Req2Spec: Transforming Software Requirements into Formal Specifications Using Natural Language Processing

Anmol Nayak[1(✉)], Hari Prasad Timmapathini[1], Vidhya Murali[1], Karthikeyan Ponnalagu[1], Vijendran Gopalan Venkoparao[1], and Amalinda Post[2]

[1] ARiSE Labs at Bosch, Bengaluru, India
{Anmol.Nayak,Hariprasad.Timmapathini,Vidhya.Murali, Karthikeyan.Ponnalagu,GopalanVijendran.Venkoparao}@in.bosch.com
[2] Robert Bosch GmbH, Stuttgart, Germany
Amalinda.Post@de.bosch.com

Abstract. [**Context and motivation**] Requirement analysis and Test specification generation are critical activities in the Software Development Life Cycle (SDLC), which if not done correctly can lead to defects in the software system. Manually performing these tasks on Natural Language Requirements (NLR) is time consuming and error prone. [**Question/problem**] The problem is to facilitate the automation of these activities by transforming the NLR into Formal Specifications. [**Principal ideas/results**] In this paper we present Req2Spec, a Natural Language Processing (NLP) based pipeline that performs syntactic and semantic analysis on NLR to generate formal specifications that can be readily consumed by HANFOR, an industry scale Requirements analysis and Test specification generation tool. We considered 222 automotive domain software requirements at BOSCH, 71% of which were correctly formalized. [**Contribution**] Req2Spec will be an aid to stakeholders of the SDLC as it seamlessly integrates with HANFOR enabling automation.

Keywords: Requirements formalization · Natural Language Processing · Requirements analysis · Test specification generation · Language model

1 Introduction

Software requirements analysis is one of the initial phases in the SDLC where requirements are analyzed on several aspects before being passed on to the downstream stakeholders for design, implementation and testing of the system. As there are many stakeholders involved in a software project delivery starting from the requirements engineer to the software tester, errors in the handling of requirements can percolate unnoticed. Getting early insights on the requirements is vital and recommended as it can reveal issues like inconsistencies, ambiguities, incompleteness [1]. There have been a few works that perform analysis of NLR [2–5],

V. Gervasi and A. Vogelsang (Eds.): REFSQ 2022, LNCS 13216, pp. 87–95, 2022.
https://doi.org/10.1007/978-3-030-98464-9_8

however they lack support for integration with HANFOR [6] and have not provided an end-to-end pipeline utilizing recent advances in NLP techniques for it to be leveraged across industry scale software projects. Industry scale NLR analysis tools such as IBM RQA [7] and QRA QVscribe [8] predominantly perform syntactic analysis (e.g. identifying vague terms, passive voice etc.) and minimal semantic analysis (e.g. they do not check for properties such as vacuity, consistency etc.). Test specification generation is another important phase in the later stages of the SDLC where significant amount of time and effort is spent. Some of the recent works have proposed automatic generation of test specification from NLR [9–12] using NLP techniques from which we have leveraged some of the components for syntactic and semantic information extraction.

Requirements formalization aims to transform NLR into pre-defined boilerplates having a restricted grammar, enabling large scale automated processing of requirements for downstream tasks such as requirements analysis and test specification generation. There have been previous attempts to formalizing NLR [13–15], however they expect the NLR to follow a restricted template/structure. Further, automated analysis of formal requirements for properties such as consistency and vacuity have been proposed [16–19], however they need the requirements to be already formalized or in the form of mathematical descriptions. While there exist several methods using formalized requirements, the widespread adoption in industry is still lacking as the hurdle to manually formalize requirements seems to be too high. With our Req2Spec method we want to lower this hurdle and we believe that it will enable utilization of formalized requirements even by requirements engineers without a background in formal methods.

We have integrated Req2Spec with HANFOR as it is an industry scale tool based on the specification pattern system by Konrad et al. [20]. It can also automatically translate the formal specifications into logics for downstream processing. HANFOR currently relies on manually formalized requirements prior to performing requirements analysis and test specification generation. Our work attempts to automate this step by using NLP techniques.

2 Background

HANFOR tool [21] consumes formalized NLR defined by an ID, a scope and a pattern. It supports 5 scopes, 4 regular patterns, 7 order patterns and 12 realtime patterns. A scope describes the boundary of the requirement. For e.g. a requirement with a *Globally* scope will hold true throughout the system, while a requirement with a *After EXPR* scope will hold true only after the expression (*EXPR*) is satisfied. A pattern describes the category of the requirement based on the pre-conditions, post-conditions, sequence of pre-conditions and post-conditions, and time duration elements. For example, a requirement with a time duration element only in the post-condition could have the pattern *If EXPR holds, then EXPR holds for at least DURATION*. The scopes and patterns are parameterized by expressions over system observables and durations. Our proposed pipeline is shown in Fig. 1. It is demonstrated with 2 scopes (*Globally* and *After EXPR*),

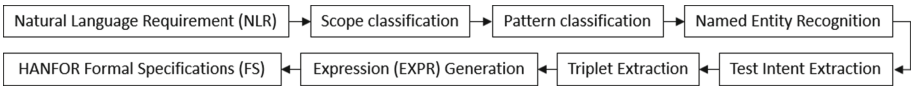

Fig. 1. Req2Spec pipeline.

1 regular pattern (*It is always the case that if EXPR holds, then EXPR holds as well*) and 1 realtime pattern (*If EXPR holds, then EXPR holds after at most DURATION*).

3 Req2Spec Pipeline

3.1 Dataset

NLR dataset consisted of 222 automotive domain requirements corresponding to the aforementioned chosen scopes and patterns as they cover the most common types of requirements found at BOSCH. For training and validation of the various NLP models, automotive software description documents dealing with functionalities such as cruise control, exhaust system, braking etc., along with the ground truths of the NLR dataset were annotated by experts.

3.2 Scope and Pattern Classification

Each NLR has to be associated with a scope and pattern to comply with HANFOR. We trained classification models for scope and pattern identification respectively using the SciBERT-Base-Scivocab-Uncased encoder [22] (a state-of-the-art model used in scientific domains) with a sequence classification head. The encoder and head were kept unfrozen and trained with the Adam [23] optimizer ($\mathrm{lr} = 1\mathrm{e}{-5}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1\mathrm{e}{-7}$). The following requirements are illustrative examples adhering to the chosen 2 scopes and 2 patterns:

1. Scope: *Globally*, Pattern: *It is always the case that if EXPR holds, then EXPR holds as well*:- If ignition is on, then fuel indicator is active.
2. Scope: *Globally*, Pattern: *If EXPR holds, then EXPR holds after at most DURATION*:- If ignition is on, then the wiper movement mode is enabled within 0.2 s.
3. Scope: *After EXPR*, Pattern: *It is always the case that if EXPR holds, then EXPR holds as well*:- Only after the vehicle speed is larger than 60 kmph, If the cruise control button is pressed, then the cruise control mode is activated.
4. Scope: *After EXPR*, Pattern: *If EXPR holds, then EXPR holds after at most DURATION*:- Only after the vehicle is in reverse gear, If the accelerator is pressed, then the rear view camera is activated within 2 s.

3.3 Named Entity Recognition (NER)

NER is the task of identifying and classifying named entities of a domain. We trained the NER model by using the SciBERT-Base-Scivocab-Uncased encoder along with a token classification head in the following setting (as described by a recent work which addresses many of the challenges of NER in the automotive domain [24]): Masked Language Modelling was performed on the pretrained SciBERT encoder with automotive domain text using the Adam optimizer ($\text{lr} = 5\text{e}{-5}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1\text{e}{-8}$). This encoder and the head were then kept unfrozen for NER training with the Adam optimizer ($\text{lr} = 1\text{e}{-5}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1\text{e}{-7}$). The annotation for NER was consisting of 9 automotive specific classes: Other (words outside named entities e.g. the, in), Signal (variables holding quantities e.g. torque), Value (quantities assigned to signals e.g. true, false), Action (task performed e.g. activation, maneuvering), Function (domain specific feature e.g. cruise control), Calibration (user defined setting e.g. number of gears), Component (physical part e.g. ignition button), State (system state e.g. cruising state of cruise control) and Math (mathematical or logical operation e.g. addition).

3.4 Test Intent Extraction

Software requirements describe the expected functionality in terms of test intent components, namely Pre-conditions and Post-conditions. Pre-conditions are the conditions which are expected to be satisfied before the Post-conditions can be achieved. For example in the requirement: *If ignition is on, then fuel indicator is active*, the Pre-condition is *ignition is on* and the Post-condition is *fuel indicator is active*. The test intent components are the primary source of information used to fill the *EXPR* slots of the scope and pattern. We utilized the Constituency Parse Tree (CPT) based syntactic test intent extraction algorithm [11] as it is able to separate dependent clauses (Pre-conditions) and independent clauses (Post-conditions) using grammar sub-tree structures.

3.5 Triplet Extraction

The test intent components have to be converted into expressions before being filled into the *EXPR* slots of the scope and patterns. For this we first convert each test intent component into a Subject-Verb-Object (SVO) triplet. Since traditional triplet extraction algorithms such as OpenIE [25] and ClausIE [26] have been designed from open source text (similar to Wikipedia articles), the quality of the extracted triples is hampered when applied to software engineering domain corpus which contains lexica and sentence structure that is niche. Hence, we have designed the following CPT based triplet extraction algorithm in our pipeline:

1. CPT is constructed using the Stanford CoreNLP [27] library for the condition and recursively traversed.

2. Subject is the sub-string until a Verb Phrase (VP) is encountered. Verb is the sub-string from the beginning of the VP until a Noun Phrase (NP) or Adjective (JJ) is encountered. Object is the sub-string from NP/JJ to the end of the condition. If a Infinitival to (TO)/VP is encountered in the Object, then the words occurring until (including) the TO/VP are concatenated to the Verb string of the triplet and the remaining sub-string is the Object.
3. This step is triggered if Step 2 resulted in a triplet with no Verb and Object strings: Subject is the sub-string until a TO/Preposition (IN) is encountered. Verb is the sub-string corresponding to TO/IN. Object is the sub-string after the TO/IN sub-string. If a TO is encountered in the Object, then the words until (including) the TO are concatenated to the Verb string of the triplet and the remaining sub-string is kept as the Object.
4. This step is triggered if Step 3 resulted in a triplet with no Object string: Subject is the sub-string until a VP/TO is encountered. Verb is the sub-string from the beginning of the VP/TO until any VB (all verb forms)/RB (Adverb)/IN is encountered. Object is the sub-string beginning from VB/RB/IN until the end of the condition.

3.6 Expression (EXPR) Generation

The natural language SVO triplets have to be rewritten into an equation format where natural language aliases are resolved. The Subject and Object are mapped to system observables (can be thought of as variables used in software code development) and the Verb is mapped to an operator. For example, *ignition (S) - is (V) - on (O)* will be mapped to $ig\_st = on$. A system observables (variables) dictionary is used for mapping the Subject and Object, whose keys are natural language descriptions of the variables and the values are the variables. Similarly, the Verb is mapped to operators using an operator dictionary, whose keys are natural language descriptions of the operators and the values are the operators. This mapping happens in 4 steps:

1. The triplet is tagged with the NER model.
2. A vector representation is created for the Subject, Verb and Object of the triplet using a pre-trained Sentence-BERT (SBERT) [28] model.
3. Subject is mapped to the variable whose vector representation of its natural language description was closest based on cosine similarity. Similarly, the Verb is mapped to the closest matching operator in the operator dictionary.
4. Object mapping follows the above process only if it does not contain a Value named entity, otherwise the Value named entity is retained as the Object.

3.7 HANFOR Formal Specifications (FS)

As the final step the *EXPR* and *DURATION* slots of the scope and pattern corresponding to the requirement have to be filled. Once filled, the scope and pattern are tied together resulting in the formal specification. Table 1 shows the intermediate outputs generated during the formalization of an illustrative sample NLR. The scope *EXPR* slot filling happens as follows:

- If the scope is *Globally*, then there is no *EXPR* slot to fill.
- If the scope is *After EXPR*, then each pre-condition whose Subject contains temporal prepositions indicating time following such as *after, beyond, subsequent to, following* etc., its expression will be filled in this *EXPR* slot.
- In case there exist multiple such pre-conditions, their expressions are then tied together with AND and OR operators appropriately.

The pattern *DURATION* slot filling happens as follows:

- If the pattern is *It is always the case that if EXPR holds, then EXPR holds as well*, then there is not *DURATION* to fill.
- If the pattern is *If EXPR holds, then EXPR holds after at most DURATION*, then the Regular Expression *\d+[.]?\d+? ?(?:seconds|minutes|hours|time units')* is checked against each post-condition to extract any time duration element. As this pattern applies a single *DURATION* element across all the post-conditions, the sub-string returned from the Regular Expression will be stripped from the post-conditions and filled in the *DURATION* slot.

The pattern *EXPR* slot filling happens as follows:

- In case there are multiple pre-conditions and post-conditions, their expressions are then tied together with AND and OR operators appropriately.
- For both the selected patterns, the pre-condition expressions are filled in the *EXPR* slot attached to the *If* clause, and the post-conditions expressions are filled in the *EXPR* slot attached to the *then* clause.

Table 1. End-to-end flow of a sample requirement through the Req2Spec pipeline.

Component	Output
NLR	If ignition is on, then fuel indicator is active
Scope	Globally
Pattern	It is always the case that if EXPR holds, then EXPR holds as well
NER	Signal: ignition; Component: fuel indicator; Value: on, active
Test intent extraction	Pre-cond: ignition is on, Post-cond: fuel indicator is active
Triplet extraction	Pre-cond: ignition-is-on, Post-cond: fuel indicator-is-active
Expression generation	Pre-cond: ig_st = on, Post-cond: fuel_ind = active
Formal specification	Globally, It is always the case that if ig_st = on holds, then fuel_ind = active holds as well

4 Results

Table 2 summarizes the performance of the different NLP components in the pipeline. 71% of the NLR requirements were successfully formalized by the

Req2Spec pipeline, leading to significant decrease in the time spent on manual formalization. Further, we believe that even though 29% of the requirements had formalization errors, they still provide a head start to the engineer who can make minor edits before feeding them to HANFOR. The error rate can be attributed to the following reasons:

1. Irreducible errors of the machine learning models.
2. Syntactic components of the pipeline such as Test Intent Extraction and Triplet Extraction are impacted by the quality of grammatical correctness and ambiguities in the requirements. For example, consider the requirement: *When cruise control is activated and speed is above 60 kmph or wiper is activated then lamp turns on.* It is unclear which of the following Test Intent pre-conditions combination is valid:
   - (cruise control is activated AND speed is above 60 kmph) OR (wiper is activated)
   - (cruise control is activated) AND (speed is above 60 kmph OR wiper is activated)
3. As the pipeline is linear, the failure of even a single component causes the error to cascade till the end, thereby leading to an incorrect formal specification.

Table 2. Performance (%) of the Syntactic (Syn) and Semantic (Sem) NLP components used in Req2Spec pipeline.

Component	Algorithm	Type	Precision	Recall	F-1	Accuracy
Scope classification	SciBERT	Syn+Sem	93	93	93	98
Pattern classification	SciBERT	Syn+Sem	95	96	96	96
Named entity recognition	SciBERT	Syn+Sem	83	83	83	88
Test intent extraction	CPT	Syn	–	–	–	79.27
Triplet extraction	CPT	Syn	–	–	–	88.73
Expression generation	SBERT	Sem	–	–	–	93.24
Formal specifications	–	–	–	–	–	71.61

5 Conclusion and Future Work

In this paper we have proposed Req2Spec, a NLP based pipeline that performs syntactic and semantic analysis to formalize software requirements into HANFOR compliant formal specifications, which can then be used to perform tasks like requirements analysis and test specification generation. We demonstrated our pipeline on 4 different types of requirements (2 scopes and 2 patterns), out of which 71% of the requirements resulted in the correct formal specifications, giving strong confidence on the feasibility of the pipeline. We believe that this can lead to productivity gains for the various stakeholders of the SDLC and overall improve the software quality, as the manual interventions required will decrease significantly. Our future work will focus on including datasets beyond the automotive domain and also extending the pipeline to handle additional scopes and patterns to increase coverage on different types of requirements.

References

1. IEEE: IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1998, pp. 1–40 (1998). https://doi.org/10.1109/IEEESTD.1998.88286
2. Fatwanto, A.: Software requirements specification analysis using natural language processing technique. In: 2013 International Conference on QiR, pp. 105–110. IEEE (2013)
3. Dalpiaz, F., van der Schalk, I., Lucassen, G.: Pinpointing ambiguity and incompleteness in requirements engineering via information visualization and NLP. In: Kamsties, E., Horkoff, J., Dalpiaz, F. (eds.) REFSQ 2018. LNCS, vol. 10753, pp. 119–135. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77243-1_8
4. Zhao, L., et al.: Natural language processing (NLP) for requirements engineering: a systematic mapping study. arXiv preprint arXiv:2004.01099 (2020)
5. Gervasi, V., Riccobene, E.: From English to ASM: on the process of deriving a formal specification from a natural language one. Integration of Tools for Rigorous Software Construction and Analysis, p. 85 (2014)
6. Becker, S., et al.: Hanfor: semantic requirements review at scale. In: REFSQ Workshops (2021)
7. IBM Engineering Requirements Quality Assistant tool. https://www.ibm.com/in-en/products/requirements-quality-assistant. Accessed 13 Oct 2021
8. QRA QVscribe tool. https://qracorp.com/qvscribe/. Accessed 13 Oct 2021
9. Dwarakanath, A., Sengupta, S.: Litmus: generation of test cases from functional requirements in natural language. In: Bouma, G., Ittoo, A., Métais, E., Wortmann, H. (eds.) NLDB 2012. LNCS, vol. 7337, pp. 58–69. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31178-9_6
10. Nayak, A., et al.: Knowledge graph from informal text: architecture, components, algorithms and applications. In: Johri, P., Verma, J.K., Paul, S. (eds.) Applications of Machine Learning. AIS, pp. 75–90. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-3357-0_6
11. Nayak, A., Kesri, V., Dubey, R.K.: Knowledge graph based automated generation of test cases in software engineering. In: Proceedings of the 7th ACM IKDD CoDS and 25th COMAD, pp. 289–295 (2020)
12. Kesri, V., Nayak, A., Ponnalagu, K.: AutoKG-an automotive domain knowledge graph for software testing: a position paper. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 234–238. IEEE (2021)
13. Böschen, M., Bogusch, R., Fraga, A., Rudat, C.: Bridging the gap between natural language requirements and formal specifications. In: REFSQ Workshops (2016)
14. Fatwanto, A.: Translating software requirements from natural language to formal specification. In: 2012 IEEE International Conference on Computational Intelligence and Cybernetics (CyberneticsCom), pp. 148–152. IEEE (2012)
15. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Generation of formal requirements from structured natural language. In: Madhavji, N., Pasquale, L., Ferrari, A., Gnesi, S. (eds.) REFSQ 2020. LNCS, vol. 12045, pp. 19–35. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44429-7_2
16. Langenfeld, V., Dietsch, D., Westphal, B., Hoenicke, J., Post, A.: Scalable analysis of real-time requirements. In: 2019 IEEE 27th International Requirements Engineering Conference (RE), pp. 234–244. IEEE (2019)

17. Moitra, A., et al.: Towards development of complete and conflict-free requirements. In: 2018 IEEE 26th International Requirements Engineering Conference (RE), pp. 286–296. IEEE (2018)
18. 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: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 420–426. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_30
19. Post, A., Hoenicke, J.: Formalization and analysis of real-time requirements: a feasibility study at BOSCH. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 225–240. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_18
20. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381 (2005)
21. HANFOR tool. https://ultimate-pa.github.io/hanfor/. Accessed 13 Oct 2021
22. Beltagy, I., Lo, K., Cohan, A.: SciBERT: a pretrained language model for scientific text. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 3615–3620 (2019)
23. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
24. Nayak, A., Timmapathini, H.P.: Wiki to automotive: understanding the distribution shift and its impact on named entity recognition. arXiv preprint arXiv:2112.00283 (2021)
25. Angeli, G., Premkumar, M.J.J., Manning, C.D.: 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), pp. 344–354 (2015)
26. Del Corro, L., Gemulla, R.: ClausIE: clause-based open information extraction. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 355–366 (2013)
27. 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)
28. Reimers, N., Gurevych, I.: Sentence-BERT: sentence embeddings using siamese BERT-networks. arXiv preprint arXiv:1908.10084 (2019)