CORG: A Component-Oriented Synthetic Textual Requirements Generator*

Aya Zaki-Ismail[1][0000−0002−1580−6619] ✉, Mohamed Osama[1][0000−0002−6940−0833], Mohamed Abdelrazek[1][0000−0003−3812−9785], John Grundy[2][0000−0003−4928−7076], and Amani Ibrahim[1][0000−0001−8747−1419]

[1] Information Technology Institute, Deakin University, 3125 Burwood Hwy, VIC, Australia {amohamedzakiism, mdarweish, mohamed.abdelrazek, amani.ibrahim}@deakin.edu.au

[2] Information Technology Institute, Monash University, 3800 Wellington Rd, VIC, Australia John.Grundy@monash.edu

Abstract. The majority of requirements formalisation techniques operate on textual requirements as input. To establish and verify the reliability and coverage of such techniques, a large set of textual requirements with diverse structures and formats is required. However, such techniques are typically evaluated on only a few manually curated requirements that do not provide enough coverage of the targeted structures. Motivated by this problem, we introduce a Component-oriented synthetic textual requirements generator (CORG) that can generate large numbers of synthesised diverse-structure textual requirements, along with key components breakdowns. CORG utilises a controlled random-selection (CRS) strategy throughout the backtracking-based generation. We evaluate the coverage, diversity, performance and correctness of CORG. The evaluation results show that CORG can generate comprehensive diverse-structure combinations in reasonable time without being affected by the size of the produced requirements.

Keywords: Requirements engineering · Text generation

1 Introduction

Several requirements formalisation and extraction techniques [9, 26, 27] rely on textual requirements. The reliability of such techniques is critical, as they represent the foundation for the remaining requirements engineering and analysis tasks (e.g. 3C quality issues detection [26]). However, the evaluation of such techniques is typically limited to a curated set of few requirements [9, 27]. The main drawbacks of this evaluation approach are:-

– Structure-biased requirements: In this case, few of the targeted structures are covered and essential ones required for an exhaustive evaluation may be missed [9]

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

– Small number of requirements: The number of requirements used in evaluation is, in many cases, relatively small. This is because real-world requirements are typically confidential and not published making them hard to obtain, and manually developed ones require a considerable amount of time and effort to develop.

An example of such limitations is present in the evaluation of the requirements formalisation approach described in [9]. This approach accepts textual requirements incorporating events, conditions and actions. However, the evaluation was performed on requirements with event-action and condition-action formats as indicated in Fig.1: part *I*, in addition to some manually adjusted requirements from the (If A Then B) format to the (B If A) format as in Fig.1: part *II*. However, requirements holding the popular event-condition-action behavioural requirements format [24] are completely missed (i.e., not used in any order) as shown in Fig.1: part *III*.

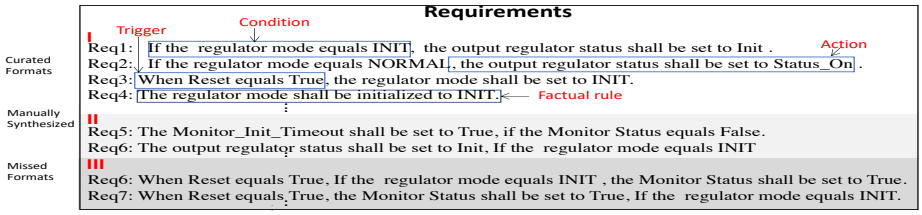

Fig. 1: Curated, synthesised and missed requirements samples by Gosh et al., [9]

In this paper, we tackle the problem of the utilisation of incomplete test data sets for the evaluation of formalisation techniques, by proposing a component-oriented synthetic textual requirements generator "CORG". It can be used to automatically generate comprehensive combinations of structurally-diverse synthesised textual requirements. In line with achieving such goal, all the generated requirements are expected to be equally useful for this problem without a need for a human analyst to assess their semantics. This is because formalisation techniques are insensitive to the semantics of the input requirements (i.e., their underlying analysis depends mainly on the requirements' grammatical syntax to be transformed into a corresponding formal notation [9, 26]). In addition, what enables a reliable evaluation of such techniques, is the co-existence of diverse structures within a combinatorially complete set of requirements.

We also propose the output layout of the synthesised requirements and the corresponding formal grammar. CORG adheres to this grammar during the generation process to produce requirements in the target output layout. The grammar and layout are based on a Requirement Capturing Model (RCM) [28]. RCM is a reference model that incorporates the key properties (i.e., the key components and sub-components) that may exist in a system requirement sentence. It extends popular requirements expression formats (e.g., EARS [17], etc).

We evaluated the capabilities of CORG in generating: (1) a complete set of the possible combinations of requirement's properties, (2) diverse structures, (3) correct requirements, and (4) realistic requirements. In addition to evaluating the generation time.

2 Background

RCM [28] is a semi-formal representation model that aggregates the behavioural NL-requirements components defined in literature and provides their respective sub-components and arguments breakdown. It supports a wide range of requirements because the model adapts to any permutation of its components.

RCM supports four requirements component types (i.e, Fig.1 shows samples):

- Action: expresses the tasks performed by the system .
- Trigger: represents events that implicitly fire actions within the system cycle.
- Condition: stands for specific constraints that should be satisfied and explicitly checked by the system for an action to occur.
- Conditional scope: represents the governing conditions required for either checking the preconditions (triggers and conditions) called "preconditional scope" or performing the action(s) "called action scope".

Sub-component types associated with the above components:

- valid-time: the period of time for a component to be valid (e.g., the inhibitor shall transition to [True] for at most 1 seconds).
- pre-elapsed-time: the consumed time –from a reference point– before a component starts (e.g., within 2 seconds the IDC transitions to [True]).
- in-between-time: the time between two consecutive repetitions of an event (e.g., the signal turns to [true] every 2 seconds).
- hidden-constraint: a constraint held for only one argument (i.e., system entity) within the component (e.g., **the entry** whose index exceeds 2 shall be incremented, the underlined text is held for the bold text).

Table 1 shows the possible sub-components of each component type.

Table 1: Sub-components association with each component type

Components / Sub-Components	PreConditional-Scope	Action-Scope	Condition	Trigger	Action
Hidden Constraint	✓	✓	✓	✓	✓
Valid-time	✓	✓	✓	✓	✓
Pre-elapsed-time			✓		✓
In-between-time				✓	✓

3 CORG Formal Grammar

We designed the output layout (targeting RCM) for the generated requirements, components, sub-components, and arguments breakdowns as follows:-

Listing 1.1: Generated Requirements Output Layout

```
req(reqText, CompList).
CompList::= [comp(SubCompList, compText),...]
SubCompList::= [subComp(Type, BreakDowns, subCompText), ...]
Type::= trig|cond|precondScope|actScope|act|hidden|v-time|pre-time|in-time
BreakDowns::=  TimeInfo|RelClause|SubClause|Clause
TimeInfo::=  [prePosition, quanitfication, value, unit, Nil]
RelClause::= [relNoun, relPronoun, subj, verb, [complement1, complement2,...] ]
SubClause::= [Nil, head, subj, verb, [complement1,complement2,...]]
Clause::= [Nil, Nil, subj, verb, [complement1,complement2,...]]
```

"name()" indicates composite entity, "[]" is list representation, and "Nil" means an empty item in the list. "req()" is a composite entity representing a requirement sentence, "reqText" is the text of the generated sentence, and "CompList" is a list of component entities each represented in "comp()". "compText" holds component text, and "SubCompList" is a list of sub-components entities each represented in "subComp()". "Type" is the (sub)component type, "subCompText" is the sub-Component text, and "BreakDowns" is a TimeInfo, RelClause, SubClause, or Clause according to the sub-component's type (all of them are lists of 5 items storing the sub-component arguments, where each item has a specific role based on its position in the list). "TimeInfo" represents breakdowns of time-related RCM subcomponents (i.e., valid-time, pre-elapsed-time and in-between-time). "RelClause", "SubClause", and "Clause" represent the breakdowns of the RCM hidden constraint sub-component, condition/trigger/condionalScope, and action component respectively.

Then, we developed a formal grammar to govern the generation process in line with the output layout. We only support present, future, imperative tenses and active/passive voices with correct syntax according to the English grammar. We define the formal grammar of the supported structures as follows:-

Listing 1.2: CORG Formal Grammar

```
<Sentence> ::= <Subclause>*.<Clause>.<Subclause>*
<Subclause> ::= Subordinator.<clause>
<Clause> ::= [<Subject>].[<RelClause>].<Predicate>.<TimeInfo>*
<RelClause> ::= HiddenConstHead.[Property].<Predicate>
<Subject> ::= <NounPh>
<NounPh> ::= Noun.<Modifier>*
<Modifier> ::= Preposition.<NounPh>
<Predicate> ::= [Modality].<MainVerb>.<Complement>+
<MainVerb> ::= Verb |(be).Verb.(ed)
<Complement> ::= [Preposition].(Noun|SystemValue)
Modality ::= "shall"|"will"|...
Subordinator ::= ConditionHead|TriggerHead|ScopeHead
ConditionHead ::= "if"|"provided that"|...
TriggerHead ::= "when"|"once"|"whenever"|...
ScopeHead ::= "after"|"before"|"until"|"while"|...
HiddenConstHead ::= "whose"|"that"
<TimeInfo> ::= TimePreposition. [QuantifyingRel]. Value. Unit
TimePreposition ::= Valid-Time-Prep|Pre-elapsed-Time-Prep|In-Time-Prep
Valid-Time-Prep ::= "for"|"up to"|...
Pre-elapsed-Time-Prep ::= "within"|"in"|"after"|...
In-Between-Time-Prep ::= "every"|...
QuantifyingRel ::= "less than"|"less than or equal"|"at most"|...
Value ::= Number
Unit ::= "seconds"|"minuets"|"milliseconds"|...
```

where, "*" indicates the presence of zero or more items, "+ means one or more, "." means the composition of different items, "..." means other words in the input dictionary, "< >" means non-terminal, and "[ ]" means optional item. Nouns, Verbs, Properties, SystemValues and Prepositions are not further decomposed (terminals) and are fetched from the input dictionary. The proposed grammar allows temporal operators through the element subordinator. A requirement sentence consists of at least one clause. A clause consists of at least a main predicate expressing the core meaning of the sentence, and optionally a subordinator -conditional or temporal conditional head– can be attached to extend the meaning as a subordinating clause.

4 CORG

CORG takes as input: (1) a dictionary of the domain lexical words and verb frames, and (2) size of requirements to be generated. It then utilises backtracking (a well-known approach previously adopted in textual generation [20]) through the built-in backward chaining inference engine of the Prolog programming language (a descriptive logic programming language consisting of a set of definite clauses (facts and rules) correlated to artificial intelligence and computational linguistics [22]).

Backtracking is typically a depth first search (DFS) mechanism, in which, an arbitrary decision is made at each choice-point. When a dead-end is reached, the inference engine backtracks to the last decision-point that can have a different path, makes a different choice, then proceeds from there. It can iterate over all the possible arrangements of a search space and provide all combinations and permutations [11]. However, it exploits all the possible permutations of the combination at hand before transitioning to the next combination. CORG performs a controlled random selection CRS (i.e., based on Prolog equi-distributed randoms [16]) at specific choice-points. This mitigates the exploitation pattern of DFS, and ensures combinations comprehensiveness and diversity maintenance even at small generation sizes while preventing structurally broken combination(s) (e.g., a requirement without an action is in-correct).

For each element in the (sub-)components set, CRS assigns a 0 or 1 random number to decide its inclusion/exclusion in the current combination. The combination is then rearranged to allow permutations and maintain generation diversity. The underlying equi-distribution allows CRS to produce a different combination at each call (i.e., ensures the coverage of the possible combinations whenever the generation size exceeds the combinations count). The generation then goes through Java APIs, SimpleNLG [8], and Stanford-NLP [15] as in Fig.2.

Text generation typically consists of five main tasks: content determination, textual structuring, sentence aggregation, lexicalisation, realization [7]. These tasks are applied differently according to the nature of the addressed problem. The underlying text generation approach adopted by CORG has been widely used in addressing a similar problem (i.e. sentence generation for quality testing and reliability evaluation) where several attempts have been carried out to

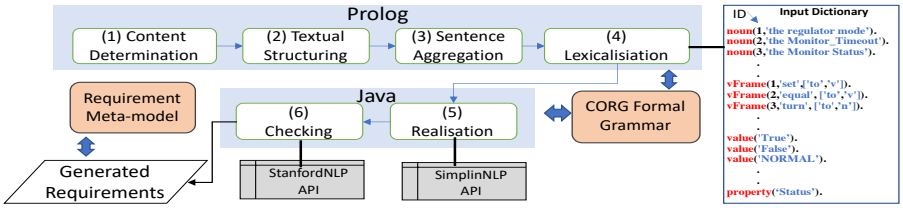

Fig. 2: CORG Framework

generate textual sentences to test programming languages compilers [14, 23, 29], and regular expressions [25, 30]. We also add a checking task to ensure fault free generation. The first four tasks are implemented with Prolog and the remaining two are implemented with Java as in Fig.2. CORG follows the proposed formal grammar and generates requirements in the proposed format. The six tasks are described in the following subsections and supported with a step by step generation example in Fig.3.

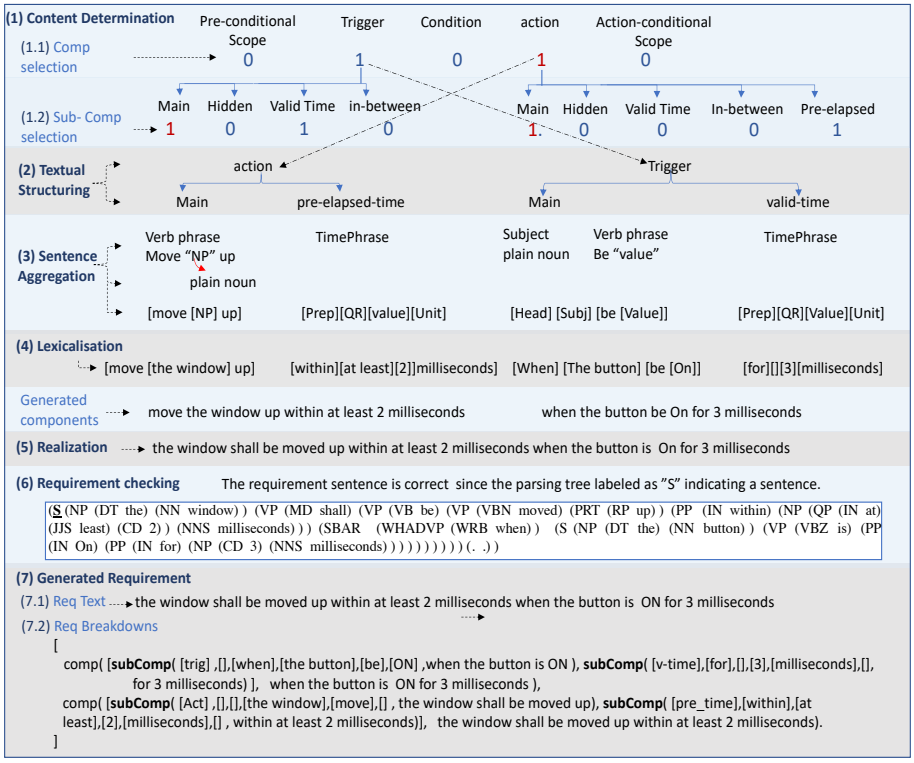

Fig. 3: Step by step CORG generation example.

4.1 Content determination

This task determines which components and sub-components will be included in the generated requirement R. It consists of two levels, the selection(s) made at each level lead(s) to different choices in the next level(s). At the **first level**, CRS is applied on the components set: Preconditional scope, condition, trigger, action scope except the mandatory action component as in Fig.3.1.1. In the **second level**, CRS is applied on the sub-comoponents set: different for each chosen component in the previous level as in Fig.3.1.2 (i.e., Table.1 for sub-components association). Eventually, the count and types of components and sub-components contributing in the generated requirement are identified. This task embodies the first four lines of the formal grammar and the CompList, and SubCompList in the output layout of the generated requirements.

Components and sub-components available for selection can be controlled before the generation process through CORG settings. This allows for adaptation into different domains and usage scenarios (*e.g.*, CIRCE [1] uses only event-condition-action ECA components).

4.2 Textual structuring

This task determines the order of the components within the sentence and the sub-components within the component. Alternative arrangement/permutation can be achieved through the random reordering technique. The approach takes one element (of a given combination) at a time and inserts it at a random free position in the new version as indicated in Algo.1.3. Similar to CRS, the random reordering is capable of providing a different permutation at each call (i.e., which ensures the comprehensiveness of the permutations for the same combination in large enough sizes and maintains diversity in small sizes). Fig.3.2 shows the reordered components after applying random reordering.

Listing 1.3: Random Reordering Algorithm

```
OutComps = φ
While(InComps ≠ φ){
    CrrComp = InComps.removeFirst()
    Len = OutComp.length()
    RandomIdx = getRandom(0, Len+1)
    OutComp.insertAt(RandomIdx, CrrComp)
}
```

4.3 Sentence Aggregation

This task selects the grammatical structures to apply on an individual component. By the end of this task, the outlined formal grammar clauses in Task1 shall be assigned a complete random grammatical structure as in Fig.3.3. A clause may include up to four different types of parts (i.e., Subject, RelClause, Predicate and TimeInfo as in Grammar 1.2). Each part has more than one valid structure. Table 2 lists the alternative structures of each part conforming to the proposed formal grammar, where NounSize and VFrameSize indicate the count of distinct nouns and verb-frames in the dictionary respectively. In addition, it highlights how these structures are used to select a new random structure.

Table 2: Alternative structure for clause's breakdowns Controlled in CORG

Type	Grammatical Rules Pseudo	Type	Grammatical Rules Pseudo
Subject	**if** nominalCount = 1 **then** *ID ← getRandom(1,NounSize)* *Noun ← getNominal(ID)* *return Noun* **else** *Prep ← getCompositionPrep()* *Noun ← Prep + getNominal(nominalCount -1)* *return Noun* **end if**	Predicate	Dictionary selection: the grammatical frames of each verb stored in the dictionary *ID ← getRandom(1, VFrameSize)* *VFrame ← selectFrame(ID)* *return VFrame*
TimeInfo	**if** Type = ValidTime **then** *Prep ← getRandVTimePrep()* **else if** Type = PreElapsed **then** *Prep ← getRandPreTimePrep()* **else if** Type = InTime **then** *Prep ← getRandInTimePrep()* **end if** *QR ← getRandQantRel()* *Val ← getRandValue()* *Unit ← getRandUnit()* *TI ← aggregat(Prep,QR,Val,Unit)* *return TI*	RelClause	**if** Type = OnProperty **then** *RelP ← getPropRelPronoun()* *Prop ← getRandProperty()* **else if** Type = OnNoun **then** *RelP ← getNounRelPronoun()* *Property ← φ* **end if** *VFrame ← getVerbFrame()* *RVF ← realizeFrame(VFrame)* *RC ← aggregat(RelP,Prop,RVF)* *return RC*

4.4 Lexicalisation

In this task, the chosen grammatical rules in the previous task are populated with randomly selected lexical words conforming to the grammatical roles as in Fig.3.4. These words are fetched from the dictionary. Table 3 lists the structures (each has a grammatical role) in the dictionary with descriptions. Each structure instance has an "Id" to allow random selection from the same structure type (i.e, a random number is generated to fetch the lexical word whose id equals the generated number and whose structure (grammatical role) is regulated by the syntactic rules as indicated in Table.2). By the end of this task, the sub-components' breakdowns in the output layout shall be complete.

Table 3: Dictionary Metamodel

Structure	Description	Example
noun(*ID*, *Noun*)	Nominal Noun	noun(1,'the car')
vFrame(*ID*,*Verb*,*ArgList*)	verb-frame representing the predicate structure. -*ArrgList: expresses the verb associated: arguments notations and prepositions in order within the frame *Arg-notations: ('v' → sys-value) and ('n' → noun)	vFrame(1, 'set', ['to','v'])
property(*ID*, *Property*)	Property for nominal nouns used in relative clauses	property(1, 'door')
sysValue(*ID*, *Value*)	domain values for nominal nouns or properties	sysValue(1,'[Locked]')

4.5 Realisation

The generated components for each requirement require tense adjustment as per the English grammar. This task considers adjusting the tense of each com-

ponent (each expressed by a clause). In this task, all components' types are assigned to the present tense except actions (future tense), as shown in Algo.1.4. For diversity, we put the action in the imperative form using active and passive voices randomly. Fig.3.5 shows an example of the components after tense adjustment. We adjust the tense using SimpleNLG. *First*, we prepare the subject, verb and complement of the current component. *Then* we feed them to the SimpleNLG realiser along side other grammatical flags (*e.g.*, tense(in both levels), voice, person). *Finally*, we readjust the affected parts in the requirement and their breakdowns.

Listing 1.4: Tense Adjustment

```
If(Comp.Type = "act")
     Voice = get random voice
     If(Voice = "Active"}
        adjustToImperative(Comp)
     Else
        adjustToPassiveFuture(Comp)
Else
    adjustToActivePresent(Comp)
```

4.6 Requirements Checking

In the generated requirement a "," is appended to all the components (except the last one). Depending on the generated components, random re-ordering may cause grammatical errors because of the incorrect punctuation. We use StanfordNLP to detect ungrammatical sentences by analysing the output of the parse tree. The parse tree represents the syntactic structure of the given string according to a specific context-free-grammar. A requirement sentence whose obtained parsing tree is labeled with an "s" (i.e., sentence) is correct [21]. Fig. 4 shows two parsing tree (a) and (b) of the same requirement (with different punctuation), tree in (b) is grammatically correct while the other one in (a) is incorrect. (a) can be corrected by removing incorrect comma(s) as in (b). In general it is not recommended to depend on the output of the parse tree alone in such checking. However, in our case it is effective because the realisation task ensures the correctness of each component. According to Fig.3.6, the generated requirement in the tracing example is correct.

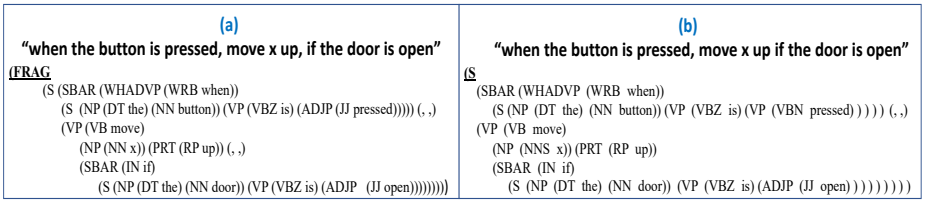

Fig. 4: Parsing trees of (the same requirement sentence with different punctuation)

5 Evaluation

In this section, we evaluate CORG [3] using assessment criteria (coverage and time performance metrics) that have proven effective in evaluating text generation in

[3]CORG Source Code: https://github.com/ABC-7/CORG/tree/main/CORG

several approaches [14,25,30]. We conducted five experiments to evaluate CORG on such metrics in addition to assessing diversity, correctness, and realisticness:

- Coverage: Does CORG provide comprehensive combinations of the components and sub-components according to RCM?
- Time Performance: How does the requirements size affect CORG's generation time?
- Diversity: Is CORG capable of providing all the possible arrangements of each combination?
- Correctness: Does CORG correctly generate requirements as expected?
- Realisticness: Can CORG generate semantically sound requirements

The dictionary[4] used in the generation contains 16 nominal nouns, 7 verb frames, 4 values and one property (i.e., obtained from requirements used in [9]).

5.1 Generation Coverage

In this experiment, we generated 500 unique requirements [4] since the possible combinations of components and sub-components are 448 (i.e., calculated using ${}^{n}C_{r}$). The basis of our evaluation is the correct combinations of the possible power set of components and sub-components that can constitute one requirement. We evaluate the coverage on two levels.

First level tests the coverage of the possible correct components combinations. Fig.5.(a) shows a Venn-diagram highlighting in dark the correct combinations of components among all the possible combinations (i.e., a requirement without action is incorrect). Fig. 5.(b) shows the percentage of each components combination within the generated requirements. It can be seen that, the generated requirements cover all the correct combinations and do not incorporate any incorrect combinations (e.g., a requirement with no action).

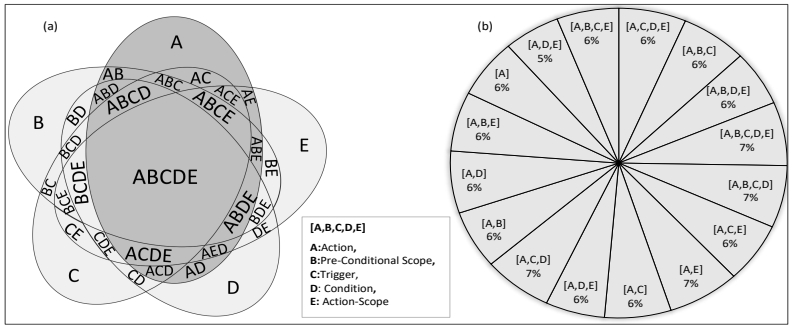

Fig. 5: (a) Venn-diagram for components combination (correct combinations highlighted in dark gray). (b) Components combinations percentages in generated requirements

[4] Dictionary and Generated-requirements:https://github.com/ABC-7/CORG/tree/main/Experiment

In the second level, we evaluate the coverage of the sub-components of each individual component. Fig. 6 shows the ability of CORG to cover the generation of the possible sub-components combinations (i.e., where, the action has $2^4 = 16$ possible combinations of its associated sub-components, condition and trigger have $2^3 = 8$, and the other components have $2^2 = 4$ combinations). Each column shows the percentages corresponding to each combination of the sub-components for the intended component within the entire requirements. It can be seen that, no combination is missed.

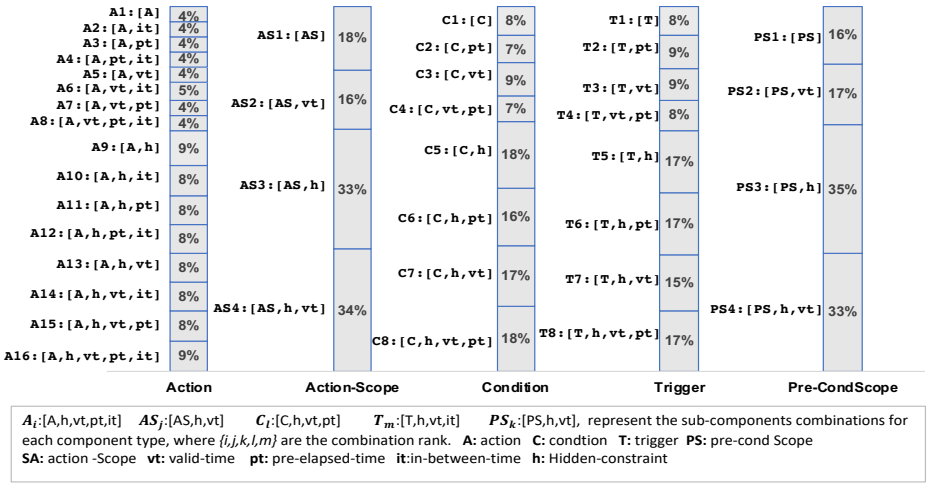

Fig. 6: The possible sub-components combinations coverage

5.2 Time Performance

We evaluated CORG generation time (on Prolog) for unique and redundant requirements on ten different data-set sizes. We measured the average time of five samples for each data-set size in both (unique and redundant data-sets). Fig. 7 shows that CORG generates up-to 10000 unique requirements in around seven seconds and around one second for the redundant requirements. Unique data-set generation requires more time since more time is required to eliminate and replace redundant generation attempts with unique ones. The unique generation time depends on the dictionary size (larger dictionary sizes guarantee less generation time especially for larger data-set sizes).

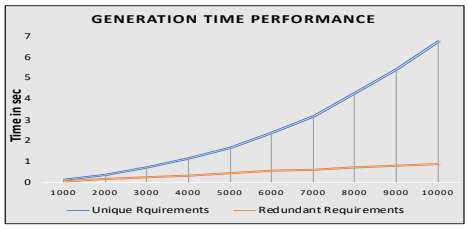

Fig. 7: CORG avg. Generation Time

5.3 Diversity Evaluation

The textual structuring step in section 4.2 is responsible for maintaining structure diversity within the generated requirements (i.e., by getting different permutation each time for the given combination). To assess diversity, we set an experiment to generate all requirements holding one combination, then assess diversity (i.e., permutations) within the generated requirement. The presumed combination is a requirement with condition, trigger and action components (i.e, any generated requirement would have the three components). This combination has six permutations: {(Cond,Act,Trig), (Cond,Trig,Act), (Trig,Act,Cond), (Trig,Cond,Act), (Act,Trig,Cond), (Act,Cond,Trig)}. Finally, we informed CORG to generate just 10 requirements with this setting. Fig. 8 shows that, the six arrangements are generated in the first seven requirements (i.e., the remaining requirements have repeated arrangements). It is worth noting that, this displayed output is before the tense adjustment step.

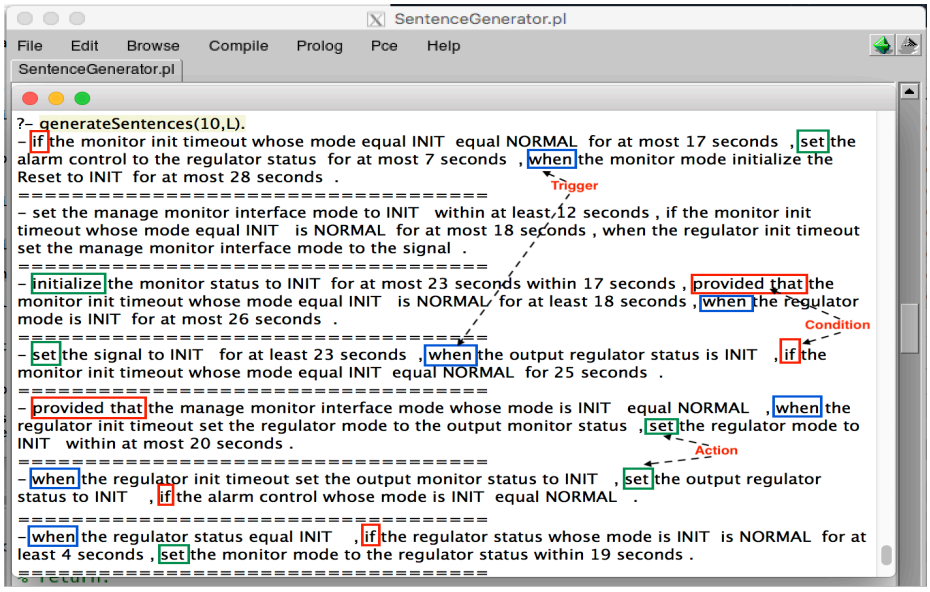

Fig. 8: 10 generated requirements for the combination (condition, trigger, action) highlighting the corresponding complete six permutations

5.4 Correctness Evaluation

We feed the 500 requirements generated in the coverage experiment to an automated NLP-based requirements extraction approach [28]. Then, we automatically compared the generated breakdowns (i.e, components, sub-components

and, arguments) to the extracted ones[5] through string matching. The experiment shows that all requirements are generated correctly as expected. The used extraction approach depends mainly on StanfordNLP typed dependency which has a percentage of error [28]. To provide the extraction with fully correct interpretations, the sub-components text of each generated sentence are addressed apart by StanfordNLP and their corresponding typed dependencies are aggregated. Finally, the aggregated typed dependencies are used in the extraction.

5.5 Realisticness Evaluation

We evaluated the ability of CORG to generate realistic requirements (similar to human-written ones). To achieve this, we fed the tool with the dictionary of a group of manually-specified requirements for a target system. Then, we used CORG to check if these requirements can be generated or not - reverse engineering utilising the inference engine of Prolog to check if a given output could be produced from the given input.

We conducted the experiment on a data-set of 19 requirements collected from the literature - used in [9]. We fed CORG with (a) the requirements dataset and (b) the system dictionary. CORG successfully constructed the breakdown of all input requirements[6]. This experiment shows that a subset of the generated requirements is both syntactically and semantically correct.

5.6 Strengths and Limitations

The key strengths of CORG are: (1) providing combinatorially complete coverage for components and sub-components, (2) allowing the customisation of components and sub-components contributing in the generation process, (3) ensuring structure-diversity in small and large data-set sizes, (4) associating the generated requirements with their breakdowns and (5) large number of requirements may be generated from a small dictionary with very few details (i.e., just lexical words and verb frames). It is also worth noting that CORG can be easily enhanced to ensure the generation of semantically reasonable requirements by adding association rules to the input dictionary to only allow certain lexical words and verb frames to be selected together.

The main limitation of CORG is that the generated requirements are not all semantically reasonable because the concrete system relations are not considered in the dictionary. However, this does not affect the effectiveness of the generated requirements in enabling a reliable evaluation of the formalisation approaches because such approaches are only affected by the syntax of the input.

[5]Extraction-Output:https://github.com/ABC-7/CORG/blob/main/Extractionlog.xml

[6]Input-requirements, Used-dictionary, and Resulting-breakdowns for realisticness experiment: https://github.com/ABC-7/CORG/tree/main/ValidationExperiement

6 Related Work

We cover the related work from three perspectives:

(a) sentence generation from lexical words: data driven textual generation is a widely used text generation approach (HALogen [13], Nitrogen [12] and FERGUS [2]. These approaches adopt a two-stage architecture. In the first stage, a forest of the possible expressions is constructed. In the second stage, the expressions are selected using a probabilistic model. In contrast, CORG applies CRS instead of the forest construction to save generation time. GENERATE [10] randomly generates sentences using a small set of English phrases, syntactic rules and transformation rules to form valid sentences. As input, it takes a dictionary containing both the nouns and verbs (semantically coded to assure that invalid sentences such as "The building smoked a cigar" will not be produced). The dictionary consists of twenty verbs and twenty nouns. For flexibility, CORG supports dictionaries with or without semantics.

(b) Requirements generation: most approaches generate textual requirements from software engineering models [19]. In [18], NL requirements are generated from UML class diagrams. This approach uses a rule set in conjunction with a linguistic ontology to express the components of the diagram. Alternatively, the approach presented in [4] relied on system domain-specific grammar to provide description and information regarding specific requirements technical terms. In [5] the Semantics of Business Vocabulary and Business Rules (SBVR) was used as an intermediate representation for transforming UML into constrained natural language. Similarly, CORG uses a defined grammar for the generated requirements controlled by a set of rules. However, the content of the generated requirement(s) is syntactically correct but not limited by a specific system since no relations are enforced in the dictionary. Other approaches generate creative requirements (i.e., more useful and novel requirements) for the sustainability of software systems. Bhowmik et al. [3] propose a framework to obtain creative requirements by making unfamiliar connections between familiar possibilities of requirements. In [6], a novel framework generates creative requirements utilising NLP and ML techniques for both novel and existing software. The framework reuses requirements from similar software in the application domain and leverages the concept of requirement boilerplate to generate candidate creative requirements. Such approaches include a manual checking process to discard useless outcome. Similarly, CORG eliminates useless requirements –syntax oriented– through an automatic checking process.

(c) Text generation for evaluation: several attempts [14, 23, 29] have been carried out to generate sentences to test the parsing of programming languages compilers. In addition, textual strings are generated in [25,30] to evaluate regular expressions. The main feature of these techniques is that the generated text must obey to the formal grammar of the compiler/regular expression. Similarly, requirements generated by CORG follow a formal grammar for describing systems behavior. We share with such approaches the goals and metrics of the generated text (i.e., combinations coverage for robust evaluation and generation performance).

7 Conclusion

In this paper, we introduced CORG; a synthetic requirements generator that can produce all the possible combinations and diverse structures with respect to the RCM set of key requirements properties. First, we defined a formal grammar for the generated requirements. Then, we employed the backtracking technique with a controlled random-selection to ensure combinatorial comprehensiveness and maintain diversity in small data-sets. Evaluation results show that CORG is able to generate comprehensive combinations with diverse structures regardless of the size of the generated requirements. In the future, we aim to investigate CORG capabilities in: (1) generating requirements in other languages, (2) filtering semantically unreasonable requirements utilising both human vetting and dictionary rules (which can be useful for generating creative requirements).

References

1. Ambriola, V., Gervasi, V.: On the systematic analysis of natural language requirements with c irce. Automated Software Engineering **13**(1), 107–167 (2006)
2. Bangalore, S., Rambow, O.: Exploiting a probabilistic hierarchical model for generation. In: Proceedings of the 18th conference on Computational linguistics-Volume 1. pp. 42–48. Association for Computational Linguistics (2000)
3. Bhowmik, T., Niu, N., Mahmoud, A., Savolainen, J.: Automated support for combinational creativity in requirements engineering. In: 2014 IEEE 22nd International Requirements Engineering Conference (RE). pp. 243–252. IEEE (2014)
4. Burden, H., Heldal, R.: Natural language generation from class diagrams. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation. pp. 1–8 (2011)
5. Cabot, J., Pau, R., Raventós, R.: From uml/ocl to sbvr specifications: A challenging transformation. Information systems **35**(4), 417–440 (2010)
6. Do, Q.A., Bhowmik, T., Bradshaw, G.L.: Capturing creative requirements via requirements reuse: A machine learning-based approach. Journal of Systems and Software **170**, 110730 (2020)
7. Gatt, A., Krahmer, E.: Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. Journal of Artificial Intelligence Research **61**, 65–170 (2018)
8. Gatt, A., Reiter, E.: Simplenlg: A realisation engine for practical applications. In: Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009). pp. 90–93 (2009)
9. Ghosh, S., Shankar, N., Lincoln, P., Elenius, D., Li, W., Steiener, W.: Automatic requirements specification extraction from natural language (arsenal). Tech. rep., SRI INTERNATIONAL MENLO PARK CA (2014)
10. Hackenberg, R.G.: Generate: A natural language sentence generator. CALICO Journal **2**(2), 5–8 (2013)
11. Khachiyan, L., Boros, E., Elbassioni, K., Gurvich, V.: A new algorithm for the hypergraph transversal problem. In: International Computing and Combinatorics Conference. pp. 767–776. Springer (2005)
12. Langkilde, I.: Forest-based statistical sentence generation. In: Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference. pp. 170–177. Association for Computational Linguistics (2000)

13. Langkilde-Geary, I.: An empirical verification of coverage and correctness for a general-purpose sentence generator. In: Proceedings of the international natural language generation conference. pp. 17–24 (2002)
14. Lutovac, M.M., Bojić, D.: Techniques for automated testing of lola industrial robot language parser. Telfor Journal **6**(1), 69–74 (2014)
15. Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J.R., Bethard, S., McClosky, D.: The stanford corenlp natural language processing toolkit. In: Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations. pp. 55–60 (2014)
16. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**(1), 3–30 (1998)
17. 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)
18. Meziane, F., Athanasakis, N., Ananiadou, S.: Generating natural language specifications from uml class diagrams. Requirements Engineering **13**(1), 1–18 (2008)
19. Nicolás, J., Toval, A.: On the generation of requirements specifications from software engineering models: A systematic literature review. Information and Software Technology **51**(9), 1291–1307 (2009)
20. O'Donnell, M.: Sentence Analysis and Generation–a Systemic Perspective. Ph.D. thesis, University of Sydney (1994)
21. Osama, M., Zaki-Ismail, A., Abdelrazek, M., Grundy, J., Ibrahim, A.: Score-based automatic detection and resolution of syntactic ambiguity in natural language requirements. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 651–661. IEEE (2020)
22. Pereira, F.C., Shieber, S.M.: Prolog and natural-language analysis. Microtome Publishing (2002)
23. Purdom, P.: A sentence generator for testing parsers. BIT Numerical Mathematics **12**(3), 366–375 (1972)
24. 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. pp. 511–516. ACM (2007)
25. Radanne, G., Thiemann, P.: Regenerate: a language generator for extended regular expressions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 202–214. ACM (2018)
26. 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)
27. 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 (2021)
28. 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 (2021)
29. Zelenov, S.V., Zelenova, S.A.: Generation of positive and negative tests for parsers. Programming and Computer Software **31**(6), 310–320 (2005)
30. Zheng, L., Ma, S., Wang, Y., Lin, G.: String generation for testing regular expressions. The Computer Journal (2019)