Component-based Capturing Model for Real-time Requirements

No Author Given

No Institute Given

Abstract. The use of embedded systems increasingly pervasive in critical systems with time-dependent functionality, vehicles and medical equipment, etc. As such systems have become dramatically larger in size and complexity it is increasingly important to formally verify these systems using automated tools. Most existing formalisation techniques require system engineers to (re)write their requirements using a set of predefined templates in order to utilise their structure (and assumed semantics) to generate the corresponding formal representation. However, this still requires understanding and memorising requirement templates, in addition to the inability of most templates to capture diverse requirements. In this paper, we propose generic segment-based Requirement Capturing Model (RCM) as a reference model corresponds to one requirement and mappable to temporal logic(s) to support automatic transformation into formal notations. RCM focuses on the main segments of a requirement sentence instead of treating the entire sentence as a block. We describe the design of RCM and how it supports the mapping of common formalisms (different versions of temporal logic). The evaluation shows that RCM breakdowns cover wider range of requirements formats than existing predefined solutions. In addition, it shows the viability of constructing RCM form NL-requirement.

Keywords: requirement representation · requirement modeling · requirement engineering.

1 Introduction

The complexity of many critical systems is rapidly increasing as a result of the incorporation of cutting edge technology in various industrial fields, such as vehicles, robotics, internet-of-things, smart infrastructure, smart homes and smart cities-based systems. Encountering errors during the operation of such systems can lead to severe injuries. Prevention of such outcomes mandates stringent monitoring of the quality of the system during the software and system development life-cycle. This can be supported by not only ensuring proper verification and validation at the end of the software development life cycle but also during each individual phase [40].

Critical systems frequently encompass time-dependent embedded systems that have special characteristics (e.g., (1)integrated system, (2) performing tasks within specified time frame, if the brake system of a car exceeds the time limit, it may cause accidents. (3) reacting to external stimuli and reacting accordingly like GPS, etc). Embedded systems must typically achieve a high level of robustness and reliability [22]. Therefore, formal methods for rigorously verifying the behavior of such systems before committing to code is critically important.

The use of formal methods mandates system requirements to be expressed in formal notations as a prerequisite. The majority of critical system requirements are still predominantly written in informal notations or natural languages (NL) which are inherently ambiguous [27,40] and have incomplete syntax an semantic. To automate the formalization process, several bodies of work within the literature focused on proposing pre-defined requirement templates, patterns [18], boilerplates [30], and structured control English [36], to capture system requirements while eliminating the ambiguities. Such templates have a complete syntax that ensures the feasibility of automating transformation of the requirements into formal notations using a suite of manually crafted, template-specific transformation rules. However, Some of the predefined are domain dependent that is may be hard to generalize [38] or can only capture limited subsets of requirements structures [36]. The technique of putting constraints on the formatting of the entire requirement sentence is vulnerable to extension to maintain coverage. This extension mostly cause editing in the corresponding customized formalization technique. In addition, most existing parsing algorithms for transforming system requirements to formal notations are customized for these notations. Thus, a need to transform the same requirements into different formal notations mandates significant rework of the parsing algorithms.

Complementary to this research direction, we target a new requirements formalisation paradigm aiming to increase flexibility and coverage. This is achieved by introducing a well-defined intermediate representation **a reference model** that defines the key requirement elements that could exist in an input requirement regardless of the format or structure of these elements. Once this reference model is well-established, we can then develop techniques to automatically extract these requirement elements from the input requirement. Finally, we can develop a suite of transformation rules - similar to these developed by existing techniques - to transform system requirements represented in this reference model into formal notations.

In this paper, we focus on the first step in this paradigm. We introduce RCM, Requirement Capturing Model, which acts as an intermediate representation between informal and formal notations. We developed the RCM based on reviewing most of the existing templates, patterns, CNL, etc in the literature.

The rest of this paper is organized as follows: section 2 provides a motivating example and outlines the main challenges. In section 3, we present an overview of our proposed RCM model. Section 4, presents related work. Finally, we conclude the paper and discuss future work in section 5.

2 Motivation

Is it possible to bridge the gap between natural language requirements and formal specifications, achieving a promising level of domain independence and generating different formal notation. The main challenges and key insights of the RCM are outlined in Table.1 .

Table 1. Key Inovations in RCM

	Challenges	Key Insights
1	Having a flexible approach to represent most textual requirements	Proposing a model based on standalone components instead of the structure of the entire sentence as a block (i.e, addressing what are the exist information not how the information expressed).
2	Having a generic approach mappable to formal notation	all standalone components and sub-components are eventually expressed/mapped to unique and precise structures mappable to formal notation.
3	Integration with various temporal logic notations	Propose a rich model encapsulates common formal semantic with the semiformal one.
4	Avoid loss of information due to natural language	Propose structure cope with NL-characteristics (e.g., a requirement may be expressed in multiple sentences).

3 Requirements Capturing Model

Ideally we need a way to take diverse requirements of such systems and represent them in a unified form. This allows checking the completeness, consistency and correctness of the requirements sourced from various representations (requirements written by different engineers mostly have different formats). Similarly, we want to be able to translate this unified requirements representation into several different formal notations, each with its own strength in terms of support for automated verification of the model. This allows us to formally verify, at an early stage of development, diverse real-time system requirements properties including temporal, event-based, and logical aspects.

3.1 The RCM Approach

To address these issues that Jen and her system engineering colleagues encoutered in our motivating examples, we have developed a novel *Requirement*

Capturing Model (RCM). RCM is a more comprehensive approach to representing NL-based requirement for real-time critical systems. RCM supports translating these common representations into a variety of formal requirement modelling notations for specialised automated requirements verification.

Within the frameworks of existing solutions a format is defined based on the structure of the entire requirement sentence. Thus, these approaches are sensitive to restructure-based paraphrasing. Conversely, our RCM overcomes this constraint by holding the exiting key elements of the requirement independent on their order and occurrence within the source natural language requirement sentence. Another advantage is that RCM supports storing a detailed level of information (at a formal level) enabling a direct transformation into different formal notations (e.g. temporal logic versions in our case) for further formal verification. RCM can also be used to provide a refined and unified version of the requirements in the form of constrained natural language CNL (textual writing is the most popular form of requirements representation among stakeholders [21]), for validation purposes.

We developed RCM by first analysing a large number of NL-based critical system requirements [6, 8, 9, 17, 26–28, 30, 36, 37, 43, 44] and along with a range of target formal requirements modelling languages [11,18,22,29,30,34–36,40,41,43, 45]. Then, we identified through an iterative process key common requirements modelling elements that would allow us to (i) capture a wide range of NL-based automotive system requirements in this common model; and (ii) translate this model into multiple widely used formal notations (temporal logic versions in our case).

3.2 The RCM Domain Model

In a given system, we define system requirements as a set of requirements R. Each requirement R_i, may have one or more primitive requirements PR where $\{R_i = < PR_n >$ and n>0$\}$. Each PR_j represents only one requirement sentence, and may include condition(s), trigger(s), action(s) and operational scope(s) – in the NL-requirement. RCM captures primitive requirements ,each corresponding to one sentence, of the same requirement separately and stores them as one unit. The detailed meta-model structure of the proposed RCM to represent such requirement R_i is shown in Fig.1.

3.3 RCM Description

Before starting the description of the RCM, we define a crafted REQ example shown in Fig. 2 and utilize it throughout this section. REQ combines variations of common critical system requirements properties. It builds upon the requirements examples presented in section2 where the used variables x, y, z, m, and threshold_1 represent the general case of specific values.

The core ,main building block, of the RCM model is the Primitive requirement PR_j. A PR_j corresponds to one discrete sentence of the source natural

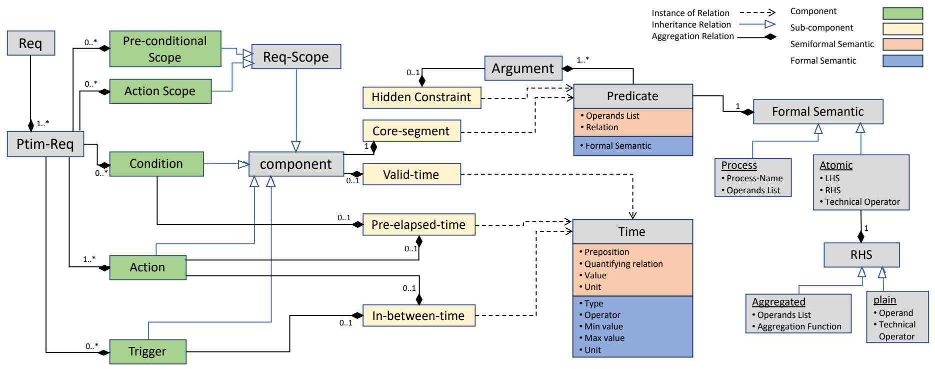

Fig. 1. RCM meta-model (simplified)

REQ: After an error occurs, when the system halts, if x is greater than threshold_1, an emergency signal is sent every 2 milliseconds. After an error occurs, when the system halts, if x is below the above threshold and (y is on and z is on), M should be set to 0 after at most 1 seconds.

Fig. 2. REQ: example presents multi-sentence requirement

language requirement. For our REQ example, it has two primitive requirements as it contains two sentences as indicated in Fig.3.

➢ **Id** = "1_req"
➢ **Primative-Req**

[0]	After an error occurs, when the system halts, if x is greater than threshold_1, an emergency signal is sent every 2 milliseconds.
[1]	After an error occurs, when the system halts, if x is below the above threshold and (y is on and z is on), M should be set to 0 after at most 1 second.

Fig. 3. Representing REQ in RCM

A PR_j consists of five components:

- **Trigger:** is an event that initiates action(s) (e.g., "when the system halts" in Fig.5). This component type is ubiquitous throughout the requirements of most critical systems.
- **Condition:** is a constraint that should be satisfied to allow a specific system action(s) happen (*e.g.*, "if X exceeds threshold_1" in Fig.5). 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.

- **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 0" in Fig.5). 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* [**?**]).
- **Req-scope:** determines what context under which (i) "condition(s) and trigger(s)" should be valid – called a pre-conditional scope; and (ii) "action(s)" should occur – called an action scope. The scope may define the starting boundary or the ending one (e.g., "after an error occurs", "before the system turns to Shutdown_Mode" in Fig.5).
  Fig. 4. presents the possible cases for starting/ending a context (*e.g.,* None, after operational constraint, until operational constraint is true or only before operational constraint becomes true). These are the main variations, 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 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.

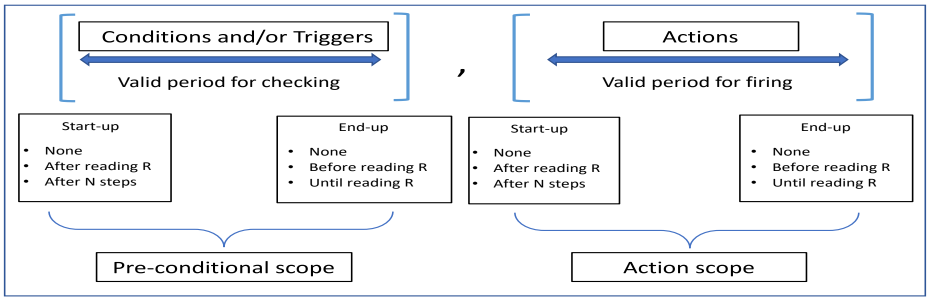

Fig. 4. Requirement Scope

By projecting the provided example in Fig. 5 to the requirement scope parts Fig. 4, we can find that the requirement has pre-conditional scope with only Start-Up, which is "After an error occurs". This can be mapped to "After reading R".

Each of these components optionally exists in a primitive requirement except action(s). Fig.5 shows the components of the primitive requirement PR[1] in Fig.3.

A component is further broken into sub-components. There are five distinct sub-components types:

- **Core-segment**; is mandatory to all components types. The core-segment express the main portion of the component including: the operands, the

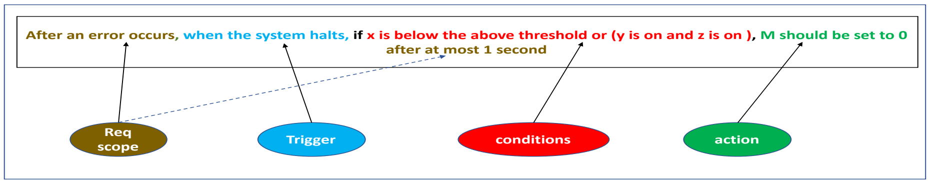
After an error occurs, when the system halts, if x is below the above threshold or (y is on and z is on), M should be set to 0 after at most 1 second
Req scope
Trigger
conditions
action

Fig. 5. Components of Primitive-Req[1] presented in Fig. 3

operator and negation flag/property (e.g., in "if X exceeds threshold_1" the "X" and "threshold_1" are the operands and "exceeds" is the operator in the semiformal semantic and "¿" is the operator in the formal semantic). It could be mapped later to a proposition in temporal logic as indicated in Fig.1.

- **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 seconds" the action is hold for 1 second length of time [**?**]). It is eligible to all components types but it is optional as indicated in Fig.1.
- **Pre-elapsed-time**; is the consumed time length from an offset point ,before an action to occur of a condition to be checked (e.g., "After 1 second" in Fig.**??**). This type of sub-component is eligible to action and condition component types only as indicated in Fig.1.
- **In-between-time**; express the length of time between two consecutive events to occur in the repetition case (e.g., "every 2 milliseconds" in Fig.**??**). Such sub-component type is eligible to action and trigger components as indicated in Fig.1.
- **Hidden constraint**; against to the above types that represent a a defined role in RE, hidden constraint it does not provide a new RE role but it is introduced to cope with the nature of natural language by allowing an explicit constraint to be defined for 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" [**?**], the **that is activated** is a constraint defined for the operand *the high beam headlight*). To store this information without loss, RCM stores the hidden constraint inside the relevant operand object as indicated in Fig.1. This structure is intrinsic for allowing 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*".

As a requirement may have nested relations among triggers, conditions or actions, these need to be captured properly to maintain the important relations among them. To satisfy this criterion, we found that the tree structure is the most suitable representation, where the identified predicates are represented as leaf nodes and the relations among them (logical-relations and ordering-relations) are represented as interior nodes. Fig. 6 shows the representation of the multiple conditions exist in REQ example Fig. 5. RCM not only supports logical relation

(AND, OR) among multiple triggers, conditions, or actions but also supports ordering relations (”is preceded by”, ”is succeeded by”) (*e.g.,* if S holds and is succeeded by Y, P holds [22]).

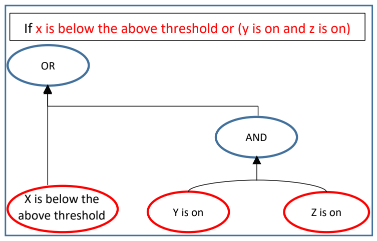
If x is below the above threshold or (y is on and z is on)
OR
AND
X is below the above threshold
Y is on
Z is on

Fig. 6. Capturing relations for nested conditions using tree

3.4 RCM and Temporal Logic

RCM components and sub-components are expressed by predicates or time structures as indicated in Fig.1. These structures are eventually mapped to proposition and time notations in the corresponding temporal logic formula in order to model a given requirement. A 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)). Within which, each proposition letter is expressed by a true/false statement. The booleans connecting propositions can be obtained from each tree-representation of each component type discussed before. However, the temporal modalities can be identified based on the component type.

In this sub-section, we provide a mapping into one example, Temporal Logic. We chose MTL [2,24] as a target notation, as it is widely used by researchers [23]. RCM can be also mapped into more abstract temporal logic with either linear (*e.g.,* LTL [42]) or branching (*e.g.,* CTL [7]) structures. It worth noting that, LTL and CTL are the most widely used by checking models as indicated [10,14]).

Metric temporal logic (MTL) [2, 24] is a real-time extension of linear-time temporal logic (LTL) [42]. MTL has time constraints in addition to temporal operator. MTL consists of propositional variables, logical operator (*e.g.,* $\neg$, $\vee$, $\wedge$ and $\implies$), temporal operators (*e.g.,*until U_I) where I is an interval of time. In addition, MTL has a timed-version of always, eventually operators. One the other hand, MTL doesn't handlle point Intervals (*e.g.,* $[t_1, t_1]$ as indicated in [23]). MTL assumes the existence of external and discrete clock updates constantly (fictitious-clock model as discussed in [1]). In Table 2, we summarise the meta-models mapping between RCM and MTL.

Table 2. RCM mapping to MTL

RCM				**MTL Mapping**
		Metric	**Versions**	
Temporal Modality	Req-scope	StartUP	After<Predicate> After S	$G(S \implies F(P))$
		EndUP	Before<Predicate> Before S	$F(S) \implies (F(P{\vee}S)US)$
			Until<Predicate> Until S	$F(P)US$
		StartUP and EndUp	After<Predicate> & Before<Predicate> Between Q and S	$G((Q \wedge \neg S \wedge F(S)) \implies F(P \vee S)US))$
			After<Predicate> & Until<Predicate> After Q Until S	$G((Q \wedge \neg S) \implies F(P)US)$
	Condition		If <Predicate> If S	$G(S \implies P)$
	Trigger		When<Event> When S	$G(S \implies P)$
Time notation	Condition/ Trigger /Action	Pre-elapsed-time	After<Fixed-time>	$F_{t=c}(P)$
			within<Min-time-length>	$F_{t\leq c}(P)$
			within<Max-time-length>	$F_{t\geq c}(P)$
	Condition/ Trigger /Action	Validation-time	for<Fixed-time>	$G_{t=c}(P)$
			for<Min-time-length>	$G_{t\leq c}(P)$
			for<Max-time-length>	$G_{t\geq c}(P)$
	Action/ Trigger	In-between-time	every<Fixed-time>	$G(F_{t=c}(P))$
			every<Min-time-length>	$G(F_{t\leq c}(P))$
			every<Max-time-length>	$G(F_{t\geq c}(P))$

3.5 Higher Level of Information (Formal Semantic)

The basic units of the RCM are predicate and time structures that are mapable to temporal logic. Temporal logic has various versions each with slight difference/extension. In order to support transformation into multiple versions of temporal logic with minimal adjustment in the parsing technique. RCM encapsulate formal semantic with the semiformal semantic within predicate and time structure as indicated in Fig.1. 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)").
– Atomic 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").
– Atomic 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, 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 Related Work

Since the ambiguity of natural language is a very common research and practice problem, researchers have tried to find alternative solutions to avoid this problem and increase the viability of automating formal methods. Many approaches [3, 12, 16, 32, 34] use requirements in the form of constrained natural languages (CNL) as input while transforming them into different formats of formal notations (*e.g.,* Linear temporal logic (LTL), SAL, temporal logic, Prolog language, etc.). Further processing is applied for validation and verification. On the other hand, other researchers focused only on proposing a viable solution to this problem.

CNL is a restricted form of NL especially created for writing technical documents as defined in [20]. CNL typically has a defined sub-set of NL grammar, lexicon and/or sentence structure [25]. Different forms of CNL are also provided as a reliable solution for requirements representation. In [36], the authors proposed Attempto Controlled English (ACE) which defines a fixed list of verbs, nouns and adjectives for the requirement set. ACE also applies restrictions on the structure of the sentence. In addition, the paper provides a tool for converting ACE into Prolog. However, it always assumes a standard interpretation for any ambiguity and the user either accepts it or rephrases the requirement.

Similarly, in [39] the paper presents Context Free Grammar (CFG) grammar for writing a requirement. However, the formats of the requirement components are limited. Additionally, restrictions are added on some words. Constraints on words are also used in the proposed Structured English in [22]. Structured English supports a variety of requirement representations especially in real-time which is in contrast to ACE and CFG.

Boilerplates are also presented in the RE field as a viable solution. They provide a fixed syntax and lexical words with replaceable attributes. Boilerplates are more limited than CNL and require adaptation to different domains. In [38], a very constrained RUP's boilerplate is provided which can handle a very limited range of requirements. While in [30], the presented Easy Approach to Requirements Syntax (EARS) boilerplates are less restricted and can support a wider range of requirements. For validating the conformity of the written requirement and the boilerplate, authors in [4, 5] provide checking techniques .

The use of patterns is a more flexible solution. However, many patterns are required for requirements coverage as each pattern is designed to only support the structure of a group of requirements. Since a new structure is added, a new pattern should be created to cope with. This leads to a continuous increase in the patterns size. In [43] a universal pattern is presented to support many requirements formats (trigger, then action). Moreover, many specific kernel patterns formats are presented in [18] to represent the core of the requirements.

These alternatives of defined formats have additional downsides. (1) The user needs guidance on how to phrase requirements in terms of CNL. (2) Reduces the power of expression, writing speed. (3) Might be so restricted that it becomes irritating to use. (3) Might have difficulty of obtaining compliance from users.

Several approaches are also provided for representing natural language requirements in graphical notations like [13, 15, 19, 31, 33]. However, we are interested and focused more in the textual-based representation. This is because, textual based representation is easier to be processed and converted to any other semi-formal notations (*e.g.,* graphical representation) or formal notation (*e.g.,* temporal logic).

All of these approaches focus on different perspectives of the requirements representation problem. Although they shows a significant success within the scope each created for, none of them can be considered as generic or comprehensive. However, approaches presented in [18, 22] cover a large portion of many safety-critical systems requirements' properties.

Complementary to these approaches, RCM covers the principle component of any requirement. It defines the key requirement components (trigger, action, condition.etc), and leave out their extraction to extraction algorithms. RCM does not enforce users with a structure but it allows any order of requirements properties following English grammar. RCM structure capture/keep a deep level of information that allow direct mapping to temporal logic. However, RCM requires complex extraction technique to be constructed from NL-requirements.

5 Conclusion and Future Work

We presented a new intermediate requirements representation model - RCM - that defines key requirement elements and attributes that could exist in an input requirement. We evaluated our model. We are developing an automated requirements extraction technique to populate the RCM from textual requirements. We are also working on a suite of transformation rules that can be used to convert RCM-based requirements into formal notations.

References

1. Alur, R.: Techniques for automatic verification of real-time systems. Ph.D. thesis, stanford university (1991)
2. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. Information and Computation **104**(1), 35–77 (1993)
3. Ambriola, V., Gervasi, V.: Processing natural language requirements. In: Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference. pp. 36–45. IEEE (1997)
4. 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. pp. 599–602. ACM (2013)
5. Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F.: Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns. In: Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on. pp. 1–8. IEEE (2014)
6. Bitsch, F.: Safety patterns—the key to formal specification of safety requirements. In: International Conference on Computer Safety, Reliability, and Security. pp. 176–189. Springer (2001)
7. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: 25 Years of Model Checking, pp. 196–215. Springer (2008)
8. Dick, J., Hull, E., Jackson, K.: Requirements engineering. Springer (2017)
9. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: Spear v2. 0: Formalized past ltl specification and analysis of requirements. In: NASA Formal Methods Symposium. pp. 420–426. Springer (2017)
10. Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M.: Comparison of model checking tools for information systems. In: International Conference on Formal Engineering Methods. pp. 581–596. Springer (2010)
11. Fu, R., Bao, X., Zhao, T.: Generic safety requirements description templates for the embedded software. In: 2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN). pp. 1477–1481. IEEE (2017)
12. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: Arsenal: automatic requirements specification extraction from natural language. In: NASA Formal Methods Symposium. pp. 41–46. Springer (June 2016)
13. Grundy, J., Hosking, J., Huh, J., Li, K.N.L.: Marama: an eclipse meta-toolset for generating multi-view environments. In: Proceedings of the 30th international conference on Software engineering. pp. 819–822. ACM (2008)

14. Haider, A.: A survey of model checking tools using ltl or ctl as temporal logic and generating counterexamples (12 2015). https://doi.org/10.13140/RG.2.1.3629.1925
15. Harmain, H., Gaizauskas, R.: Cm-builder: A natural language-based case tool for object-oriented analysis. Automated Software Engineering **10**(2), 157–181 (2003)
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. pp. 451–456. Association for Computational Linguistics (1999)
17. Jeannet, B., Gaucher, F.: Debugging embedded systems requirements with stimulus: an automotive case-study (2016)
18. Justice, B.: Natural language specifications for safety-critical systems. Master's thesis, Carl von Ossietzky Universität (2013)
19. Kamalrudin, M., Grundy, J., Hosking, J.: Tool support for essential use cases to better capture software requirements. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 255–264. ACM (2010)
20. Kittredge, R.I.: Sublanguages and controlled languages. In: The Oxford Handbook of Computational Linguistics 2nd edition (2003)
21. Kocerka, J., Krześlak, M., Gałuszka, A.: Analysing quality of textual requirements using natural language processing: A literature review. In: 2018 23rd International Conference on Methods & Models in Automation & Robotics (MMAR). pp. 876–880. IEEE (2018)
22. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering. pp. 372–381. ACM (2005)
23. Konur, S.: A survey on temporal logics for specifying and verifying real-time systems. Frontiers of Computer Science **7**(3), 370–403 (2013)
24. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-time systems **2**(4), 255–299 (1990)
25. Kuhn, T.: A survey and classification of controlled natural languages. Computational Linguistics **40**(1), 121–170 (2014)
26. Lúcio, L., Rahman, S., bin Abid, S., Mavin, A.: Ears-ctrl: Generating controllers for dummies. In: MODELS (Satellite Events). pp. 566–570 (2017)
27. Lúcio, L., Rahman, S., Cheng, C.H., Mavin, A.: Just formal enough? automated analysis of ears requirements. In: NASA Formal Methods Symposium. pp. 427–434. Springer (2017)
28. Macias, B., Pulman, S.G.: A method for controlling the production of specifications in natural language. The Computer Journal **38**(4), 310–318 (1995)
29. Marko, N., Leitner, A., Herbst, B., Wallner, A.: Combining xtext and oslc for integrated model-based requirements engineering. In: 2015 41st Euromicro Conference on Software Engineering and Advanced Applications. pp. 143–150. IEEE (2015)
30. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (ears). In: Requirements Engineering Conference, 2009. RE'09. 17th IEEE International. pp. 317–322. IEEE (Aug 2009)
31. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(1), 2–57 (2002)
32. Michael, J.B., Ong, V.L., Rowe, N.C.: 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. pp. 263–274. IEEE (2001)

33. Moreno, A.M.: Object-oriented analysis from textual specifications. In: Ninth International Conference on Software Engineering and Knowledge Engineering, Madrid, Spain (June 1997) (1997)
34. Nelken, R., Francez, N.: Automatic translation of natural language system specifications into temporal logic. In: International Conference on Computer Aided Verification. pp. 360–371. Springer (1996)
35. Pohl, K., Rupp, C.: Requirements Engineering Fundamentals. Rocky Nook (2011)
36. R. S. Fuchs, N.E.: Attempto controlled english (ace). In: CLAW 96, First International Workshop on Controlled Language Applications
37. Rolland, C., Proix, C.: A natural language approach for requirements engineering. In: International Conference on Advanced Information Systems Engineering. pp. 257–277. Springer (1992)
38. Rupp, C.: Requirements-Engineering und-Management: professionelle, iterative Anforderungsanalyse für die Praxis. Hanser Verlag (2009)
39. Scott, W., Cook, S.C., et al.: A context-free requirements grammar to facilitate automatic assessment. Ph.D. thesis, UniSA (2004)
40. Sládeková, V.: Methods used for requirements engineering. Master's thesis, Univerzity Komenského (2007)
41. Sturla, G.: A two-phased approach for natural language parsing into formal logic. Ph.D. thesis, Massachusetts Institute of Technology (2017)
42. Szałas, A.: Temporal logic of programs: a standard approach. In: Time and logic. pp. 1–50. UCL Press Ltd. (1995)
43. Teige, T., Bienmüller, T., Holberg, H.J.: Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. pp. 6–9. MBMV (2016)
44. Thyssen, J., Hummel, B.: Behavioral specification of reactive systems using stream-based i/o tables. Software & Systems Modeling **12**(2), 265–283 (2013)
45. Yan, R., Cheng, C.H., Chai, Y.: Formal consistency checking over specifications in natural languages. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1677–1682. IEEE (2015)