RCM: Requirement Capturing Model for Automated Requirements Formalisation

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*

Keywords: Requirement Representation, Requirement Modeling, Requirement Engineering, Requirement Formalisation.

Abstract: Most existing automated requirements formalisation techniques require system engineers to (re)write their requirements using a set of predefined requirement templates with a fixed structure and known semantics to simplify the formalisation process. However, these techniques require understanding and memorising requirement templates, which are usually fixed format, limit requirements captured, and do not allow capture of more diverse requirements. To address these limitations, we need a reference model that captures key requirement details regardless of their structure, format or order. Then, using NLP techniques we can transform textual requirements into the reference model. Finally, using a suite of transformation rules we can then convert these requirements into formal notations. In this paper, we introduce the first and key step in this process, a Requirement Capturing Model (RCM) - as a reference model - to model the key elements of a system requirement regardless of their format, or order. We evaluated the robustness of the RCM model compared to 15 existing requirements representation approaches and a benchmark of 162 requirements. Our evaluation shows that RCM breakdowns support a wider range of requirements formats compared to the existing approaches. We also implemented a suite of transformation rules that transforms RCM-based requirements into temporal logic(s). In the future, we will develop NLP-based RCM extraction technique to provide end-to-end solution.

1 INTRODUCTION

Formal verification techniques requires system requirements to be expressed in formal notations (Buzhinsky, 2019). 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 (Lúcio et al., 2017b; Sládeková, 2007). To automate the formalisation process, several bodies of work within the literature focused on proposing pre-defined requirement templates, patterns (Justice, 2013), boilerplates (Mavin et al., 2009), and structured control English (R. S. Fuchs, 1996), to express one system requirement sentence while eliminating the ambiguities. Such templates have 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., (Yan et al., 2015)). However, some of the predefined templates are domain dependent and are hard to generalise (Rupp, 2009), or can only capture limited subsets of requirements structures (R. S. Fuchs, 1996). In addition, most existing formalisation algorithms are customized 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.

Complementary to this research direction, instead of considering introducing new sentence-based templates covering a wider range of requirements and complicating the requirements specification process, we introduce a Requirement Capturing Model (RCM), as a reference model that defines the key properties that make up a system behavioral requirement sentence, *regardless of the syntactic structure of these properties, lexical-words, or their order*. RCM separates the writing styles (format and structure)

[a] https://orcid.org/0000-0002-1580-6619
[b] https://orcid.org/0000-0002-6940-0833
[c] https://orcid.org/0000-0003-3812-9785
[d] https://orcid.org/0000-0003-4928-7076
[e] https://orcid.org/0000-0001-8747-1419

Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J. and Ibrahim, A.
RCM: Requirement Capturing Model for Automated Requirements Formalisation.
DOI: 10.5220/0010270401100121
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 110-121
ISBN: 978-989-758-487-9

from the abstract requirement properties and the formal notations. Our new RCM model thus 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 rewriting existing requirements; (3) formalize 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. Our key contributions in this paper are:

- Introduce RCM as a reference model and intermediate representation between informal and formal notations.
- A suite of transformation rules from RCM to Metric Temporal Logic (MTL), to demonstrate how an RCM-based requirements model can be transformed into formal notations.
- Evaluating the RCM representation power by comparing it to 15 other existing approaches using 162 behavioral requirements for critical systems. We provide the RCM representation and corresponding automatically generated (MTL and CTL) formal notations for the 162 requirements.

2 MOTIVATION

Jen is a system 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 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 existing templates.

Jen found that none of the existing techniques she found can 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 of these solutions rely on pre-defined 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, (3) a fixed/small set of English verbs or other lexical words. Thus, Jen needs to rewrite her requirements to confirm the defined format which puts more overhead on her especially if the defined formats are limited and cannot be extended to new scenarios.

Table 1: Examples of critical system requirements and approaches to represent.

RQ1: R_STATUS shall indicate the rain sensor. It shall be ON, when the external environment is raining. **Techniques:** Universal pattern (Teige et al., 2016), Structured English (Konrad and Cheng, 2005), Rup's boilerplates (Rupp, 2009), ACE(R. S. Fuchs, 1996), EARS(Mavin et al., 2009), CFG(Sládeková, 2007) and BTC(Justice, 2013)
RQ2: When the external environment rains for 1 minute, the wipers shall be activated within 30 seconds until the rain sensor equals OFF. **Techniques:** Universal pattern (Teige et al., 2016) and BTC(Justice, 2013)
RQ3: While the wipers are active, the wipers speed shall be readjusted every 20 seconds. **Techniques:** Structured English (Konrad and Cheng, 2005)

Taking into consideration all combinations of styling, ordering, and omission/existence of different requirements model properties will increase the size of the defined formats. Consequently this will increase the complexity of using them by system engineers and the complexity of the parsing algorithms needed to transform them to formal models. Furthermore, most existing formalization techniques apply on-the fly transformation on the given structured requirement sentences to generate formal notations. These transformations are hard-coded or tightly customized according to the target formal notation properties and formats. It would be much more useful if the common parts are computed once and transformed to intended notations as needed.

3 RELATED WORK

Many requirements formalisation approaches assume requirements are specified in a constrained natural language (CNL) with specific style, format and structure to be able to transform into formal notations - e.g. (Ghosh et al., 2016; Nelken and Francez, 1996; Michael et al., 2001; Holt and Klein, 1999; Ambriola and Gervasi, 1997; Sturla, 2017; Pease and Li, 2010). These CNL are meant to avoid natural language related quality problems (e.g., ambiguity inconsistency, etc.) and increase the viability of automating the formalisation process.

CNL is a restricted form of NL, created for writing technical documents as defined in (Kittredge, 2003) with the aim to reduce/avoid NL problems (e.g., ambiguity inconsistency, .etc). CNL typically has a defined sub-set of NL grammar, lexicon and/or sentence structure (Kuhn, 2014). Different forms of CNL are also provided as a reliable solution for requirements representation. Fuchs et al. (R. S. Fuchs, 1996) propose Attempto Controlled English (ACE) with a restricted list of verbs, nouns and adjectives for the requirement set in addition to restrictions on the structure of the sentence. ACE can be transformed into Prolog. It can handle requirements with condition and action components. Multiple CNLs proposed later inspired by ACE (e.g., Atomate language (Van Kleek et al., 2010), PENG(Schwitter, 2002)) for formal generation purposes and for other purposes (e.g., BioQuery-CNL (Erdem and Yeniterzi, 2009)).

Similarly to ACE, Scott and Cook(Scott et al., 2004) presented Context Free Grammars (CFGs) for requirement specification. Although the format of the requirement components is more limited than ACE with additional restrictions on words, it covers time-related properties. Yan et al. (Yan et al., 2015) 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 adjective or adverbial word, (3) prepositional phrases are not allowed except "in + time point" at the end of the clause. The CNL does not consider time information except pre-elapsed time.

Boilerplates are also widely used. These provide a fixed syntax and lexical words with replaceable attributes. Boilerplates are more limited than CNL and require adaptation to different domains. In (Rupp, 2009), the provided RUP's boilerplate can handle a limited range of requirements. EARS (Mavin et al., 2009) boilerplates are less restricted and can support a wider range of requirements. Esser et al. (Esser and Struss, 2007) proposed a suite of requirement templates (TBNLS) with support mapping to propositional logic with temporal relations. For validating the conformity of the written requirement and the boilerplate, authors in (Arora et al., 2013; Arora et al., 2014) provide checking techniques.

Requirement patterns provide a more flexible solution. However, When a new requirement structure is added, a new pattern should be created for it, which increases the size of the patterns set. In (Teige et al., 2016) a universal pattern was presented to support many requirements formats (trigger, then action). They then introduced additional time-based kernel patterns in (Justice, 2013). Although these patterns cover many requirement properties, they do not still cover the possible combinations of the supported properties eligible to one requirement specification. In addition, the approach lack complex time properties - e.g. In-between-time and pre-elapsed-time properties. Dwyer et al. (Dwyer et al., 1999) proposed several patterns applicable for non-real-time requirement specifications. These patterns are categorized into two major groups: occurrence patterns and order patterns, while considering scopes (e.g., globally, before R, after R) for a given specification pattern. The work is extended later in (Konrad and Cheng, 2005) to cope with real-time requirement specifications. The real time patterns considers 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 behavioral requirements. ECA became widely used by several researchers in diffident areas. An ECA rule assumes that when an event E occurs, the condition C will be evaluated, and if true, the action A will be executed. ECA notations have been extended to capture time information (Qiao et al., 2007). However, ECA rules do not support (*e.g.*, factual rules), and do not consider scopes for action and the time notations apply on events.

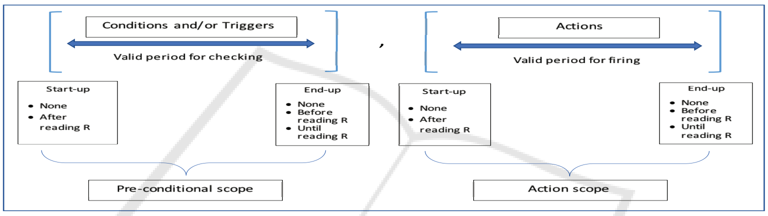
4 REQUIREMENT CAPTURING MODEL

In this section, first, we explain the process we followed to develop the RCM. Then, we describe the RCM metamodel in details. Finally, we provide RCM to formal notations transformation procedure.

4.1 RCM Development Process

To identify the key requirement properties we needed to support in a generic reference model for safety-critical requirements, we reviewed a large number of natural language-based critical system requirements collected from many sources: (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; Rolland and Proix, 1992; Macias and Pulman, 1995) and 15 requirement representation approaches listed in Table.3.

Table 2: A list of identified requirement properties from existing approaches.

Property		Description
Component	**Trigger**	is an event that initiates action(s) (e.g., "when the system halts" in Figure 3). This component type is ubiquitous throughout the requirements of most critical systems.
Component	**Condition**	is a constraint that should be satisfied to allow a specific system action(s) to happen (*e.g.,* "if X is ON" in Figure 3). 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 checked for.
Component	**Action**	is a task that should be accomplished by the system in response to triggers and/or constrained by conditions (e.g., "M should be set to TRUE" in Figure 3). In case that, a primitive requirement consists of an action component only, it would be marked as **a factual rule** expressing factual information about the system (e.g., *The duration of a flashing cycle is 1 second* (Houdek, 2013)).
Component	**Req-scope**	determines the context under which (i) "condition(s) and trigger(s)" can be valid – called a pre-conditional scope as it is linked to the condition or trigger; and (ii) "action(s)" can occur – called an action scope, as it applies only on the action. The scope may define the starting boundary or the ending one (e.g., "after sailing termination", "before <B_sig> is True" in Figure 3). Figure 3 presents the main variations for starting/ending a context (*e.g.,* None, after operational constraint is true, until operational constraint becomes true or before operational constraint becomes true). Other alternatives can be expressed by the main variation. 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" does not care about their status. 
Sub-Component	**Valid-time**	represent the valid time period of the given component (e.g., in "the vehicle warns the driver by acoustical signals $< E >$ for 1 second" the action is hold for 1 second length of time (Houdek, 2013)). Valid-time can be a part of any component.
Sub-Component	**Pre-elapsed-time**	is the consumed time length from an offset point –before an action to occur or a condition to be checked (e.g., "After less than 2 seconds" in Figure 3). This type is only eligible to action and condition components.
Sub-Component	**In-between-time**	express the length of time between two consecutive events to occur in the repetition case (e.g., "every 1 seconds" in Figure 3). Such sub-component type is eligible to action and trigger components as indicated in Figure 2.
Sub-Component	**Hidden constraint**	allows an explicit constraint to be defined on a specific operand within a component. For example, in "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 second"(Houdek, 2013). The **that is activated** is a constraint defined on the operand *the high beam headlight*).

We identified 19 distinct properties that we grouped into 8 abstract properties (4 components and 4 sub-components). These are listed with their description in Table.2. Figure 1 shows a manually crafted example requirement that reflects most of these components and sub-components used through the properties description for a better understanding.

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.

Figure 1: Crafted multi-sentence requirement "REQ".

We then analysed 15 of the existing approaches (outlined in the related work section) against these 19 requirement properties as presented in Table.3. The approaches (rows) are encoded A1 to A15, and requirement properties are encoded as columns. An approach can be represented in more than one row. This reflects that some approaches might support multiple properties, but these properties cannot be used in the same requirement – the template or pattern does not support having certain properties in one requirement. The cell value equals "1" if the property is supported in this template.

This table does not reflect the limitations/restrictions that these approaches apply on a given property formatting or order - i.e. condition must come before action, or scope comes before condition. Our analysis of this table illustrates that: (1) no approach covers all requirement properties possibly because this would make it too complex to use; (2) almost all approaches support action components as a core element; (3) approaches: A1 and A11 are the most expressive approaches as they cover majority of the properties; and (4) the valid-time property for the StartUp and the EndUP

Table 3: Exisiting approaches proposed properties and Supported formats.

Properties Codes → **A:**action / **C:**condtion / **T:**trigger / **hidden:**Hidden-constraint / **SP:**pre-cond Startup-phase / **EP:**pre-cond Endup-Phase / **SA:**action Startup-phase / **EA:**action Endup-phase / **vt:**valid-time / **pt:**pre-elapsed-time / **rt:**in-between-time

Approach		Requirement properties																		
		Action				Condition			Trigger			Req-Scope								
Code	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	Hidden
A1	BTC (Justice, 2013; Teige et al., 2016)	1	1		1	1	1					1				1	1	1	1	
		1	1		1				1	1		1				1	1	1	1	
A2	EARS (Mavin et al., 2009)	1				1			1											
		1				1						1								
		1							1			1								
A3	EARS-CTRL (Lúcio et al., 2017b)	1							1			1						1		
A4	ECA (Van Kleek et al., 2010)	1			1	1	1	1	1	1	1	1								
A5	boilerplates (Rupp, 2009)	1				1														
		1							1											
A6	Safety templates (Fu et al., 2017)	1		1	1											1		1		
		1		1	1	1										1		1		
		1							1		1	1		1						
A7	Req Lang (Marko et al., 2015)	1	1			1			1			1		1						
A8	CFG (Scott et al., 2004; Sládeková, 2007)	1				1														1
		1							1											1
		1														1				1
		1																1		1
A9	ACE (R. S. Fuchs, 1996)	1				1														1
A10	PENG (Schwitter, 2002)	1				1			1			1		1		1		1		1
A11	Structured English (Yan et al., 2015)	1			1	1			1			1		1		1		1		
A12	TBNLS (Esser and Struss, 2007)	1	1			1	1					1		1				1		
A13	Real-time (Konrad and Cheng, 2005)	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	Dawyer (Dwyer et al., 1999)	1				1						1		1				1		
A15	Pattern_based(Berger et al., 2019)	1	1		1	1	1													

phases of the pre-conditional scope is not supported by any of these approaches although its appearance in the analysed requirements.

4.2 RCM Domain Model

The RCM is designed to capture the requirements properties listed above while relaxing the ordering and formatting restrictions presented by the existing techniques. In RCM, a system is represented as a set of requirements R. Each requirement R_i represented by one RCM and may have one or more primitive requirements PR where $\{R_i = < PR_n >$ and n>0$\}$. Each PR_j represents only one requirement sentence, and may include condition(s), trigger(s), action(s) and requirement scope(s). The detailed meta-model of the RCM to one requirement R_i is presented in Figure 2.

The figure shows that a primitive *requirement* is composed of four requirement *component* types: *condition*, *trigger*, *action* and *requirement scope*. Except for action(s), the existence of each of these components is optional in a primitive requirement. A requirement component has a *component core-segment* that expresses the main portion of the component, and optionally could also have a *valid-time*: the component's valid time-length. The *pre-elapsed-time* sub-component can only appear with a condition or action component. An *in-between-time* sub-component can only appear with Trigger or Action components according to the reviewed scenarios (e.g., requirements and representation formats). A *hidden-constraint* is an optional sub-component to an operand. To store this information without loss, RCM stores the hidden constraint inside the relevant operand object as indicated in Figure 2. This structure is intrinsic to allow the nested hidden constraints. For example, "*the entry of A1* ***whose index is larger than the first value in A2*** *that is larger than S1 shall be set to 0*".

All five sub-components are instances of either *Predicate* or *Time* structure. The *Predicate* structure consists of the operands, the operator and negation flag/property (e.g., in "if X exceeds 1" the "X" and "1" are the operands and "exceeds" is the operator in the semi-formal semantic and ">" is the operator in the formal semantic). The *Time* structure stores the unit, value and quantifying relation (e.g., "for less than 2 seconds", "2" and "seconds" are the unit and value respectively, "less than" is the semi-formal quantifying relation whose formal semantic is "<"). Since the *Predicate* and *Time* structures are the infrastructure of the entire properties, they are designed to encapsulate the semi-formal and formal semantic allowing mappability to multiple TL. The details of formal semantic are described in section.4.3.2.

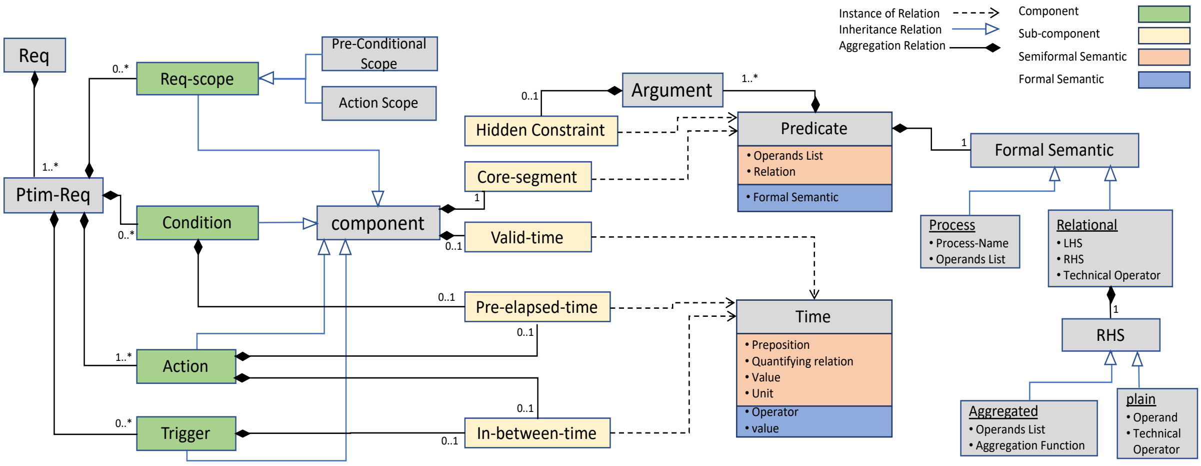


Figure 2: RCM meta-model (simplified).

Components with the same type can be stored as a tree –the most suitable to keep nested relation appropriately, where leafs are the components, and inner nodes are coordinating relationships (e.g., check the conditions components of PR[1] in Figure 3).

Figure 3, shows the RCM representation of the REQ example. It has two primitive requirements: PR[1] and PR[2]. Components of each primitive requirement are presented in separate blocks in the figure. In each block, sub-components are separated by horizontal line. Figure 3 highlights the encapsulation of semi-formal semantic (in black) and formal semantic (in red). Components with the same type (e.g., conditions in PR[1]) are represented by tree structure. The figure also provides the the corresponding MTL representation, see subSection.4.3.3.

4.3 RCM Transformation

In this section, we illustrate transformation into temporal logic (TL)- as an example of formal notations. We first illustrate: (1) the mapping between the RCM to TL, and (2) the formalization of the RCM infrastructure (i.e., Predicate and Time structures). Then, we provide the transformation process.

4.3.1 RCM and Temporal Logic

In order to formally model a given requirement represented by RCM in temporal logic (TL), we have to define a set of transformation rules. A TL formula F_i is built from a finite set of proposition variables AP by making use of boolean connectives (e.g., ”AND”, ”OR”) and the temporal modalities (e.g., U (until)) (Haider, 2015; Brunello et al., 2019). Within such formula, each proposition letter is expressed by a true/false statement 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(Brunello et al., 2019)”. Such sentence can be captured by the following TL formula:

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

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

We use the following to build the mapping between RCM and TL:

1. **Propositions and Time Notations:** Given that, RCM components and sub-components are expressed as predicates or time structures as indicated in Figure 2. These structures are eventually mapped to proposition and time notations in the corresponding temporal logic formula (e.g., the action component ”**M shall Transition to TRUE** <u>after less than 2 seconds</u>” mapped to ”$F_{t<2}$(S)”, where S and ”t<2” represent the predicate in bold and time phrase underlined).
2. **Coordinating Relations:** The booleans connecting propositions can be obtained from coordinating relations connecting multiple components with the same types. Such relations are represented by tree for each component type as discussed before (e.g., the condition components ”X is ON for 1 second or (Y is ON and Z is ON)” 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 ”pre-conditional-scope startup-phase” mapped to ”$\Longrightarrow$”

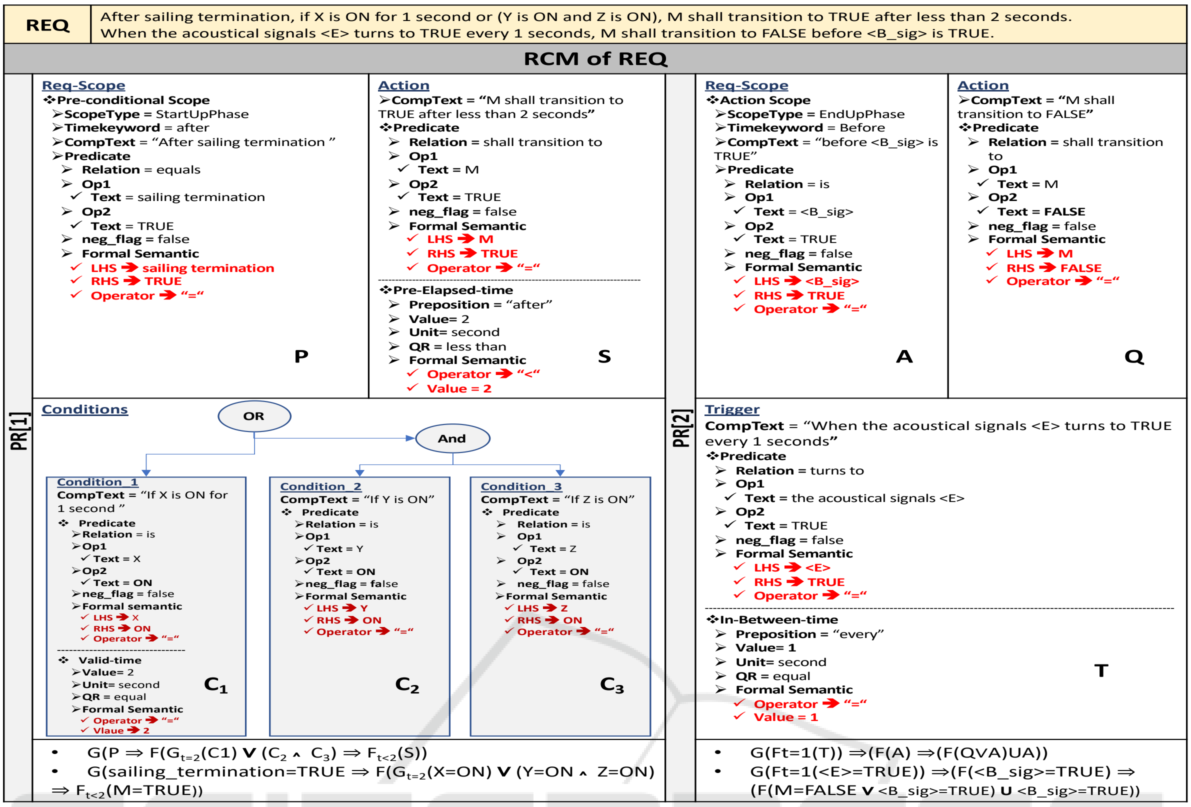


Figure 3: An example presents multi-sentence requirement "REQ" and the corresponding RCM representation.

Table 4: RCM mapping to MTL & CTL.

RCM					TL Mapping		
Properties (component/ subcomponents)			Versions	Applicable on	MTL	CTL	
Action		1	A: do something		A	A	
Pre-condition	Condition	2	If S	Action (*P in mapping*)	$G(S \implies P)$	$AG(S \implies P)$	Temporal Modality
	Trigger	3	When S		$G(S \implies P)$	$AG(S \implies P)$	
	Conditions and triggers	4	When S, IF Q		$G((S \wedge Q) \implies P)$	$AG((S \wedge Q) \implies P)$	
Req-Scope: (Preconditional-Scope / Action-Scope)	StartUP	5	After S	Precondition/ action (*P in mapping*)	$G(S \implies F(P))$	$AG(S \implies AG(AF(P)))$	
	EndUP	6	Before S		$F(S) \implies (F(P \vee S)\mathbf{U}S)$	$A[((AF(P \vee S)) \vee AG(\neg S))WS]$	
		7	Until S		$F(P)US$	$AF(P)US$	
	StartUP and EndUp	8	-After Q & Before S -Between Q and S		$G((Q \wedge \neg S \wedge F(S)) \implies F(P \vee S)\mathbf{U}S))$	$AG((Q \wedge \neg S) \implies A[((AF(P \vee S)) \vee AG(\neg S))\mathbf{W}S])$	
		9	-After Q & Until S - While Z {Q=Z& S=¬ Z}		$G((Q \wedge \neg S) \implies F(P\mathbf{U}S))$	$AG((Q \wedge \neg S) \implies A[(AF(P \vee S)\mathbf{W}S])$	
Pre-elapsed-time		10	After c time	Condition/ Action (*P in mapping*)	$F_{t=c}(P)$		Time notation
		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		$F_{t>c}(P)$		
Validation-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		$G(F_{t>c}(P))$		
Hidden-Constraint		25	Whose S	P is Any component		$AG(\exists S \implies P)$	branching

To demonstrate the robustness of the RCM and capability to transform to different formal notations, we provide here a mapping into two examples of temporal logic, MTL (Alur and Henzinger, 1993) and CTL (Clarke and Emerson, 2008), as shown in Table 4 as a proof of multiple map-ability. We chose these notations as they are widely used in model checking as indicated in (Konur, 2013) and (Haider, 2015) respectively. We base our temporal-modality and time-notation mapping on the mapping done in (Konrad and Cheng, 2005).

The first column in Table.4 shows the RCM properties (components and sub-components) employed in formal roles, each attached with alternatives if any (e.g., The pre-conditions may be conditions, triggers, or both of them based on the given requirement). Possible structures corresponding to each property version are listed in the third column (i.e., the used keywords (e.g., when) are just examples, any replaceable keyword could be used). 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.

4.3.2 RCM and Formal Semantics

Temporal logic has multiple versions exhibiting slight differences. In order to support the transformation to multiple versions with minimal adjustment in the transformation technique, RCM encapsulates formal semantics with semi-formal semantics. Design-wise, RCM augments the formal semantic in the basic units, predicate and time structures in Figure 2, that are mapable to temporal logic, as indicated in the previous subsubsection. The formal semantic of a predicate covers three formats:

- **Process Format:** is suitable to predicates express functions or process (e.g., "the monitor sends a request *REQ_Sig* to the station" $\longrightarrow$ "send(the_monitor, the_station,*REQ_Sig*)").
- **Relational Format with Plain RHS:** the type is suitable for assignment predicates (e.g., "set X to True" $\longrightarrow$"X = True"), comparison predicates (e.g., "If X exceeds Y" $\longrightarrow$"X $>$ Y") and changing state predicates (e.g., "the window shall be moving up" $\longrightarrow$"the_window = moving-UP").
- **Relational Format with Aggregated RHS:** this format is similar to the previous one but the RHS is expressed with aggregating function (e.g., "If the fuel level is less than the min value of Thr1 and Thr2" $\longrightarrow$"the_fuel_level $<$ "min(Thr1, Thr2)").

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

4.3.3 RCM Transformation Algorithm

To accomplish the automatic transformation from RCM-to-MTL, we use the mapping rules provided in Table 4 on the obtained formal semantics of the given primitive requirements. Algo.1 shows the automatic transformation pseudo-code annotated in Figure 4 with each step output for PR[1] in the REQ Figure 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{9:25})
        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 Formulat
    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 Subsection.4.3.2. Then, we compute the

Step1: ✓ sailing_termination=TRUE ✓ $G_{t=2}$(X=ON) ✓ Y=ON ✓ Z=ON ✓ $F_{t<2}$(M=TRUE)	**Step2:** ✓ sailing_termination=TRUE ✓ $G_{t=2}$(X=ON) ∨ (Y=ON ∧ Z=ON) ✓ $F_{t<2}$(M=TRUE)
Step3: ✓ preConds: $G_{t=2}$(X=ON) ∨ (Y=ON ∧ Z=ON)	**Step4:** ✓ lHS: G(sailing_termination=TRUE ⇒ F($G_{t=2}$(X=ON) ∨ (Y=ON ∧ Z=ON)) ✓ rHs: $F_{t<2}$(M=TRUE)
Step5: ✓ mTLFormula: G(sailing_termination=TRUE ⇒ F($G_{t=2}$(X=ON) ∨ (Y=ON ∧ Z=ON) ⇒ $F_{t<2}$(M=TRUE))	

Figure 4: Step by Step generation of PR[1] from Figure 3.

formal semantics of the entire tree (i.e., leaf nodes represent components and inner nodes represent logical relations as discussed before) of each component type through the recursive function aggRel. After that, we construct the main parts of the formula (i.e., preCondtions, LHS and RHS) in Step3 and 4 with the help of RCM-to-MTL mapping rules in Table 4. Finally, we generate the entire formula based on the bound sides either "LHS ⟶ RHS" or "RHS" as in Step5.

5 EVALUATION

5.1 Dataset Description

We evaluate the coverage of our proposed RCM on 162 requirement sentences. These requirements were extracted from existing case studies in the literature and grouped into three sub datasets as follows: (1) expressiveness dataset (81 requirements): these are requirements collected from papers that introduced different requirement templates and formats in different domains and 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), (2) formalisation dataset (28 requirements): these are requirements extracted from papers that introduced requirement formalisation techniques including (Ghosh et al., 2016; Yan et al., 2015) with total of 28 requirements and (3) online sources (43 requirements): these are requirements extracted from an online available critical-system requirements including (Houdek, 2013). These requirements are available online in [1].

Figure 5 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 existed in a 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.

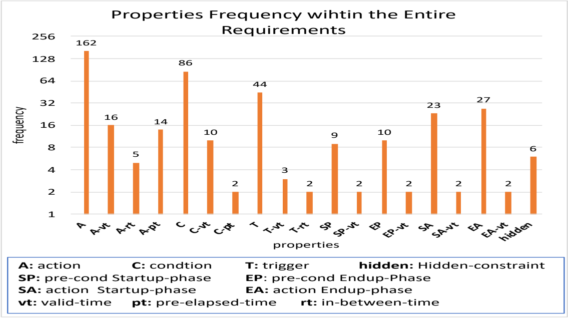


Figure 5: Properties Frequency wihtin the Entire Requirements.

Figure 6 shows the relative complexity of the 162 requirements. We grouped the requirements based on the count of their existing properties (i.e., number of properties per requirement increases ↑, its complexity increases ↑). The following examples show two requirements with one and six properties respectively, where each property is separately underlined: (1) "the monitor mode shall be initialized to INIT", and (2) "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 Figure 6, each group represents the count of properties regardless of the type of the property - i.e., 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 6 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 9%, 49% and 22%, only consist of one, two and three properties respectively. On the other hand, 20% of the requirements sentences consist of more than three properties.

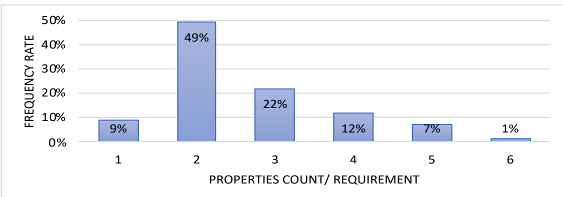


Figure 6: Frequency rate of Requirements per Properties Count.

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

5.2 Evaluation Experiments

Experiment1. RCM Expressiveness: We evaluated our proposed RCM reference model's ability to capture and represent the requirements in our test dataset compared to 15 exiting approaches in Table.3. To do this, we manually labelled all the requirements in the dataset against the 19 requirement properties we identified in section 4. After that, we wrote a script to check each requirement (identified properties) against all existing approaches to assess if the approach provides a boilerplate or a template that supports representing the requirement or not. The results are available online [2]. Figure 7 summarises the results of our analysis as percentage of the test requirements that each approach supports.

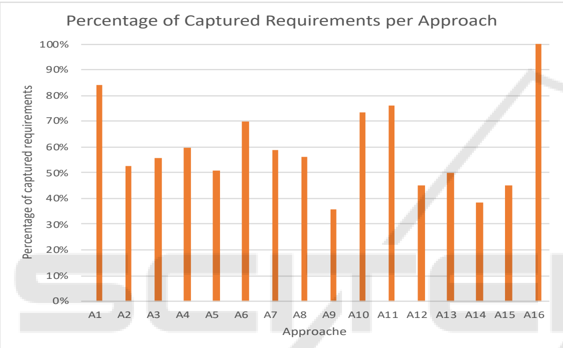


Figure 7: Percentage of Captured Requirements per Approach (RCM represented by A16 and the other represent by codes proposed in subsection4.2).

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 Pre-conditional scope (SP), or (2) restrictions on the included properties in a requirement format e.g., A2:EARS does not support the existence of the trigger (core-segment) and a ReqScope (core-segments) using the same format. In addition, ≈4% of the test requirements were not covered by any of these approaches combined. An example 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), StatrtUp-phase Pre-conditional 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 3.

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

Existing approaches require extension in two cases: (1) considering new requirement properties, and (2) considering new formats i.e, defining a set of properties that can exist together in one format regulated by customized grammatical rules. In contrast, since RCM covers all properties of the other approaches and more and puts no constraints on properties used in requirement, it is powerful enough to represent all requirements that can be represented by all the other approaches. It can also be used in other scenarios not currently supported by any of the 15 approaches, due to the fact that it does not enforce any restriction on the input requirement formats.

RCM encounters two main limitations: (1) it is designed for behavioral requirements of critical systems, and (2) it requires complex NL-extraction techniques i.e., the current NL-extraction processes primitive requirements expressed in one sentence.

Experiment2: RCM to Formal Notations: We applied our RCM-to-MTL and RCM-to-CTL transformation rules to the dataset of the 162 requirements. In this experiment, we used our NLP-approach to extract RCM from the 162 requirements(out of scope of this paper). We then manually reviewed all the extracted RCM models, fixed all the broken RCM extractions manually. Once we had the full list of 162 RCM models, we applied the automatic RCM-to-Formal transformation as outlined in Sec.4.3.3. The full list of RCMs representation 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 linear. For example, the requirement "the cognitive threshold of a human observer shall be set to a deviation that is less than 5. (Houdek, 2013)" 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

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

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

could be "AG(($\exists$ deviation$<$5) $\implies$ (the cognitive threshold of a human observer = deviation))" in CTL.

Similarly, CTL could represent requirements with hidden constraints correctly, but it provides partial solutions for requirements with time notation e.g., validation-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] $\implies$ [auto control mode.crrStatus = terminated])", but a correct formula could be "G([air_ok signal = low] $\implies$ [Ft=3(auto control mode.crrStatus = terminated)]" in MTL notation).

6 SUMMARY

We introduced a new requirements capturing model - RCM - for representing safety-critical system requirements. RCM defines a wide range of key requirement elements and attributes 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 to 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 others due to the flexibility in including/excluding its properties conforming the input requirement. In addition, we provided a suite of RCM-to-MTL transformation rules and presented the corresponding automatically generated MTL representation of the evaluation dataset. For our future work, we are developing an automated requirements extraction technique to populate RCM from textual requirements in addition to requirements quality checking and visualising tool of the RCM model.

ACKNOWLEDGEMENTS

Grundy is supported by Australian Research Council Laureate Fellowship FL190100035.

REFERENCES

Alur, R. and Henzinger, T. A. (1993). Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77.

Ambriola, V. and Gervasi, V. (1997). Processing natural language requirements. In *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pages 36–45. IEEE.

Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F., and Gnaga, R. (2013). 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*, pages 599–602. ACM.

Arora, C., Sabetzadeh, M., Briand, L. C., and Zimmer, F. (2014). Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns. In *Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on*, pages 1–8. IEEE.

Berger, P., Nellen, J., Katoen, J.-P., Abraham, E., Waez, M. T. B., and Rambow, T. (2019). Multiple analyses, requirements once: simplifying testing & verification in automotive model-based development. *arXiv preprint arXiv:1906.07083*.

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. (2019). 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.

Clarke, E. M. and Emerson, E. A. (2008). Design and synthesis of synchronization skeletons using branching time temporal logic. In *25 Years of Model Checking*, pages 196–215. Springer.

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

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420.

Erdem, E. and Yeniterzi, R. (2009). Transforming controlled natural language biomedical queries into answer set programs. In *Proceedings of the BioNLP 2009 Workshop*, pages 117–124.

Esser, M. and Struss, P. (2007). Obtaining models for test generation from natural-language-like functional specifications. In *International workshop on principles of diagnosis*, pages 75–82.

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.

Haider, A. (2015). A survey of model checking tools using ltl or ctl as temporal logic and generating counterexamples.

Holt, A. and Klein, E. (1999). 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*, pages 451–456. Association for Computational Linguistics.

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

Jeannet, B. and Gaucher, F. (2016). Debugging embedded systems requirements with stimulus: an automotive case-study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS)*.

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

Kittredge, R. I. (2003). Sublanguages and controlled languages. In *The Oxford Handbook of Computational Linguistics 2nd edition*. Oxford University Press.

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.

Konur, S. (2013). A survey on temporal logics for specifying and verifying real-time systems. *Frontiers of Computer Science*, 7(3):370–403.

Kuhn, T. (2014). A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170.

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.

Macias, B. and Pulman, S. G. (1995). A method for controlling the production of specifications in natural language. *The Computer Journal*, 38(4):310–318.

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.

Michael, J. B., Ong, V. L., and Rowe, N. C. (2001). Natural-language processing support for developing policy-governed software systems. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, pages 263–274. IEEE.

Nelken, R. and Francez, N. (1996). Automatic translation of natural language system specifications into temporal logic. In *International Conference on Computer Aided Verification*, pages 360–371. Springer.

Pease, A. and Li, J. (2010). Controlled english to logic translation. In *Theory and Applications of Ontology: Computer Applications*, pages 245–258. Springer.

Qiao, Y., Zhong, K., Wang, H., and Li, X. (2007). Developing event-condition-action rules in real-time active database. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 511–516.

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

Rolland, C. and Proix, C. (1992). A natural language approach for requirements engineering. In *International Conference on Advanced Information Systems Engineering*, pages 257–277. Springer.

Rupp, C. (2009). *Requirements-Engineering und-Management: professionelle, iterative Anforderungsanalyse für die Praxis*. Hanser Verlag.

Schwitter, R. (2002). English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, pages 228–232. IEEE.

Scott, W., Cook, S. C., et al. (2004). *A context-free requirements grammar to facilitate automatic assessment*. PhD thesis, UniSA.

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

Sturla, G. (2017). *A two-phased approach for natural language parsing into formal logic*. PhD thesis, Massachusetts Institute of Technology.

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.

Van Kleek, M., Moore, B., Karger, D. R., André, P., and Schraefel, M. (2010). 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*, pages 951–960.

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.