generate MTL formula
    if lHS ≠ φ then
        return lHS + "⟶" + rHS
    else
        return rHS
    end if
end procedure
```

First, we get the formal semantics of each component according to "RCM and Formal Semantics". Then, we compute the formal semantics of the entire tree through the recursive function aggRel. The leaf nodes represent components and the inner nodes represent the logical relations of each component type (discussed in "RCM Meta Model"). After that, we construct the main parts of the formula (i.e. preConditions, LHS and RHS) in Step3 and Step4 through the RCM-to-MTL mapping rules listed in Table 5. Finally, we generate the entire formula based on the populated sides (either "LHS ⟶ RHS" or "RHS") as in Step5.

## RCM Correctness Checking Approach

The correctness of RCM can be assessed by confirming the existing components, sub-components and their breakdowns against the corresponding source English sentences. To achieve this, the source sentence and the corresponding RCM should be expressed in the same notation (NL in our case) to enable checking. The main challenges in this approach is that the generated sentence from a given RCM is not identical to the input requirement sentence. The main reason is that the transformation is applied from a higher level of formality (semi-formal level) into a lesser level of formality (informal level) [50]. To overcome this, our correctness checking approach generates NL sentence(s) from the RCM structure, then measures the relational similarity between the input requirement sentence(s) (used to obtain the RCM structure) and the generated sentence(s). This handles the textual mismatch between the sentences and measures the correctness based on the identification of components, sub-components, and their connecting relations (i.e. internal and external). Our RCM correctness checking approach consists of two main processes: (1) NL generation (i.e. transforming RCMs into NL requirement sentences), and (2) relational similarity checking (i.e. checking the similarity between the original requirement sentence and the one generated from RCM).

### NL Generation

In this process, we transform a given RCM into a requirement sentence. We utilise our RCM-to-NL generation technique introduced in [50]. The technique consists of two tasks:

- Realisation task: express each component (structured in RCM as predicate) in a correct clause grammar. The realisation is achieved with the support of the Simple NLG library [51]. In this task, a correct grammatical syntax is assigned to the semi-formal breakdowns/elements of the component core-segment. All of the component types are assigned a present tense except for the Action – assigned a future tense.
- Structuring task: arrange the sub-components within each existing component in RCM and arrange the components within the generated sentence. The ordering is achieved based on priority indices assigned to each (sub-)component in RCM. A lookup table that maps each (sub-)component to a priority index is used in this task, where each index preserves the location of the (sub-)component in the generated sentence. First, priority indices are assigned from the lookup table to all RCM components and sub-components existing in a given RCM. Then, these indices are used to structure the generated sentence of the given RCM based on its (sub-)components.

### Relational Similarity Checking

In this process, we propose a relational graph-based similarity approach that captures and represents the constituting components or clauses within the requirement sentence and the relations among them. Within the proposed approach, each clause (basic unit within the English sentence) is identified and the arguments within such clause are grouped and linked. In addition, the relations between the constituting clauses are also captured and a link between every two related clauses is added to the relational graph of the requirement.

We also developed a component-aware formula measuring the similarity between the constructed relational graphs of the requirements that differentiates between internal and external similarities within the constructed graphs. This supports the understanding of how the requirements are related and identifying the similarity aspects between the requirements.

Our approach is primarily divided into two processes: (1) Relational Graph Construction, and (2) Relational Graph Similarity Measurement. In the first process, a relational graph is constructed for each input requirement sentence. The relational graph identifies the constituting clauses within each sentence and represents each clause. The participating words in the clause are represented as nodes and linked to a central clause-representative word (identified based on the semantic relations between the clause words). External relations between the different clauses within each sentence are also identified and a link or edge is constructed

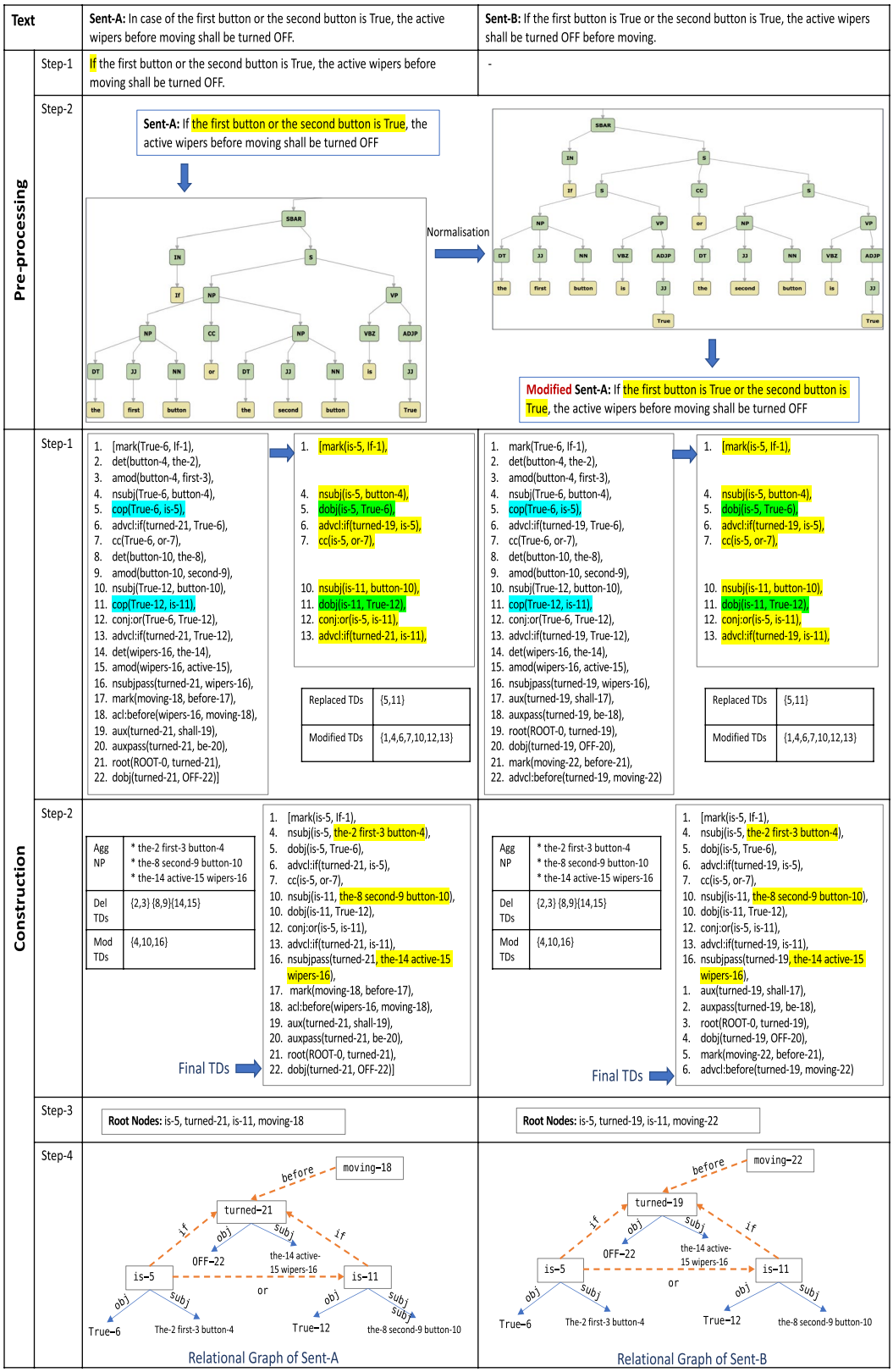



**Fig. 5** Normalised sentences

linking the central clause word. In the second process, a newly developed similarity measurement formula is used to compute the degree of similarity (a value between 0 and 1) between the constructed relational graphs.

In addition to capturing the contributing components of each requirement and their relations, we designed our approach to overcome five main challenges inherent to supporting NL similarity measurement. These challenges are summarised as follows:

- Different clauses order: two requirements may be completely similar (i.e. identical in meaning) but have different order for their constituting clauses. For example, "The light shall be turned ON if the button is pressed" and "If the button is pressed the light shall be turned ON".
- Interleaved clauses: having nested or interleaved clauses (i.e. a clause within another clause) can be challenging when measuring similarity. For example, "The IDC before termination shall be set to False" and "Before termination, the IDC shall be set to False". Our graph construction can successfully resolve this case by correctly identifying the involved clauses within the requirement.
- Excess words: words within a requirement sentence that do not affect the meaning of the sentence are not considered in the graph construction. For example, "If the error signal is active, an alarm shall be sent to the driver" and "If the error signal is *still* active, an alarm shall be sent to the driver". Here, the word "still" does not show in the constructed graph and does not affect the similarity between the two requirements. This is done by analysing the Typed Dependencies (TDs) relations (will be discussed in details in the relational graph construction subsection).
- Interchangeable keywords: our approach supports the identification of closed words in the English language. For example, "If the button is pressed the light shall be turned ON" and "In case the button is pressed the light shall be turned ON". In these requirements, "if" and "in case" are closed words of the same class. More details about this are provided in the preprocessing task.
- Coordination expression: some requirements may contain coordination (e.g. and) between their clauses. For example, "If Btn_1 or Btn_2 is pressed, the light shall be turned ON" and "If Btn_1 is pressed or Btn_2 is pressed, the light shall be turned ON". We provide details on how this is handled in the preprocessing task.

**Table 6** Roles of TDs relations in relational graph

| RG roles | TDs relations |
|---|---|
| NP Aggregation | amod, nummod, det, nmod |
| Root Nodes | mark, nsubj, nsubjpass, dobj, iobj |
| Internal Relation | nsubj, nsubjpass, nmod, obl, obj, iobj |
| External Relation | acl, advcl, ref, conj |

### Relational Graph Construction Process

The relational graph is responsible for visualising: (1) the external relations between the clauses in the sentence, and (2) the internal relation(s) within each clause. Each clause is represented by a one-level tree where: (1) the nodes are words/phrases of the clause, (2) the root is the tree representative, and (3) the edges express the internal relation(s) within the clause. Edges connecting the roots of such trees represent the external relations in the relational graph. We rely on the Stanford CoreNLP library to construct the relational graph of each input requirement. This process consists of two main tasks (preprocessing and construction). Figure 5 outlines the step-by-step construction process of the relational graphs for two sample requirements (the final constructed relational graphs are shown in Step-4).

**Preprocessing task:**

to provide a reliable and robust similarity measurement performance, the preprocessing task focuses on handling the interchangeable keywords and the coordination expression issues. We rely on the parse tree provided by the Stanford parser to resolve these issues. The two steps in this task are:

- Closed words unification: in this step, the closed words within each input requirement are identified and replaced with their respective class representative (e.g. "if", "in case of", and "provided that" are replaced with "If") as shown in Step-1 in Fig. 5. This unifies the constructed external relations and simplifies the assessment task.
- Coordination normalisation: in this step, we transform the internal coordination within a clause into an external coordination between clauses. This is done to normalise the structures of the requirements involving coordination and standardise the constructed relational graph. This maintains both the correctness and consistency of the constructed internal and external relations. First, we obtain the parse trees of the clauses containing internal coordination. Second, each coordination clause is

repeated according to its number of coordination (i.e. count of coordinated arguments). Each repetition contains one of the different coordinated arguments. Third, the sentence fragments (repeated clauses) are grouped together with their respective conjunction. Finally, the grouped clauses replace the original coordination clause. These four steps are repeated for all clauses containing internal coordination. Step-2 in the preprocessing task in Fig. 5 shows the normalisation of Sent-A.

**Construction task**: The construction task relies on the Typed dependencies (TDs) extracted using the Stanford CoreNLP library. We analyse the identified relations within the TDs to group and relate the arguments for each clause within the sentence, and construct the external relations between the identified clauses. This is achieved through the following four steps:

- Relation adaptation: to support the construction of the desired relational graph, we modify the TDs involving copular verbs (e.g. am, is, are). When copular verbs are the main verbs in the sentence, the TDs do not identify their relations in the same manner as normal verbs. Instead, the mention "cop(o?, v?)" in the TDs (mentions are the structures of TDs in Stanford CoreNLP) is used to identify the verb as a copular verb to its related object ("o" in the mention). In addition, the subject is linked to the object not the verb itself using the mention "nsubj(o?, s?)". To adapt and standardise the identified relations, we first identify all "cop(o?, v?)" and "nsubj(o?, s?)" mentions, where the first argument (o?-refers to the object) is the same (i.e. the mentions are for the same object). Then all the mentions with the argument "o?" are replaced with "v?" (similar annotations as in the case of a normal verb). Finally, the mention "obj(v?, o?)" is appended to the typed dependencies to establish an object-verb relation. Step-1 in the construction in Fig. 5 shows the updated TDs and the replaced ones for both Sent-A and Sent-B.
- Noun phrase aggregation: noun phrase relations are identified through the mention headers "amod", "nummod", "det", and "nmod" in the TDs of the CoreNLP library. We analyse these mentions to obtain an aggregated noun phrase (NP). Then, the mentions contributing in the NP aggregation are removed from the TDs. Next, for any mention containing the last word in the aggregated NP (the word initially used to refer to the entire NP), we replace this word with the corresponding aggregated NP. Step-2 in the construction in Fig. 5 shows the final TDs after aggregation while highlighting the aggregated noun phrases, the updated TDs, and the removed TDs. This step aims at simplifying the constructed relational graph by referring to each identified NP as a single entity or node.
- Roots identification: in this step, we identify the roots that will be used to refer to each identified clause within the sentence. We use the unique words of the first argument in the mentions with headers "mark", "nsubj", "nsubjpass", "dobj", and "iobj" (mentions that identify the internal relations to the main verbs within the clauses). The extracted first argument is used as the "Root" of the clause as highlighted in Step-3 in the construction in Fig. 5.
- Relation identification: in this step, we identify both the internal relations in the extracted clauses and the external relations between these clauses. A relation "rel(A2?, A1?)" is identified as an internal relation for any non-root argument "A1" related to the Root argument "A2". External relations are identified when a Root argument "A2?" has a relation to either another Root argument or a non-root argument in another sub-tree. Step-4 in the construction in Fig. 5 shows the constructed relational graphs for both sentences. The final constructed relational graph follows the mapping between the TDs mentions and the relational graph roles shown in Table 6 (e.g. "mark" TD->Root).

### Relational Graph Similarity Measurement Process

In this process, the relational graphs constructed previously for each sentence are compared against one another to measure the similarity between the corresponding requirements as indicated in Algorithm 2. Two main tasks are involved in this process: (1) Internal Relation Similarity, and (2) External Relation Similarity. The developed formula we utilise for the calculation of both the internal and external similarity of the relational graphs gives a value between zero and one (zero being completely different, and one being identical or similar requirements). In this measurement, two trees are similar, if all of their internal and external relations are the same. Algorithm 3 computes the common internal and external relations between two graphs.

**Algorithm 2** similarityAssessment(Sent1, Sent2)

```
RG1 ← constructRelGph(Sent1)
RG2 ← constructRelGph(Sent2)
InRel1 ← getInRel(RG1)
InRel2 ← getInRel(RG2)
totInRel ← max(InRel1, InRel2)
ExtRel1 ← getExRel(RG1)
ExtRel2 ← getExRel(RG2)
totExtRel ← max(ExtRel1, ExtRel2)
commonRG ← getCommonRelGph(RG1, RG2)
PassedInRel ← getInRel(commonRG)
PassedExtRel ← getExRel(commonRG)
if totInRel == 0 then
    X ← 0
else
    X ← PassedInRel / totInRel
end if
if totExtRel == 0 then
    Y ← X
else
    Y ← PassedExtRel / totExtRel
end if
Similarity ← (X+Y)/2
```

Internal similarity measurement: The internal similarity is calculated using the developed formula shown in Eq. 1. This formula gives a value between zero and one for the variable $X$ representing the internal similarity between two requirements $j$, and $k$. The main idea is to get a ratio of the matching internal relations to the maximum number of relations within the relational graphs. InternalMatching($j$, $K$) represents the number of identical Roots having identical internal relations and arguments between two requirements $j$ and $k$. For example, Sent-A and Sent-B in Fig. 5 have four internal matchings because the four identified Root nodes and their internal relations (their arguments) are all identical. "InternalRel(j)" and "InternalRel(k)" represent the total number of identified Roots in each relational graph (also four in case of Sent-A and Sent-B).

**External similarity measurement:** similarly, the developed formula in Eq. 2 shows how the entire external similarity of two graphs is calculated. ExternalMatching($j$, $k$) represents the number of matching external Root links or edges between two requirements $j$ and $k$. A matching is counted only if the link and the connected Roots are identical. ExternalRel($j$) and ExternalRel($k$) represent the total number of external Root links in each relational graph. It is worth noting that, in case the number of the external relations of the two requirements being compared is zero. the external similarity variable $Y$ will be set to the value of the internal similarity. This is done to avoid dividing by zero and keep the correctness of the normalised combined similarity value.

We combine both values to provide a single measure or indication of the similarity between two requirements by taking the average of the internal and external similarities as in Eq. 3. Getting a combined similarity measurement of one means that the two requirements are redundant, and zero means they are completely different within the scope of our measurement. Values in between zero and one give an insight into the degree of inter-dependency or inter-relationship between the two requirements.

$$X = \frac{\text{InternalMatching}(j, K)}{\text{Max}(\text{InternalRel}(j), \text{InternalRel}(k))} \tag{1}$$

$$Y = \frac{\text{ExternalMatching}(j, K)}{\text{Max}(\text{ExternalRel}(j), \text{ExternalRel}(k))} \tag{2}$$

$$\text{Similarity}(j, k) = \frac{(X + Y)}{2}. \tag{3}$$

**Algorithm 3** getCommonRelGph(RG1, RG2)

```
commonRG ← φ
visited ← φ
for tree1 ∈ RG1 do
    CandList ← matchedRoots(tree1, RG2, visited)
    bestT ← φ
    for candT ∈ CandList do
        if candT.InRel.size() == tree1.InRel.size() then
            for i = 1,2, ... tree1.InRel.size() do
                if tree1.InRel[i] ∉ candT.InRel then
                    break
                end if
            end for
        end if
        if i == tree1.InRel.size() then
            matchL ← φ
            for j = 1,2, ... tree1.ExtRel.size() do
                if tree1.ExtRel[j] ∈ candT.ExtRel then
                    matchL.add(tree1.ExtRel[j])
                end if
            end for
            if matchL.size() == candT.ExtRel.size() then
                bestT ← candT
                break
            else if (bestT=φ)∨(matchL.size()>bestT.ExtRel.size()) then
                bestT ← candT
                bestT.ExtRel ← matchL
            end if
        end if
    end for
    if bestT ≠ φ then
        commonRG.add(bestT)
        visited(bestT)
    end if
end for
Return commonRG
```

## Evaluation

### Dataset Description

We evaluate the coverage of our proposed RCM on 162 requirement sentences. These requirements are collected from: (1) papers that introduced different requirement templates and formats in different domains considering different writing styles in [2, 4–6, 11, 31–36], (2) papers that introduced requirement formalisation techniques [7, 13], and (3) online available critical system requirements [42]. The dataset is available online in [1].

Figure 6 presents the percentages of each of the 19 requirement properties (components/sub-components) within the entire dataset. The figure shows that time-based and Hidden-constraints exist in few requirements compared to the key requirement components such as Action, Trigger, and Condition. Overall, the distribution of the properties is biased towards the popular properties that exist in most approaches.

Figure 7 shows the complexity of the 162 requirements (when the number of properties per requirement increases ↑, its complexity increases ↑). We grouped the requirements based on the count of their existing properties. The following examples show two requirements with one and six properties, respectively, where each property is separately underlined:

- the monitor mode shall be initialised to INIT.

[1] Dataset: https://github.com/ABC-7/RCM-Model/tree/master/dataSet.

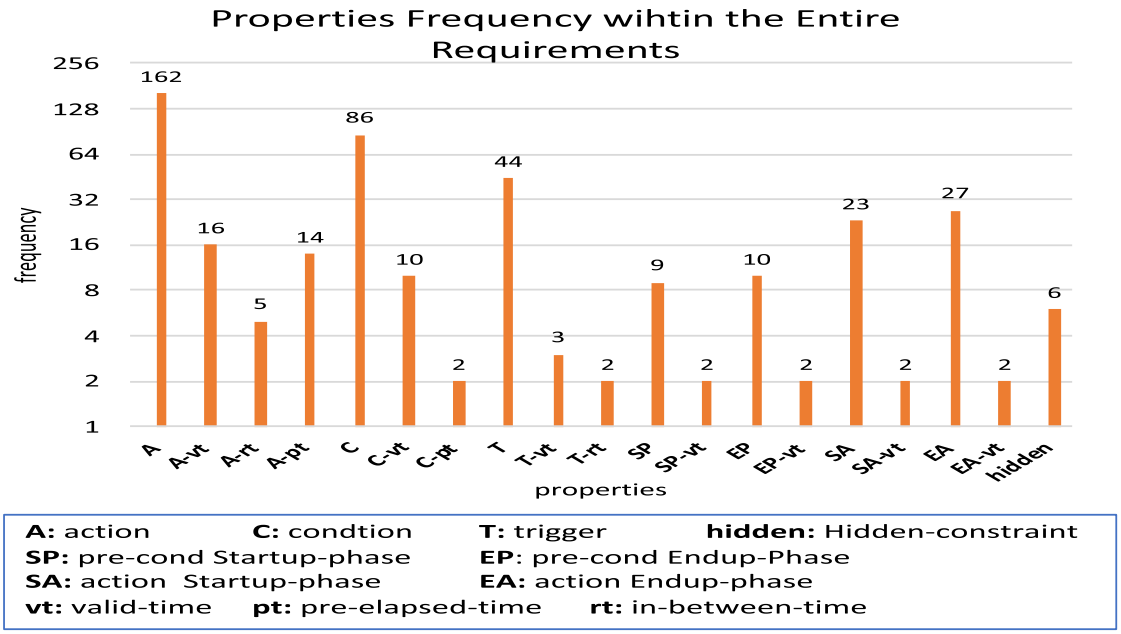



**Fig. 6** Properties frequency within the entire requirements

- after X becomes TRUE for 2 seconds, when Z turns to 1 for 1 second, Y shall be set to TRUE every 2 seconds.

In Fig. 7, each group represents the count of properties regardless of their type (e.g. R1: requirement with Condition and Action, and R2: requirement with Trigger and Action, both have 2 properties). For each group, we calculated the percentage of requirements. Figure 7 presents the properties count used for each requirements group on the *X*-axis and the corresponding requirements percentage on the *Y*-axis. This shows that a large portion of the entire requirements sentences (49% and 22%), only consists of two and three properties, respectively. In contrast, only 20% of the requirements sentences consist of more than three properties. This indicates that most requirements in the dataset are not complex.

## Experiments

**Experiment1. RCM expressiveness**: We evaluated the ability of RCM to capture and represent the requirements in our test dataset compared to 15 exiting approaches in Table 2. To do this, we manually labelled all the requirements in the dataset against the 19 requirement properties we identified in "RCM Development Process". Then, we wrote a Java script to check each requirement (the types of the existing properties) against all the existing approaches to assess if the approach provides a boilerplate or a template that supports

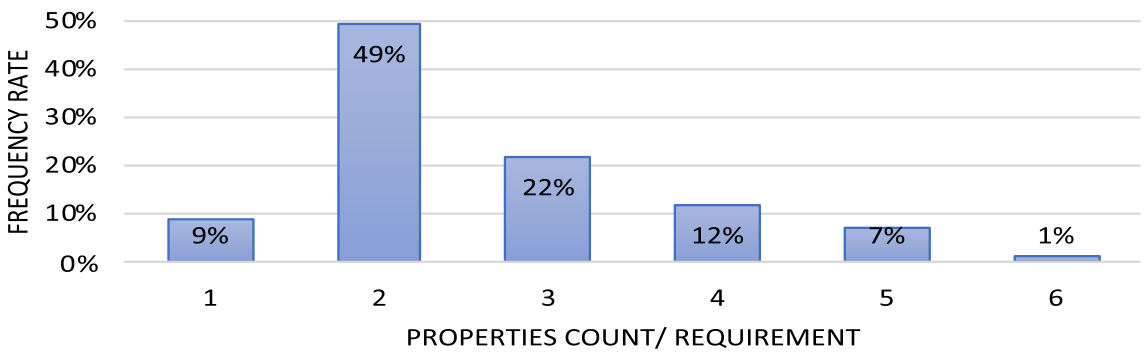



**Fig. 7** Frequency rate of requirements per properties count

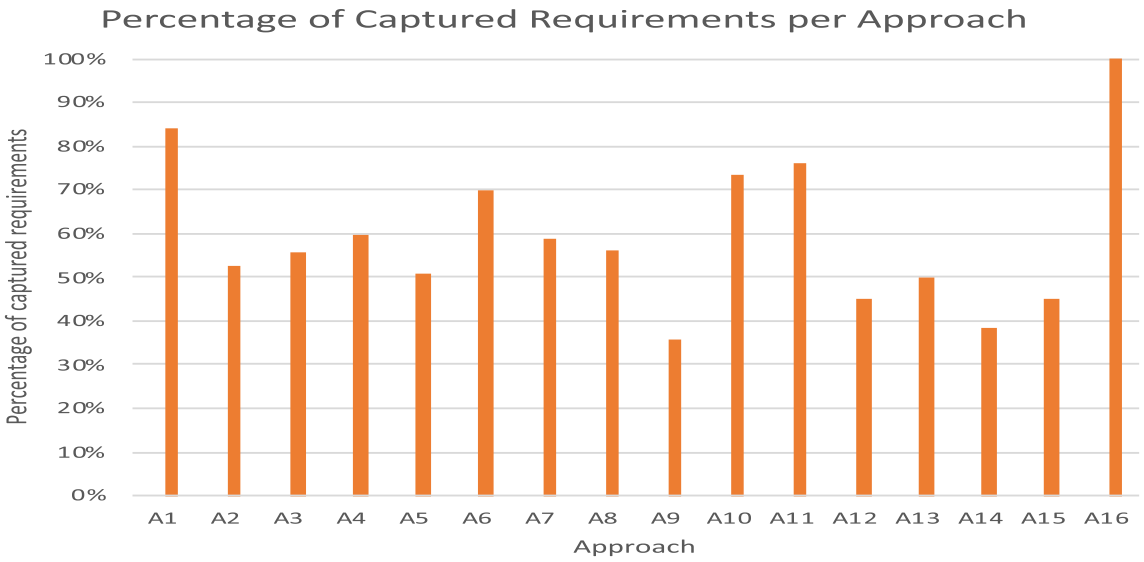



**Fig. 8** Percentage of captured requirements per approach (RCM is represented by A16 and the other approaches are represented by the same symbols used in "RCM Meta Model")

**Table 7** Measured performance of the proposed approach

| Evaluation-type | Correct RCMs | Incorrect RCMs |
|---|---|---|
| Old-manual assessment | 122 | 40 |
| New-manual assessment | 132 | 30 |
| Automatic assessment | 126 | 36 |

representing the requirement or not. The results are available online.[2] Figure 8 summarises the results of our analysis showing percentage of the test requirements that each approach supports.

This shows that none of the existing 15 approaches is able to represent the entire dataset of requirements. This is mainly for two reasons: (1) missing properties in the used templates (e.g. A1 does not support StartUP-phase Preconditional-Scope (SP)), or (2) restrictions on the included properties in a requirement format (e.g. A2:EARS does not support the existence of a Trigger (core-segment) and a Req-Scope (core-segments) using the same format). In addition, ≈ 4% of the test requirements were not covered by any of these approaches combined. An example requirement is "if the maximum deceleration is [insufficient] before a collision with the vehicle ahead, the vehicle warns the driver by acoustical signals for 1 seconds every 2 seconds", where the existing properties are: Condition (core-segment), StartUP-phase Preconditional-Scope (SP core-segment), Action (core-segment), Action valid time (Vt), and Action In-between-time (Rt). These properties do not exist together in the same representation of any of the 15 approaches, see Table 2.

In contrast, our proposed RCM can represent all of the 162 requirements sentences. This is because it covers all the properties that exist in the other approaches and puts no restriction on the included properties in one requirement (i.e. any property can exist in the requirement format).

To use any of the existing approaches, they may require to be extended in two cases: (1) supporting a new requirement property, and (2) supporting a new format (i.e. allowing set of properties to exist together in one requirement sentence regulated by customised grammatical rules). In contrast, RCM covers all the properties of the other approaches and more, and puts no constraints on the properties used in requirements. Hence, it is more expressive and can represent all the requirements that can be represented by the other approaches. It can also be used in other scenarios that are not currently supported by any of the other approaches (because it does not enforce any restrictions on the input requirement formats). Nevertheless, the main limitation of RCM is that it is designed for behavioural requirements of critical systems.

**Experiment2: RCM to formal notations**: We applied our RCM-to-MTL and RCM-to-CTL transformation rules to the dataset of 162 requirements. In this experiment, we used our NLP-approach proposed in [10] to extract RCM from the 162 requirements. We then manually reviewed all

[2] Approaches representations, and evaluation: https://github.com/ABC-7/RCM-Model/blob/master/Approaches-Evaluation.xlsx.

**Table 8** Measured performance of the proposed approach

| TP | FP | TN | FN | Recall | Precision | *F*-measure |
|---|---|---|---|---|---|---|
| 30 | 12 | 120 | 0 | 100% | 83% | 91% |

the extracted RCM models, and fixed all the broken RCM extractions manually. Once we had the full list of 162 RCM structures, we applied the automatic RCM-to-Formal transformation as outlined in "RCM Transformation Algorithm". The full list of RCMs and the corresponding automatically generated MTL and CTL formulas are available online.[3]

We successfully transformed 156 out of the 162 requirement RCM models into MTL notations. The other 6 requirements were partially correct. These 6 requirements turned out to involve Hidden-constraints expressed with ∃ and ∀ properties with a branching structure that is not supported by MTL, since it is a linear version of temporal logic. For example, the requirement "the cognitive threshold of a human observer shall be set to a deviation that is less than 5 [42]" was correctly represented in RCM, but the generated MTL is partially correct "G(the cognitive threshold of a human observer = the deviation)". A correct generation can be "AG((∃ deviation<5) ⟹ (the cognitive threshold of a human observer = deviation))" in CTL notation.

In contrast, CTL can represent requirements with Hidden-constraints correctly, but it provides partial solutions for requirements with time notation (e.g. Valid-time, Pre-elapsed-time and In-between-time). In total, it is capable of representing 120 requirements correctly and provides partial solutions 42 ones due the inclusion of time notation (e.g. the requirement "if air_ok signal is low, auto control mode is terminated within 3 sec" has a partially correct generated CTL formal "AG([air_ok signal = low] ⟹ [auto control mode.crrStatus = terminated])", but a correct formula can be "G([air_ok signal = low] ⟹ [Ft=3(auto control mode.crrStatus = terminated)]" in MTL notation).

**Table 9** Failed requirements analysis

| Id | Matched In-Rel | Matched Ex-Rel | Similarity | Reason of failure |
|---|---|---|---|---|
| 1 | 0.5 | 0 | 0.25 | Stanford interpretation failure |
| 2 | 1 | 0 | 0.5 | Meaning changed as a result of the ordering step in sentence generation algorithm |
| 3 | 1 | 0 | 0.5 | Meaning changed as a result of the ordering step in sentence generation algorithm |
| 4 | 0.33 | 0 | 0.17 | Stanford interpretation failure |
| 5 | 0.67 | 0.5 | 0.58 | Stanford interpretation failure |
| 6 | 0.5 | 0 | 0.25 | Stanford interpretation failure |

[3] RCM-Representation and formal notation: https://github.com/ABC-7/RCM-Model/tree/master/RCM-Auto-Transformation.

## RCM Correctness Evaluation

In this experiment, we automatically evaluate the correctness of the automatically constructed RCMs in [9] for the same 162 input requirements dataset. To decide whether a derived RCM is a correct RCM (i.e. conforming to the input requirement), we utilised the same generation approach proposed in [50]. Then, we applied our proposed similarity approach as follow:

- A relational graph is constructed for each of the two sentences (i.e. the source sentence and the generated one).
- The internal and external similarities of the two constructed graphs are measured and compared. Two sentences A and B are equivalent if all the internal relations within each tree and the external relations between the Root nodes are the same.

First, The conducted experiment to transform NL requirements into RCMs in [10] uses StanfordNLP 3.9.1. In this paper we use StanfordNLP 4.2.0. Hence, we rerun the transformation and manually assessed the obtained results. Then, we utilised the proposed similarity approach for assessing the obtained RCMs automatically through the illustrated two process (relational graphs construction and relational graphs similarity measurements). Table 7 shows the old and new manual assessment for the RCMs compared to the automatic assessment. It shows that the automatic similarity checking identified 126 out of the 132 correct RCMs, in addition to identifying the entire incorrect RCMs.

Table 8 lists the measures for the proposed checking approach:

- TP: # of defected RCMs correctly identified as defected.
- FP: # of correct RCMs identified as defected.
- TN: # of correct RCMs identified as correct.
- FN: # of defected RCMs identified as correct.

The table shows that none of the defected RCMs are marked as correct (i.e. FN = 0), achieving 100% recall. This shows the effectiveness of our approach, where the user does not have to review the RCMs marked as correct. In addition, the approach achieved good performance as only six correct RCMs are marked incorrect (i.e. FP = 6). There are two main reasons behind this failure: (1) Stanford interpretation failure, and (2) meaning mismatch resulting from the RCMs to NL transformation technique [50] as indicated in Table 9. The table shows that, four requirements have mismatch because of Stanford wrong interpretation, while the remaining 2 are because of the NL generation algorithm. It also worth noting that, the reason behind the decrease in precision to 83% is because the total number of incorrect RCMs is relatively small (i.e. TP = 30). Overall, the F-measure of the approach is 91%.

## Summary

We introduced a new requirement capturing model—RCM—for representing safety-critical system requirements. RCM defines a wide range of key requirement properties that may exist in an input requirement. The model allows for standardising the textual requirements extraction process and simplifies the transformation rules to convert requirements into formal notations. We compared the coverage of our RCM model to 15 existing requirements modelling approaches using 162 diverse requirements. Our results show that RCM can capture a wider range of requirements compared to the other approaches because its properties can be customised according to the input requirement. In addition, we provided a suite of RCM-to-MTL transformation rules and presented the corresponding automatically generated MTL and CTL representation of the evaluation dataset.

**Acknowledgement** Osama and Zaki-Ismail are supported by Deakin PhD scholarships. Grundy is supported by ARC Laureate Fellowship FL190100035.

**Funding** Not applicable.

## Declarations

**Conflict of Interest** The authors declare that they have no conflict of interest.

## References

1. Buzhinsky I. Formalization of natural language requirements into temporal logics: a survey. In: 2019 IEEE 17th international conference on industrial informatics (INDIN), vol. 1. IEEE; 2019. p. 400–6.
2. Lúcio L, Rahman S, Cheng C-H, Mavin A. Just formal enough? Automated analysis of ears requirements. In: NASA formal methods symposium. Berlin: Springer; 2017. p. 427–34.
3. Sládeková V. Methods used for requirements engineering. Master's thesis, Univerzity Komenského; 2007.
4. Justice B. Natural language specifications for safety-critical systems. Master's thesis, Carl von Ossietzky Universität; 2013.
5. Mavin A, Wilkinson P, Harwood A, Novak M. Easy approach to requirements syntax (ears). In: Requirements engineering conference, 2009. RE'09. 17th IEEE international. Aug 31-Sep 4; Atlanta, USA; Piscataway; 2009. p. 317–22.
6. Fuchs NE, Schwitter R. Attempto controlled English (ACE). In: CLAW 96, first international workshop on controlled language applications; Leuven, BE, March, 1996.
7. 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). IEEE; 2015. p. 1677–82.
8. Rupp C. Requirements-Engineering und-Management: Professionelle, Iterative Anforderungsanalyse Für die Praxis. Munich: Hanser Verlag; 2009.
9. Zaki-Ismail A, Osama M, Abdelrazek M, Grundy J, Ibrahim A. RCM: requirement capturing model for automated requirements formalisation. In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development; —volume 1: MODELSWARD. SciTePress; 2021. INSTICC.
10. 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: Proceedings of the 9th international conference on model-driven engineering and software development—volume 1: MODELSWARD. SciTePress; 2021. p. 270–7. https://doi.org/10.5220/0010270602700277. INSTICC
11. Teige T, Bienmüller T, Holberg HJ. Universal pattern: formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In 19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachenzur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'16),p. 6–9. Albert-Ludwigs-Universit¨at Freiburg, 2016.MBMV.
12. Konrad S, Cheng BH. Real-time specification patterns. In: Proceedings of the 27th international conference on software engineering. New York, NY, USA: ACM; 2005. p. 372–81.
13. 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. Berlin: Springer; 2016. p. 41–6.
14. Nelken R, Francez N. Automatic translation of natural language system specifications into temporal logic. In: International conference on computer aided verification. Berlin: Springer; 1996. p. 360–71.
15. Michael JB, Ong VL, Rowe NC. Natural-language processing support for developing policy-governed software systems. In: TOOLS 39. 39th international conference and exhibition on technology of object-oriented languages and systems, Santa Barbara, CA: 2001. IEEE; 2001. p. 263–74.
16. Holt A, Klein E. A semantically-derived subset of English for hardware verification. In: Proceedings of the 37th annual meeting of the association for computational linguistics on computational linguistics. Association for Computational Linguistics; Maryland, USA; 1999. p. 451–6.
17. Ambriola V, Gervasi V. Processing natural language requirements. In: Automated software engineering, 1997. Proceedings, 12th IEEE international conference. Lake Tahoe, NV; 1997. p. 36–45.
18. Sturla G. A two-phased approach for natural language parsing into formal logic. PhD thesis, Massachusetts Institute of Technology; 2017.

19. R. Poli, M. Healy, A. Kameas (Eds.). Controlled English to logic translation. In: Theory and applications of ontology: computer applications. Berlin: Springer; 2010. p. 245–58.
20. Kittredge RI. Sublanguages and controlled languages. In Ruslan-Mitkov (ed.). The Oxford handbook of computational linguistics, 2nd edn. Oxford: Oxford University Press; 2003.
21. Kuhn T. A survey and classification of controlled natural languages. Comput Linguist. 2014;40(1):121–70.
22. Van Kleek M, Moore B, Karger DR, André P, Schraefel M. Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web. In: Proceedings of the 19th international conference on world wide web; 2010. p. 951–60.
23. Schwitter R. English as a formal specification language. In: Proceedings. 13th international workshop on database and expert systems applications. Aix-en-Provence, France: IEEE; 2002. p. 228–32.
24. Erdem E, Yeniterzi R. Transforming controlled natural language biomedical queries into answer set programs. In: Proceedings of the BioNLP 2009 workshop; Boulder, CO. 2009. p. 117–24.
25. Scott W, Cook SC, et al. A context-free requirements grammar to facilitate automatic assessment. PhD thesis, UniSA; 2004.
26. Esser M, Struss P. Obtaining models for test generation from natural-language-like functional specifications. In: International workshop on principles of diagnosis; 2007. p. 75–82.
27. Arora C, Sabetzadeh M, Briand L, Zimmer F, Gnaga R. Rubric: a flexible tool for automated checking of conformance to requirement boilerplates. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM; 2013. p. 599–602.
28. Arora C, Sabetzadeh M, Briand LC, Zimmer F. Requirement boilerplates: transition from manually-enforced to automatically-verifiable natural language patterns. In: 2014 IEEE 4th international workshop on requirements patterns (RePa). IEEE; 2014. p. 1–8.
29. Dwyer MB, Avrunin GS, Corbett JC. Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on software engineering; Los Angeles, 1999. p. 411–20.
30. Qiao Y, Zhong K, Wang H, Li X. Developing event-condition-action rules in real-time active database. In: Proceedings of the 2007 ACM symposium on applied computing; 2007. New York, p. 511–6.
31. Jeannet B, Gaucher F. Debugging embedded systems requirements with stimulus: an automotive case-study. In: 8th European Congress on embedded real time software and systems (ERTS); Toulouse, France, Jan 2016.
32. Thyssen J, Hummel B. Behavioral specification of reactive systems using stream-based I/O tables. Softw Syst Model. 2013;12(2):265–83.
33. Fifarek AW, Wagner LG, Hoffman JA, Rodes BD, Aiello MA, Davis JA. Spear v2. 0: formalized past LTL specification and analysis of requirements. In: NASA formal methods symposium. Berlin: Springer; 2017. p. 420–6.
34. Lúcio L, Rahman S, bin Abid S, Mavin A. EARS-CTRL: generating controllers for dummies. In: MODELS (satellite events); 2017. p. 566–70.
35. Dick J, Hull E, Jackson K. Requirements engineering. Berlin: Springer; 2017.
36. Bitsch F. Safety patterns-the key to formal specification of safety requirements. In: International conference on computer safety, reliability, and security. Berlin: Springer; 2001. p. 176–89.
37. Rolland C, Proix C. A natural language approach for requirements engineering. In: International conference on advanced information systems engineering. Berlin: Springer; 1992. p. 257–77.
38. Macias B, Pulman SG. A method for controlling the production of specifications in natural language. Comput J. 1995;38(4):310–8.
39. 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). IEEE; 2017. p. 1477–81.
40. 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. IEEE; 2015. p. 143–50.
41. Berger P, Nellen J, Katoen J-P, Abraham E, Waez MTB, Rambow T. Multiple analyses, requirements once: simplifying testing & verification in automotive model-based development; 2019. arXiv preprint. arXiv:1906.07083.
42. Houdek F. System requirements specification automotive system cluster (elc and acc). Munich: Technical University of Munich; 2013.
43. Zaki-Ismail A, Osama M, Abdelrazek M, Grundy J, Ibrahim A. RCM-extractor: an automated NLP-based approach for extracting a semi formal representation model from natural language requirements. Autom Softw Eng. 2022;29(1):1–33.
44. Zaki-Ismail A, Osama M, Abdelrazek M, Grundy J, Ibrahim A. ARF: automatic requirements formalisation tool. In: 2021 IEEE 29th international requirements engineering conference (RE). Notre Dam, South Bend, USA; 2021. p. 440–1.
45. Haider A. A survey of model checking tools using LTL or CTL as temporal logic and generating counterexamples. https://doi.org/10.13140/RG.2.1.3629.1925
46. 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) 2019. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik; 2019.
47. Alur R, Henzinger TA. Real-time logics: complexity and expressiveness. Inf Comput. 1993;104(1):35–77.
48. Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching time temporal logic. In: 25 Years of model checking. Berlin: Springer; 2008. p. 196–215.
49. Konur S. A survey on temporal logics for specifying and verifying real-time systems. Front Comput Sci. 2013;7(3):370–403.
50. 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: Proceedings of the 33rd international conference on software engineering and knowledge engineering; Pittsburgh, USA, Jul 2021.
51. Gatt A, Reiter E. SimpleNLG: a realisation engine for practical applications. In: Proceedings of the 12th European workshop on natural language generation (ENLG 2009); Stroudsburg, PA, USA, 2009. p. 90–93.


**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.