Knowledge Graph based Automated Generation of Test Cases in Software Engineering

Anmol Nayak
Robert Bosch Engineering and Business Solutions
Anmol.Nayak@in.bosch.com

Vaibhav Kesri
Robert Bosch Engineering and Business Solutions
Vaibhav.Kesari@in.bosch.com

Rahul Kumar Dubey
Robert Bosch Engineering and Business Solutions
RahulKumar.Dubey@in.bosch.com

ABSTRACT

Knowledge Graph (KG) is extremely efficient in storing and retrieving information from data that contains complex relationships between entities. Such a representation is relevant in software engineering projects, which contain large amounts of interdependencies between classes, modules, functions etc. In this paper, we propose a methodology to create a KG from software engineering documents that will be used for automated generation of test cases from natural (domain) language requirement statements. We propose a KG creation tool that includes a novel Constituency Parse Tree (CPT) based path finding algorithm for test intent extraction, Conditional Random field (CRF) based Named Entity Recognition (NER) model with automatic feature engineering and a Sentence vector embedding based signal extraction. This paper demonstrates the contributions on an automotive domain software project.

KEYWORDS

Knowledge Graph (KG), Named Entity Recognition (NER), Constituency Parse Tree (CPT), Requirement to Test Case generation

ACM Reference Format:
Anmol Nayak, Vaibhav Kesri, and Rahul Kumar Dubey. 2020. Knowledge Graph based Automated Generation of Test Cases in Software Engineering. In *7th ACM IKDD CoDS and 25th COMAD (CoDS COMAD 2020), January 5–7, 2020, Hyderabad, India.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3371158.3371202

1 INTRODUCTION

Software testing is an integral phase of the Software Development Life Cycle (SDLC). In commercial software projects, a Functional Requirement Document (FRD) is created in the initial phase of the project to encapsulate all the necessary specifications needed from a client. Software testing performed for these functional requirements is referred to as functional testing [10]. Once the system has been designed and implemented, it is subjected to a series of inputs given by the software tester. The behavior of the system is observed under various test conditions and validated with the software documents that describe the expected functionality of the system.

CoDS COMAD 2020, January 5–7, 2020, Hyderabad, India

ACM ISBN 978-1-4503-7738-6/20/01…$15.00
https://doi.org/10.1145/3371158.3371202

Knowledge Graph have gained widespread popularity in recent years with the advent of sophisticated Natural Language Processing (NLP) algorithms. KG consists of two fundamental components: Entities (represented as nodes in the KG) and Relationships (represented as edges connecting the nodes). The task of extracting entities from a corpus is known as Named Entity Recognition (NER). Some of the popular algorithms to perform NER are Conditional Random Fields (CRF) [7] and Bi-directional Long Short Term Memory (BiLSTM) neural networks [4]. While BiLSTM based NER models perform extremely well when there is a large amount of labelled corpus, their performance gets severely hampered in the case of sparse corpus. CRF based models perform well even in the case of sparse corpus but require efficient feature engineering to be performed in domain specific applications.

Relation Extraction (RE) algorithms are used to mine relationships between entities in the corpus. This is an important step in creating a KG as the quality of relations extracted will determine the overall quality of the KG and the ease with which information retrieval can be performed from the KG. Stanford OpenIE [1] and ClausIE [5] are widely used to perform RE in the construction of KG. Stanford OpenIE uses a pre-trained classifier to handle long sentences by breaking them into short and coherent clauses which are maximally shortened. ClausIE extracts relations by separating detection of information from the representations in terms of extractions. Since these algorithms have been designed from open source text, the quality of extracted triples is hampered when applied to software engineering domain corpus which contains lexica and sentence structure that is niche.

Some of the notable effort in KG development include the Google Knowledge Graph [14], Never Ending Language Learner (NELL) [3], Yet Another Great Ontology (YAGO) [16] and ConceptNet [15], which have captured several million entities and relations from large scale datasets like Wikipedia, WordNet etc. However, there has not been significant development of KG in the area of software engineering where the corpus are largely sparse and unstructured. An effort to create a KG for a Question-Answering (QA) system in software engineering was done [8], however it does not focus on leveraging the KG to automate the software testing process. There have been several work in generating test cases from state machine based models, however they are limited by the individual requirements as the corpus [6], [17]. This is a limitation as very often the requirement statements do not contain all the necessary conditions needed to test a system. Previous works have also focused on leveraging UML diagrams for test case generation, however all these approaches expect a UML diagram available as a prerequisite and do not automatically generate it from a corpus [12], [9], [13], [11]. An approach to use inference rules to generate test cases was proposed,

however these inference rules required intervention from domain teams and heuristics derived from a very small set of requirements [18].

The main contributions of this paper are as follows:

(1) A KG for automatically generating complete test cases that can handle missing information in the functional requirements.
(2) The KG creation tool builds a KG from corpus that is unstructured and sparse.
(3) An exhaustive list of test cases can be generated using the KG depending on the extent of testing to be done.

The remainder of this paper is organized as follows. In Section 2 we provide an overview of the software testing process. Section 3 discusses the proposed KG architecture describing the corpus and ontology. Section 4 describes the proposed KG creation tool and the algorithms in each phase. In Section 5 the results obtained by querying the KG are analyzed. Section 6 concludes the paper and proposes future work.

2 OVERVIEW

In the conventional process of software testing (Figure 1), a software tester manually creates test cases for each requirement statement by reading the software documentation which can span over several thousand pages in the case of complex systems. This is an intensively laborious process and requires a deep understanding of the documentation by the tester. Once the test cases are written by the software tester, a team reviews each of the test cases after which they are executed on a testing environment (e.g. LABCAR). A test report is generated after the test case execution finishes, with vital information such as number of test cases passed/failed, time to execute each test case etc. This process is repeated for every software project. The first step in generating test cases is to extract the test intent from the requirement statement. The test intent describes three fundamental components required to create a test case:

- Pre-conditions: These are all the conditions required to be satisfied before an Action or Post- condition of the requirement statement can be achieved.
- Actions: These are all the actions that are performed once the Pre-conditions have been satisfied.
- Post-conditions: These are all the conditions that are observed after the Pre-conditions and Actions have been satisfied.

For example consider the requirement statement: *If SignalA is on and ButtonA is pressed, then SignalB is on.* The test intent will be: Pre-conditions: *SignalA is on*, Actions: *ButtonA is pressed*, Post-conditions: *SignalB is on.*

3 KNOWLEDGE GRAPH ARCHITECTURE

The sub-components of Phase 1 and Phase 2 of the KG pipeline (Figure 2) are as follows:

3.1 Text Corpus

For creating the KG, we have used unstructured text corpus consisting of software documents, Past Requirement statements and Past

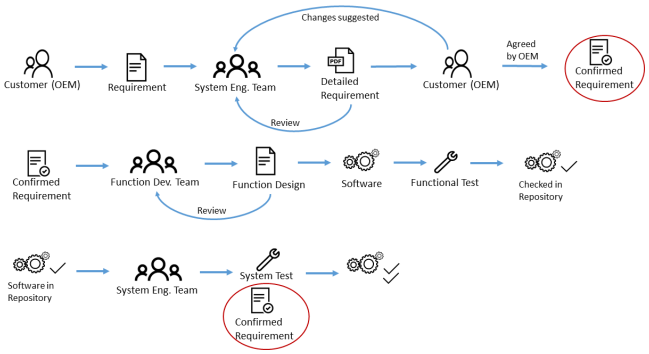

Figure 1: Flow of requirement

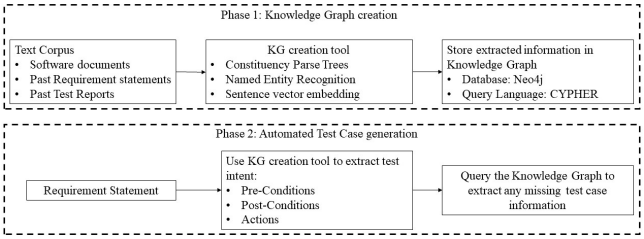

Figure 2: KG pipeline for automated generation of test cases

Test Reports from an automotive domain project. For confidentiality reasons, we have anonymized the dataset.

- Software documents: It describes the entire software system consisting of the different modules, functions, states, substates etc. It is usually created for every software project that is shipped and is used extensively in the software testing phase of the SDLC.
- Past Requirement statements: These are vital in updating the KG as a lot of times change requests to a software system that has already been designed appear in requirement statements.
- Past Test Reports: They contain necessary information such as initialization conditions which can be used in generating future test cases.

3.2 KG Ontology

A KG Ontology defines the class type of entities, their attributes and the type of relations between entities. Software testing involves writing test cases capturing several inter-dependencies and it is often needed to test a system at several levels of hierarchies. The depth of testing changes with the project specification. Taking the requirement statement example mentioned above, a tester may want to test all the possible ways of achieving *SignalB is on*. Hence, we have designed an ontology in a way to capture information to handle such exhaustive ways of generating test cases. We can traverse through the KG to any depth based on the extent of testing to be performed on the system.

In the KG Ontology for software testing is depicted in Figure 3a, every named entity will belong to one of the following class types:

- **Pre-condition only entity**: An entity found to occur only as a Pre-condition in our corpus.

- **Pre-and-Post condition entity**: An entity found to occur both as a Pre-condition and a Post-condition in our corpus.
- **Post-condition only entity**: An entity found to occur only as a Post-condition in our corpus.
- **Action only entity**: An entity found only to occur as an Action in our corpus.

The relationship between any of these entity type can be either of a **Pre-condition** or a **Post-condition**. The reason we have kept only two possible relation types is that in the scenario where there are no Action only entities in the statement, the Pre-conditions will directly connect to the Post-conditions. In the other scenario, the Pre-conditions will never connect to the Post-conditions directly but instead via the Action only entities.

Attributes describe additional information describing entities of a domain. These attributes can be used extensively in efficient KG querying. The attributes in this KG ontology are as follows:

- **Pairs**: This attribute is assigned only to Action only entity nodes. Since we can have different pairs of Pre-conditions and Post-conditions for the same set of Actions, this attribute will help disambiguate such cases. This attribute also keeps track of any other actions which need to performed in addition to this action to achieve the Pre-conditions and Post-conditions pair. For example, in the following test intent's both have the same Actions but different Pre-conditions and Post-conditions mapping:
  - Pre-conditions: *SignalA is on*; Actions: *ButtonA is pressed*; Post-conditions: *SignalB is on, SignalE is on.*
  - Pre-conditions: *SignalG is on*; Actions: *ButtonA is pressed*; Post-conditions: *SignalH is on.*

  The Pairs attribute in the *ButtonA is pressed* node will be: Pairs=Pre-conditions: *SignalA is on*|Post-conditions: *SignalB is on,SignalE is on* |Other-Actions:none||| Pre-conditions: *SignalG is on*| Post-conditions: *SignalH is on*|Other-Actions:none (Note: ||| separates each unique pair).
- **Individual-Pre-conditions**: This attribute is assigned only to Pre-and-Post condition entity nodes and Post-condition only entity nodes. In many cases where Actions are not needed to achieve Post-conditions, this attribute will store such Pre-conditions to Post-conditions mapping to disambiguate the grouping of Pre-conditions needed to achieve a Post-condition. This attribute will list all the combinations of Pre-conditions needed to achieve a certain Post-condition. For example, in the following test intent's both have the same Post-conditions but different groups of Pre-conditions:
  - Pre-conditions: *SignalB is on*; Actions: none; Post-conditions: *SignalC is on.*
  - Pre-conditions: *SignalD is on, SignalF is on*; Actions: none; Post-conditions: *SignalC is on.*

  The Individual-Pre-conditions attribute in the *SignalC is on* node will be: Individual-Pre-conditions=Pre-conditions: *SignalB is on*|Post-conditions: *SignalC is on* |||Pre-conditions: *SignalD is on, SignalF is on*|Post-conditions: *SignalC is on.*
- **Description**: This attribute is assigned to all nodes and it holds the complete description of the signal mentioned in the node. It can be used by the software tester to better understand the system and also for suggesting any corrections and modifications that should to be made in the KG.

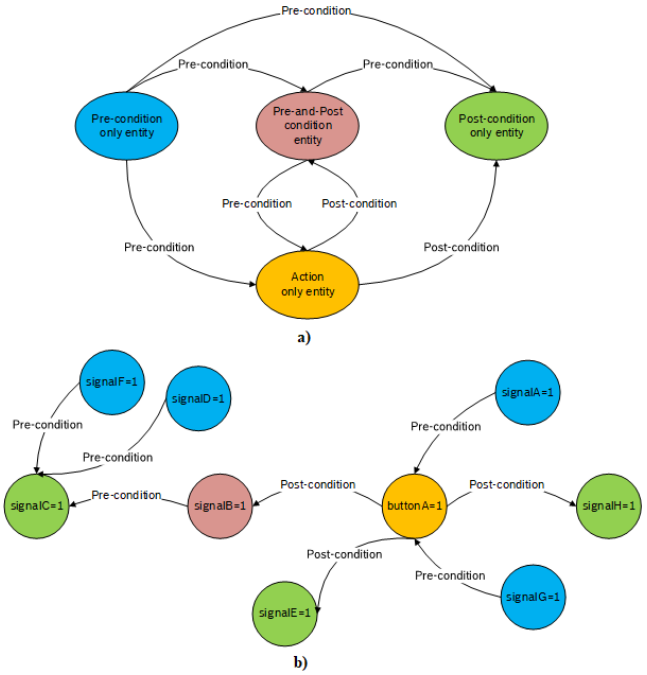

Figure 3: (a) KG Ontology for Software Testing (b) Sample KG for Software Testing in Neo4j Database

4 KNOWLEDGE GRAPH CREATION TOOL

Our KG creation tool as depicted in Figure 2, applies a series of NLP algorithms to mine information from the text corpus and populates the KG. The tool has several components that are also used in Phase 2 of the KG pipeline.

4.1 CPT based path finding algorithm

To build a KG with the software testing ontology, each sentence in the corpus was represented in the form of a test intent. To extract components of the test intent, a CPT was constructed using Stanford Constituency Parser [19]. A CPT was created as it helped us to isolate the dependent and independent conditions from a sentence using the constituent phrase types. We call the each dependent condition as *Post-conditions candidate string* and each independent condition as *Pre-conditions candidate string*. These are referred to as candidate strings as they undergo another step of filtering using NER to filter out the Actions in the test intent. To extract the Pre-conditions candidate string and Post-conditions candidate string we developed the following algorithms:

CPT path finder algorithm for finding Pre-conditions candidate strings:

(1) Apply Depth First Search (DFS) on the CPT to find a sub-tree with root node as $'SBAR'$ or $'PP'$. Let this sub-tree be T_n found at depth n of the CPT.
(2) Apply Breadth First Search (BFS) for T_n and check if there exists a pair of child subtrees $ST_{x,n+1}$ and $ST_{x+1,n+1}$, where x and $x+1$ are the index of child subtrees found at level $n+1$, such that root node of $ST_{x,n+1}$ is $'IN'$ or begins with $'WH'$ and root of $ST_{x+1,n+1}$ is $'S'$.

(3) Check if there exists more than one child subtree $ST_{y,n+2}$ of $ST_{x+1,n+1}$, where y is in range [0,number of subtrees of $ST_{x+1,n+1}$ at level $n+2$), such that the root node of $ST_{y,n+2}$ is $'S'$. For each such subtree $ST_{y,n+2}$ perform Step 4. If none exist, perform Step 4 for $ST_{x+1,n+1}$.
(4) Apply DFS and create a Pre-conditions candidate string appending each leaf node until a subtree is found that satisfies Step 1 and Step 2.
(5) Step 1 DFS continues.

CPT path finder algorithm for finding Post-conditions candidate strings:

(1) Apply BFS on the CPT to find pairs of sub-trees $T'_{x',n'}$ and $T'_{x'+1,n'}$, with $'P'$ in root node label where n' is the depth in the CPT and x' and $x'+1$ are the BFS index respectively.
(2) $T'_{x',n'}$ and $T'_{x'+1,n'}$ shouldnt be subtrees of any Pre-conditions candidate string tree $ST_{x+1,n+1}$. The leaves of $T'_{x',n'}$ and $T'_{x'+1,n'}$ are appended in sequence to create a Post-conditions candidate string. Each such pair of sub-trees creates a Post-conditions candidate string.
(3) If there exists an odd numbered sub-trees in Step 1, the leaves of this subtree creates the Post-conditions candidate string.

4.2 NER model

Named entities in software projects often follow certain domain specific semantics that should be captured in the features used for training a NER algorithm. Order of the CRF model is chosen based on the most frequently occurring n-gram (at word level) in the named entities. For example, if the entities are mostly tri-gram (3 words), then a 2^{nd} order CRF model will perform better as the context window increases in making a prediction. Automatic featuring engineering is done by extracting n-grams (at alphabet level) from named entities which occur in less than C named entity classes. These n-grams features are used in addition to the traditional features like Part Of Speech (POS) tags etc. For example, if all the entities belonging to the class *Action* begin with the n-gram phrase *act* then this n-gram feature will be useful in identifying *Action* entities. The value of C is decided based on the total number of entity classes. A small value of C is recommended (< 3), since the higher the value of C, the n-gram is a common occurrence across multiple entity classes and hence is not useful in distinguishing between entities of those classes.The larger the value of n, more will be the emphasis that will be given on the entire portion of the entity word. A standard 1^{st} order CRF with traditional features was used for benchmarking (without feature engineering). For the software testing KG, we trained a 2^{nd} order NER model on 7 classes: **Target module**, **Variable**,**Comparison**,**Action**,**Hardware**,**Value**,**Others**. The dataset consisted of 2000 tagged sentences from the software documentation. Table 1 shows the averaged results for a 5 fold validation.

For example, consider the statement: *Cruise control enters ModeA when ButtonA is pressed for more than 3s.* The named entities would be as follows- Target module:*Cruise control*, Variable: *enters ModeA*, Comparison: *more than*, Action: *pressed*, Hardware: *ButtonA*, Value: *3s*, Others: *is, when, for.*

4.3 Test intent in text form

After extracting all the Pre-conditions candidate string and Post-conditions candidate string, each candidate string is tagged with the NER Model to see if they contain both *Hardware* and *Action* entities. If so, the shortest substring consisting both the *Hardware* and *Action* entity classes is selected as the Action for the test intent. In this case, the remaining candidate string is checked to see if it contains atleast one named entity and if so, it is considered as the final Pre/Post-condition respectively for the test intent after stripping the words at either ends belonging to *Others* class. If no named entity was found, the remaining candidate string is discarded. When no *Hardware* or *Action* entity is found in the Pre/Post-conditions candidate string, the entire candidate string is treated as the final Pre/Post-condition respectively for the test intent. For example, consider the following two cases of candidate strings:

Case 1- Post-conditions candidate string contains both *Action* and *Hardware* entities: *SignalA is on if ButtonA is pressed.*

In this case the shortest substring containing both *Hardware* and *Action* entities is *ButtonA is pressed* hence this is the Action of the test intent. The remaining substring *SignalA is on if* contains a named entity *SignalA* belonging to *Target module* class and *on* belonging to *Variable* class. Hence after stripping *if* as it belongs to *Others* class, the Post-condition of the test intent will be *SignalA is on.*

Case 2- Post-conditions candidate string contains no *Action* and *Hardware* entities: *SignalA is on.* In this case there are no *Hardware* and *Action* entities found, hence the entire candidate string is treated as the Post-condition for the test intent.

4.4 Sentence vector embedding

As we have developed a software testing KG for an automotive domain corpus, it was required that every test case condition or action has to be associated to a signal for the testing environment to execute the test case. We parsed the text corpus to mine the signal descriptions and corresponding signals. These were stored in a dictionary as a key-value pair: *{SignalA is on:signalA=1, SignalB is on:signalB=1, SignalC is on:signalC=1, SignalD is on:signalD=1, SignalE is on:signalE=1,SignalF is on:signalF=1,SignalG is on:signalG=1, SignalH is on:signalH=1, ButtonA is pressed is on:buttonA=1}.* It is often the case that requirement statements do not mention conditions or actions in the exact terminology as mentioned in the software documents. For example, *SignalA is on* might be mentioned as *SignalA is switched on* or *SignalA is turned on* in the requirement statement. All the three instances must be mapped back to *signalA=1.* Hence, to tackle this problem we created 150 dimensional vector embedding's for each of the signal descriptions in our dictionary using FastText algorithm [2]. Each condition or action in the corpus/requirement will be converted to a vector embedding and then assigned a signal corresponding to that signal description in the dictionary whose vector embedding was most similar based on Cosine distance. Each node in the KG will be labelled with the signal.

4.5 Test intent in signals form

Each of the conditions and actions are tagged using the NER model to see if contains a *Comparision* class entity in it. For example, if

Table 1: Classwise metrics (%) of NER model without and with automatic feature engineering in domain corpus

Class	Prec (w/o)	Rec (w/o)	F1 (w/o)	Prec (w/)	Rec (w/)	F1 (w/)
Target module	66	68	67	81	80	80
Variable	78	76	77	84	88	86
Comparison	77	72	74	84	83	83
Action	79	68	73	90	81	85
Hardware	71	76	74	90	83	86
Value	77	69	73	89	93	91
Others	90	93	92	94	95	95

we get a Pre-condition in our test intent as follows: *signalA time is longer than signalB time.*

In the above condition, *longer than* entity belongs to the *Comparison* class. The *Comparison* class in our NER model will detect such comparisons and vector embedding's will be created for each of the signal descriptions i.e. for *signalA time* and *signalB time*. These will be mapped back to the most similar signal description in the dictionary and the corresponding signals will result in the following Pre-condition in signal form: *signalA-time > signalB-time.*

If the *Comparison* entity is not found in the condition or action, it means that it contains only a single signal and then the vector embedding is created for the entire condition or action and the corresponding signal is fetched. Once all the conditions and actions of the test intent are available in the signal form, it is then stored as a triplet using the KG Ontology mentioned above in a Neo4j database.

4.6 Automated Test Case generation

In the Phase 2 of KG pipeline, requirement statements are ingested into the KG creation tool which outputs the test intent in signals form. To check for missing conditions and actions not mentioned in the requirement statement, the KG can be queried to generate the complete test case.

5 RESULTS AND ANALYSIS

Once we have the test intent in signal form, we query the KG stored in Neo4j Database via CYPHER language queries. Some of the useful applications of querying the KG are mentioned below. The following performance metrics were used to compare results between the manual tester and proposed approach:

(1) True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN)
(2) Accuracy (Acc) = $\frac{TP+TN}{TP+TN+FP+FN}$
(3) Precision (Prec) = $\frac{TP}{TP+FP}$
(4) Recall (Rec) = $\frac{TP}{TP+FN}$
(5) F1 Score (F1) = $\frac{2*Recall*Precision}{Recall+Precision}$

5.1 Use case 1: Finding test cases when test intent has an Action

Consider the Requirement Statement: *If SignalA is on and ButtonA is pressed, then SignalB is on.* Since the test intent contains an Action the Pairs attribute of the node *buttonA=1* can be checked to generate the complete test case. The following CYPHER query can be executed: *MATCH rel=(action)-[r:'Post-condition']->(post) WHERE (action.name='buttonA=1' AND post.name='signalB=1') RETURN action.Pairs* . Filtering the results containing the test intent Pre-conditions, it will be found that *SignalE is on (signalE=1)* is a Post-condition that was missed in the requirement statement but is required to generate the complete test case.

Table 2: Confusion matrix for test intent identification in proposed approach with and without grammatical correctness in Requirement statements

Parameter	TP	FP	TN	FN
Pre-conditions (w/)	278	3	236	22
Actions (w/)	49	3	580	2
Post-conditions (w/)	164	0	456	3
Pre-conditions (w/o)	49	1	49	87
Actions (w/o)	10	2	76	3
Post-conditions (w/o)	26	1	43	32

5.2 Use case 2: Finding test cases when test intent has no Action

Consider the Requirement Statement: *If SignalD is on then SignalC is on.* Since the test does not contain an Action, we need to check if there are any missing actions or Post-conditions. The following CYPHER query can be executed: *MATCH rel=(action)-[r:'Post-condition']->(post) WHERE post.name='signalC=1') RETURN action.Pairs* . If this query returns no results, then indeed no actions need to be performed to achieve the Post-condition. In this case, to generate the complete test case, a query can be written to read the Individual-Pre-conditions attribute of the node *signalC=1*: *MATCH (post) WHERE post.name = 'signalC=1' RETURN post.Individual-Pre-conditions* . Executing this query, it will be found that *SignalF is on (signalF=1)* is a Pre-condition that was missed in the requirement statement but is required to generate the complete test case. For any of these queries, the result is always filtered by the test intent Pre-conditions.

5.3 Test case generation results on Cruise Control module

We have executed the proposed approach on the FRD provided by Robert Bosch Engineering and Business Solutions Private Limited. The FRD contains 200 requirement statements in English describing

the functionality required for a Cruise Control software system in a car. A KG was created from the software documents describing the cruise control functionality, past requirement statements in the project as well the past test reports from previous versions of the software. We compared our results for the test intent and test case coverage (Table 5) with the manually filled results given by the software tester.

5.4 Analysis of manual tester results

The test intent filled by the software tester had 436 Pre-conditions, 64 Actions and 225 Post-conditions. The test intent created by the tester is the benchmark for our results. In addition to this, the tester manually added 220 Pre-conditions and 104 Post-conditions that were missed in the requirement statement. However, it was noted from the test case review team that while all the additional Pre-conditions and Post-conditions were correctly added by the tester, 26 Actions were missing in the test cases. The test intent from the tester did not contain any errors or missing information which was verified by the test review team. This is usually the case as the scope of the test intent is only the requirement statement and it is easy for the tester to identify the correct test intent conditions and actions. However in creating the test cases, while all the conditions were captured correctly by the tester, 26 Actions to facilitate some of these Post-conditions were missed by the tester. Often times certain conditions or actions are missed by the tester while generating the test cases, when the system involves several hierarchies of inter-dependencies and traceability becomes extremely challenging.

5.5 Analysis of proposed approach results

The same 200 requirement statements were subjected to our KG pipeline. Since we found that 32 requirement statements were grammatically incorrect, we analyzed the performance of our pipeline to handle such statements (Table 3). Using various KG based queries, the proposed approach was able to capture an additional 220 Pre-conditions, 26 Actions and 85 Post-conditions for the test cases (Table 4). In the test intent with the proposed approach, the false positive rate is very low due to the CPT algorithm which prevents misclassifications in the test intent. Identifying actions in the test intent has the best performance as it is done with NER as compared to finding the Pre-conditions and Post-conditions which depend on the types of phrases and clauses used in the statement. We can see that the False Negative cases in grammatically incorrect statements are high as compared to grammatically correct statements mainly due to two reasons: CPT began to form incorrect parses for the statements; CRF based NER model being a sequence model was affected (Table 2). With the KG based queries, the proposed approach captured the 26 Actions that were missed by the manual tester for the test cases. There were certain False Positive and False Negative conditions and actions that were captured. This happens because in the information extraction process, some amount of inaccuracies are introduced by each component in the pipeline such as CPT algorithm, NER, and Sentence embedding's, which causes certain incorrect relationships between nodes in the KG. These inaccuracies manifest in the form of incorrect test case conditions and actions when the KG is queried.

Table 3: Performance metrics (%) for test intent identification in proposed approach with and without grammatical correctness in Requirement statements

Parameter	Acc	Prec	Rec	F1
Pre-conditions (w/)	95.3	98.9	92.6	95.6
Actions (w/)	99.2	94.2	96.0	95.0
Post-conditions (w/)	99.5	100	98.2	99.0
Pre-conditions (w/o)	52.6	98.0	36.0	52.6
Actions (w/o)	94.5	83.3	76.9	79.9
Post-conditions (w/o)	67.6	96.2	44.8	61.1

Table 4: Confusion Matrix for missing conditions in Test Case with Manual Tester and Proposed approach

Parameter	TP	FP	TN	FN
Manual Tester Pre-conditions	220	0	130	0
Manual Tester Actions	0	0	324	26
Manual Tester Post-conditions	104	0	246	0
KG query Pre-conditions	220	34	130	0
KG query Actions	26	5	319	0
KG query Post-conditions	85	35	211	19

Table 5: Performance metrics (%) for missing conditions in Test Case with Manual Tester and Proposed approach

Parameter	Acc	Prec	Rec	F1
Manual Tester Pre-conditions	100	100	100	100
Manual Tester Actions	92.6	Ind	0	0
Manual Tester Post-conditions	100	100	100	100
KG query Pre-conditions	91.1	86.6	100	92.8
KG query Actions	98.6	83.8	100	91.2
KG query Post-conditions	84.5	70.8	81.7	75.9

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed a methodology to generate a KG from software documents that are primarily unstructured and sparse. The proposed KG creation tool is able to extract the test intent from a requirement statement. This test intent is used to query the KG to extract an exhaustive list of test cases or even derive missing information required for the test case. Compared with traditional automated test case generation approaches, our proposed approach creates a KG from an extensive corpus automatically to capture the domain. The KG creation tool is developed to assist the software tester to test the system exhaustively. The future work that can be carried out are as follows: Firstly, an alternate approach to the NER model can be implemented using Transfer Learning, which uses a pre-trained neural network and fine tuned with the small amount of domain labelled data available. Secondly, work can be carried out in validating the information found in requirement statements using the KG. This will help in identifying requirements that contradict the information found in the KG.

REFERENCES

[1] Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D. Manning. 2015. Leveraging Linguistic Structure For Open Domain Information Extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 344–354. https://doi.org/10.3115/v1/P15-1034

[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[3] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Jr. Estevam R. Hruschka, and Tom M. Mitchell. 2010. Toward an architecture for never-ending language learning. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. 1306–1313.

[4] Tao Chen, Ruifeng Xu, Yulan Hec, and Xuan Wang. 2017. Improving sentiment analysis via sentence type classification using BiLSTM-CRF and CNN. *Expert Systems with Applications* 72 (2017). https://doi.org/10.1016/j.eswa.2016.10.065

[5] Luciano Del Corro and Rainer Gemulla. 2013. Clausie: Clause-based open information extraction. In *Proceedings of the 22nd International Conference on World Wide Web*. 355–366.

[6] Peter Fröhlich and Johannes Link. 2000. Automated test case generation from dynamic models. In *European Conference on Object-Oriented Programming*. Springer, 472–491.

[7] John Lafferty, Andrew McCallum, and Fernando C.N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning*. 282–289.

[8] Ze-Qi Lin, Xie Bing, Zou Yan-Zhen, Zhao Jun-Feng, Li Xuan-Don, Wei Jun, Sun Hai-Long, and Yin Gang. 2017. Intelligent development environment and software knowledge graph. *Journal of Computer Science and Technology* (2017), 242–249.

[9] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. 2006. Automatic test case generation for UML activity diagrams. In *Proceedings of the 2006 international workshop on Automation of software test*. ACM, 2–8.

[10] Software Systems Engineering Standards Committee of the IEEE Computer Society (Ed.). 2008. IEEE Standard for software and system test documentation. (2008).

[11] Udsanee Pakdeetrakulwong, Pornpit Wongthongtham, Waralak V. Siricharoen, and Naveed Khan. 2016. An ontology-based multi-agent system for active software engineering ontology. In *Mobile Networks and Applications*, Vol. 21. Springer, 65–88.

[12] Monalisa Sarma, Debasish Kundu, and Rajib Mall. 2007. Automatic test case generation from UML sequence diagrams. In *15th International Conference on Advanced Computing and Communications*. 60–65.

[13] Monalisa Sarma and Rajib Mall. 2007. Automatic test case generation from UML models. In *10th International Conference on Information Technology*. IEEE, 196–201.

[14] Amit Singhal. 2012. Introducing the Knowledge Graph: things, not strings. Retrieved Nov 2, 2019 from https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html

[15] Robyn Speer, Joshua Chin, and Catherine Havasi. 2017. ConceptNet 5.5: an open multilingual graph of general knowledge. In *AAAI'17 Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 4444–4451.

[16] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *WWW '07 Proceedings of the 16th international conference on World Wide Web*. 697–706.

[17] Luay Ho Tahat, Boris Vaysburg, Bogdan Korel, and Atef J. Bader. 2001. Requirement-based automated black-box test generation. In *25th Annual International Computer Software and Applications Conference*. IEEE, 489–495.

[18] Vladimir Tarasov, He Tan, Muhammad Ismail, Anders Adlemo, and Mats Johansson. 2016. Application of inference rules to a software requirements ontology to generate software test cases. In *OWL: Experiences and Directions–Reasoner Evaluation*. Springer, 82–94.

[19] Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and Accurate Shift-Reduce Constituent Parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 434–443.