V. ANNEX

This section illustrates how this demo paper will be presented in the poster and demo paper session at the RE21 conference. We will demonstrate two parts: (1) input description language, and (2) user interface of the generation tool.

A. Input Description Language

We will illustrate the description layout of the single and multi level formats.

1) Single-level Description: In this level, the user can define the count of the requirements to be generated and can specify specific issue(s) for a specific requirement as well. For each requirement, the count and types of the contributing clauses can be controlled. In addition, within each clause, a suffix/prefix can be added at any place for the expression of the intended issue. We designed the input/output format of the requirement in this level as follows:

```
ReqList::= [inReq(id, ClausesList, CoordList,
    ↪ reqText), ...]
inReq::= a composite entity encapsulating a single
    ↪ requirement sentence
id::= a unique key assigned to each requirement
ClausesList::= [inComp(cId, type, [hPrefix, head,
    ↪ sPrefix, subj, vPrefix, verb, ComplementL],
    ↪ clauseText),...]
CoordList::= a list of 5 items. Each item represents
    ↪  the coordination relations among clauses
    ↪ with the same type --structured with braces.
reqText::= the text of the generated requirement
inComp::= a composite entity encapsulating one
    ↪ clause
cId::= a unique key assigned to each clause
type::= indicates the role of the clause in the
    ↪ requirement \{act|trig|cond|actScope|
    ↪ condScope\}
hPrefix, sPrefix, vPrefix ::= excess text attached
    ↪ to the clause grammatical breakdowns
head::= subordinating clause head
ComplementL::= [prep(preposition, modifier),...]
prep::= a composite entity encapsulating a single
    ↪ prepositional phrase
```

In the above format, each item in the requirements list to be generated represents a requirement sentence entity with: id, list of constituting clauses, list of five elements each representing the coordination relations within one of the supported component types (if any) (i.e., a specific type with no coordination relation is attached to an empty string), and the requirement text. The clauses list contains the clauses of the sentence each acting as one of the five requirement roles identified in the literature (i.e, Condition, Action, Trigger, Precondition-scope and Action-scope) [8]. A clause is represented by a single component entity consisting of: cId (a uinque key identifying each clause and representing the component in the coordinating relation(s) –if any), type (indicates the requirement component role within the clause), a list of lexical words where the position of the word in the list determines its linguistic role, and clauseText (the component text). For example, the component entity "comp('A1', 'Act', ['','','','X','shall be','set',[prep('to','true')]], 'X shall be set to true)" indicates one independent clause with cID "A1" having the role action in the requirement with the subject "X", the verb "set", and the complement "to true" in the passive voice. The complement list expresses the complements in the clause in the form of prepositional entities with two parameters: preposition and modifier. if the complement is noun-phrase, adjective, or adverb without a preceding preposition, the preposition parameter is bound to ". The user can provide details at different levels of abstraction as in the examples below:

- Control the count of clauses (e.g. four) in the second generated requirement.

```
[R1, inReq(ID, [[], C1, C2, C3, C4], CoordList,
↪ ReqText) |RemReqs]
```

- Have the first requirement with one clause including the suffix "as fast as possible"

```
    [inReq(ID,[[],[],[inComp(cID,'Act', ['',
↪ Head, '', Subj, '', Verb, [prep('','as fast
↪ as possible')],ClauseText)], CoordList,
↪ ReqText) |RemReqs]
```

2) Multi-Level Description: This level allows the users to include relations among multiple requirements sentences (i.e., multiple requirements sentences with a specific relation). For example, the deadlock metric happens when two requirements sentence are depending on each other (i.e., each one requires the other to occur first). Thus, we defined the multi-level format to provide control over the format of each sentence (including clauses and their breakdowns), and the relations among multiple sentences as follows:

```
ReqList= [inReq(id, clausesList, CoordList, reqText)
    ↪ , ...]
inReq::= a composite entity encapsulating one
    ↪ requirement sentence
id::= a unique key assigned to each requirement
CoordList::= list of 5 items. Each item represents
    ↪ coordinating relations among clauses with the
    ↪  same type --structured with braces.
reqText::= the text of the generated requirement
clausesList= [inComp(type, entity, value, verb,
    ↪ NegationFlag, cId),...]
inComp::= a composite entity encapsulating a single
    ↪ clause
type::= indicates the role of the clause in the
    ↪ requirement \{act|trig|cond|actScope|
    ↪ condScope\}
entity::= the system entity where the clause is
    ↪ defined
value::= the value of the system entity
cId::= a unique key assigned to each clause
```

The above input/output format is similar to the one defined for the single-level with the following differences: (1) there is no suffix nor prefix, (2) no complement list, and (3) each clause has an entity (the system component where it is defined) and a corresponding value. The user can also control the multiple usage of: entities, values, and their relations at both the components and sentences levels (e.g., E1 refers to the same entity in multiple components or sentences). This adds more flexibility in describing any indicator of a quality issue at multiple levels of details. For example, it supports the generation of two contradicting requirements like: R1= "if the IDC inhibitor equals True, the sailing termination equals True" and ", R2= "if the IDC inhibitor equals True, the sailing termination equals False" the actions in R1 and R2 have the same entity with different values). This level of details

in describing the requirement could be provided with the proposed format and achieved by our generator.

VI. DBRG User Interface

DBRG is available online [3]. Figure 3 presents the user interface of DBRG. First, the user selects the description level of interest and provides the count of scenarios to be generated following the descriptive format of the selected level. Each scenario represents a generation case of the input description. After processing, the generated scenarios are displayed. The figure shows a descriptive format in the multi-level for a deadlock issue between two requirements and the corresponding generated two scenarios. For single-level descriptions, the user is able to provide the requirements count included in each scenario, where the count is larger than or equal to the specified requirements in the input description. However, in multi-level descriptions, the requirements count is set to the number of requirements specified in the description to control their shared relation (i.e., generating other requirements in the same scenario may break the relation defined).

After the generation process, the generated requirements along with their breakdowns can be exported to a text file via the export button, where each scenario will be stored in a separate file. This ensures the correctness of each generated scenario because: (1) each scenario is generated separately according to the input description, (2) the predicates are distinct within the scope of each generated scenario, and (3) different scenarios may contain repeated clauses leading to unwanted relation(s). For example, in Fig.3, the first requirement in the second scenario has the same action as the first requirement in the first scenario. Table I provides a sample output of the generated requirements for both single and multi level descriptions.

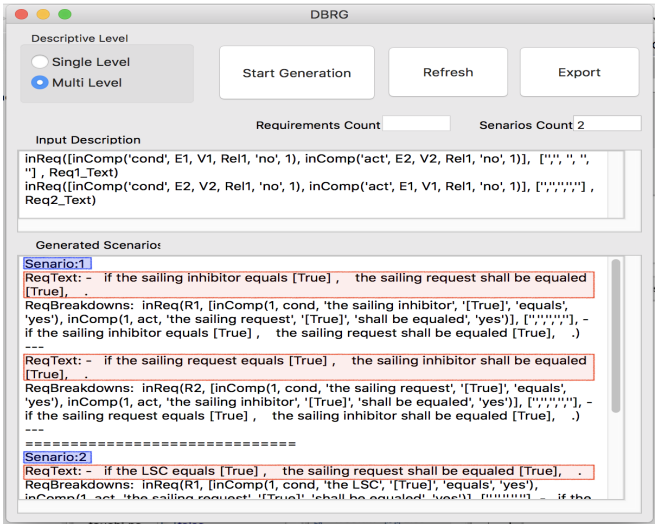

Fig. 3: DBRG Tool Main View
The beginning of each scenario and the generated requirements are highlighted in blue and red respectively.

[3]DBRG Tool and Demo: https://github.com/ABC-7/DBRG

TABLE I: Examples Input Description of Single-Level and Multi-Level Quality issues and the Corresponding bound variables Output

Quality Issue	Example of input format and Generator Output	
Informal Coreference	In	[inReq('R1', [[], []], CL, [inComp('A1', 'Act', ['', '', '','It', '', V1,[P1]], CompText)], []], ['','','','',''], Text)]

	Out	`CL = [inComp('C1', 'Cond', ['', if, '', 'the monitor status', '', is, [...]], "if the monitor status is True")],`
		`V1 = shall be set,`
		`P1 = prep(to, 'False'),`
		`CompText = "It shall be set to False "`
		`Text = "- , , if the monitor status is True , It shall be set to False , ."`
		`GenReqL = [inReq('R1', [[], []], [inComp('C1', 'Cond', ['', if, '', 'the monitor status', '', is, [prep('', 'True')]], "if the monitor status is True")], [inComp('A1', 'Act', ['', '', '','It', '', 'shall be set',[prep(to, 'False')]]], "It shall be set to False")], []], ['','','','','',''], "- , , if the monitor status is True , It shall be set to False , .")]`

Semiformal Deadlock	In	[inReq([inComp('cond', E1, V1, Rel1, 'no', 1), inComp('act', E2, V2, Rel1, 'no', 1)], ['','', '', '', ''] , Req1_Text), inReq([inComp('cond', E2, V2, Rel1, 'no', 1), inComp('act', E1, V1, Rel1, 'no', 1)], ['','','','',''] , Req2_Text)]

	Out	`E1 = 'the transmission error',`
		`V1 = V2, V2 = 'True',`
		`Rel1 = equals,`
		`E2 = 'the LSC',`
		`Req1_Text = "- , , if the transmission error equals True , the LSC equals True , .",`
		`Req2_Text = "- , , if the LSC equals True , the transmission error equals True , ."`
		`GenReqL = [inReq([inComp('cond', 'the transmission error', 'True', 'equals', 'no', 1), inComp('act', 'the LSC', 'True', 'equals', 'no', 1)], ['','','', '', '', ''] , "- , , if the transmission error equals True , the LSC equals True , ."), inReq([inComp('cond', 'the LSC', 'True', 'equals', 'no', 1), inComp('act', 'the transmission error', 'True', 'equals', 'no', 1)], ['','','','','',''] , "- , , if the LSC equals True , the transmission error equals True , .")]`