NLP-Based Requirements Formalization for Automatic Test Case Generation

Robin Gröpler[1], Viju Sudhi[1], Emilio José Calleja García[2] and Andre Bergmann[2]

[1]ifak Institut für Automation und Kommunikation e.V., 39106 Magdeburg, Germany
[2]AKKA Germany GmbH, 80807 München, Germany

Abstract
Due to the growing complexity and rapid changes of software systems, the assurance of their quality becomes increasingly difficult. Model-based testing in agile development is a way to overcome these difficulties. However, major effort is still required to create specification models from a large set of functional requirements provided in natural language. This paper presents an approach for a machine-aided requirements formalization technique based on Natural Language Processing (NLP) to be used for an automatic test case generation. The goal of the presented method is to automate the process of model creation from requirements in natural language by utilizing appropriate algorithms, thus reducing cost and effort. The application of our procedure will be demonstrated using an industry example from the e-mobility domain. In this example, requirement models are generated for a charging approval system within a larger vehicle battery charging application. Additionally, existing tools for automated model synthesis and test case generation are applied to our models to evaluate whether valid test cases can be generated.

Keywords
Requirements analysis, natural language processing, test generation

1. Introduction

In the life cycle of a device, component or system in industrial use, a rapidly changing and growing number of requirements and the associated increase in features and feature changes inevitably lead to an increasing effort for verifying requirements and testing of the implementation. To manage test complexity and reduce necessary test effort and cost, agile methods for model-based testing have been developed [1]. The effectiveness of model-based testing highly depends on the quality of the used specification model. The creation and maintenance of well-defined specification models is therefore crucial and usually comes with high effort and cost. This is especially true in agile development, where requirements are subject to frequent changes.

In this context, an approach for requirements-based testing was developed that enables efficient test processes, see Fig. 1. Model synthesis and model-based test generation methods are used to systematically and efficiently create a test suite that contains suitable test cases. This approach is based on behavioral requirements that serve as input for model synthesis. The only

29th International Workshop on Concurrency, Specification and Programming (CS&P'21)
robin.groepler@ifak.eu (R. Gröpler); viju.sudhi@ifak.eu (V. Sudhi); emiliojose.calleja@gmail.com (E. J. Calleja García); andre.bergmann@akka.eu (A. Bergmann)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-2951/paper15.pdf

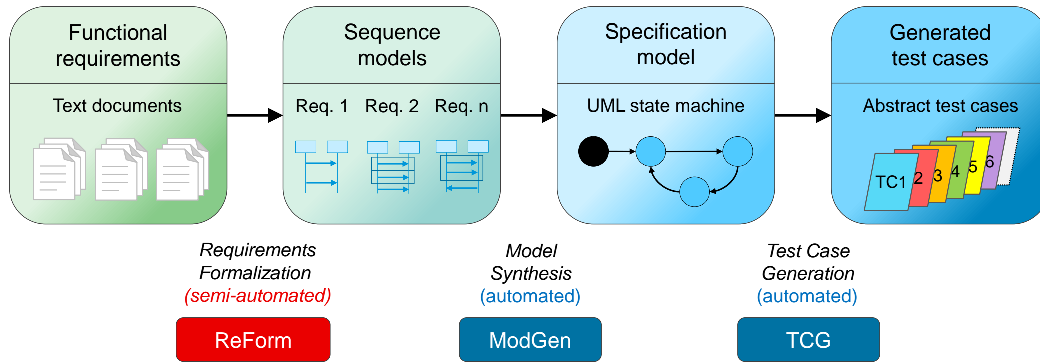

Figure 1: Toolchain for requirements-based test case generation.

time-consuming manual step is the creation of requirement models from textual requirements documents.

Recent advances in natural language processing show promising results to organize and identify desired information from raw text. As a result, NLP techniques show a growing interest in automating various software development activities like test case generation. Several NLP approaches and tools have been investigated in recent years aiming to generate test cases from preliminary requirements documents [2, 3, 4]. A major drawback of existing methods is the use of controlled natural language or templates that force the requirements engineer or designer not only to concentrate on the content but also on the syntax of the requirement. Furthermore, those algorithms are in general not applicable to existing requirements.

In this work, we propose a *new, semi-automated technique for requirements-based model generation* that reduces human effort and supports frequent requirements changes and extensions. The aim of our approach is to develop a method that

1) can handle an extended range of domains and formats of requirements, i.e. it is not limited to a specific template or controlled natural language, and
2) provides enhanced but easily interpretable intermediate results in the form of a textual and graphical representation of UML sequence diagrams.

Our approach utilizes an existing NLP parser to obtain basic syntactic information about the words and their relationship to each other. Based upon this information, several rule-based steps are performed in order to identify relevant syntactic entities which are then mapped to semantic entities. Finally, these entities are used to form requirement models as UML sequence diagrams. The main contributions of this work are

1) the development of a rule-based approach based on NLP information that automates the various steps involved in deriving requirement models, and
2) the evaluation on an industrial use case using meaningful metrics that demonstrates the good quality of our approach.

The paper is structured as follows. In Section 2, we briefly outline related work on NLP-based requirements formalization methods. In Section 3, we present the individual steps of our

methodology for deriving requirement models from textual descriptions. The method is applied to the battery charging approval system presented in Section 4. In Section 5, we define several evaluation metrics and demonstrate the results of the application. Finally, a conclusion and outlook is given in Section 6.

2. Related work

In order to circumvent the challenges of analyzing highly complex requirements, many authors restrict their NLP approaches to a specific domain or a prescribed format. In [5], the authors propose an algorithm creating activity diagrams from requirements following a predefined structure. They consider the SOPHIST method which performs a refinement and formalization of structured texts by introducing text templates with a defined syntactical structure [6]. In [7], a small set of structural rules was developed to address common requirement problems including ambiguity, complexity and vagueness. In [8], requirements are expected to be written in a controlled natural language and are supposed to be from Data-Flow Reactive Systems (DFRS). The approach in [9] is to generate test cases from use cases or user stories, both of which have to comply with a specified format. In [10], requirements engineers shall be supported with formalization of use case descriptions by writing pre-conditions and post-conditions in a predefined format from which test cases can be generated automatically. Likewise [11] aims to find and complete missing test cases by parsing exceptional behaviors from Javadoc comments written in natural language, provided the documentation is in a specified template. [12] relies on the artifacts that the programmers create while developing the product which belong to a smaller subset of specifications.

Even for simple syntactical structures of requirements it is still necessary to enable the requirements engineer to review the intermediate results, i.e. the generated model artifacts, and to adjust them where necessary. The toolchain of [13] involves eliciting requirements according to Restricted Use Case Modeling (RUCM) specifications. This applies to the work of [14], where the authors attempt to generate executable test cases from a Restricted Test Case Modeling (RTCM) language which restricts the style of writing test cases. This becomes an additional overhead to the requirement engineers who draft formal requirements. Additionally, the users are expected to inspect the generated OCL constraints before proceeding to test case generation. Similarly, in [15] the authors explore the possibility of test case generation using Petri Net simulation; however the interpretability of Colored Petri Nets as proposed in the approach may vary depending on the user's level of expertise. These intermediate results may not be easily understood by the user and it may be cumbersome for him to fine-tune or modify the predictions before generating reliable test cases.

A notable work from the authors of [16] makes use of recursive dependency matching to formulate test cases. Though our approach aligns with theirs in this step, we attempt to generate test cases from a broader set of functional requirements while they restrict themselves with user stories from which a cause-effect relationship can be learnt.

3. Methodology

We utilize an existing NLP parser and use a rule-based algorithm to perform the transformation from requirements written in natural language to requirement models. Our rule set tries to conceive all relevant rules that could satisfactorily parse the input behavioral requirement and extract its semantic content.

3.1. Linguistic pre-processing

The behavioral requirements are, in general, complex by nature. In order to reliably extract the syntactic and semantic content of these requirements, a thorough linguistic pre-processing is indispensable. For this stage, we rely on spaCy (v2.1.8) [17] - a free, open-source library for advanced Natural Language Processing. We follow the basic NLP pipeline including tokenization, lemmatization, part-of-speech (POS) tagging and dependency parsing in various stages of the algorithm.

3.1.1. Pronoun resolution

Though the formal requirements tend to avoid first person (I, me, etc.) or second person (you, your, etc.) pronouns, they may contain third person neutral pronouns (it, they, etc.) [18]. These pronouns are identified and resolved with the farthest subject, inline with the algorithm proposed in [19] and [20]. Owing to the simplicity of the task, we assume there is no particular need to use more sophisticated algorithms checking grammatical gender and person while resolving pronouns. However, we attempt to resolve pronouns only if the grammatical number of the pronoun agrees with that of the antecedent. Since pleonastic pronouns (pronouns without a direct antecedent) do not affect the algorithm, they are cited but not replaced.

Example: Consider the requirement "If the temperature of the battery is below Tmin or it exceeds Tmax, charging approval has to be withdrawn". Here, the pronoun *it* is resolved with its antecedent *the temperature of the battery*.

3.1.2. Decomposition

Textual requirements with multiple conditions and conjunctions are hard to be transformed and mapped to individual relations. This demands decomposition of complex requirements into simple clauses [21]. Multiple conditions (sentences with multiple *if*s, *while*s, etc.), root conjunctions (sentences with multiple *roots* connected with a conjunction) and noun phrase conjunctions (sentences with multiple subjects and/or objects connected with a conjunction) are decomposed to simple primitive clauses.

We resort to the syntactic dependencies obtained from the parser to decompose requirements. The algorithm considers the token(s) with dependency *mark* to decompose multiple conditions and dependency *conj* for decomposing root and noun phrase conjunctions. The span of the sub-requirement can then be defined by identifying the *edges* (for e.g. the *left-most edge* refers to the token towards the left of the dependency graph with which the parent token holds a syntactic dependency) of the token of interest.

Example: In the requirement "If the temperature of the battery is below Tmin or the temperature of the battery exceeds Tmax, charging approval has to be withdrawn", the root conjunction (arising from the two roots *is* and *exceeds*) and the subsequent multiple conditions (arising from *if*) are decomposed to three sub-requirements as "[if the temperature of the battery is below Tmin] *or* [if the temperature of the battery exceeds Tmax], [charging approval has to be withdrawn]".

3.2. Syntactic entity identification

Almost all behavioral requirements describe a particular action (linguistically, *verb*) done by an agent (linguistically, *subject*) on the system of interest (linguistically, *object*). This motivates the idea of identifying syntactic entities from the requirements. The algorithm identifies these syntactic entities by checking the dependencies of tokens with the root.

1) Action: The main action verb in the requirement (mostly, with the dependency *ROOT*) is identified and called an action. The algorithm particularly distinguishes the type of actions as: *Nominal action* which has a noun and a verb together (e.g. *send a message*), *Boolean action* which can take a Boolean constraint (e.g. *is withdrawn*) and *Simple action* which has only an action verb (e.g. *send*).
   In addition, the algorithm also tries to identify the verb type(s) (transitive, dative, prepositional, etc.) as suggested in [21] to supplement the syntactic significance of action types. This is essential particularly when we rely on action types for relation formulation.
2) Subjects and Objects: The tokens with dependencies *subj* and *obj* (and their variants like *nsubj*, *pobj*, *dobj*, etc.) are identified mostly as Subjects and Objects, respectively. They can be noun chunks (e.g. *temperature of the battery*), compound nouns (e.g. *battery temperature*) or single tokens (e.g. *battery*) in the requirement.

Also, we noted that there are several requirements involving a logical comparison (identified as an adjective or an adverb) between the expressed quantities. In order to identify comparisons (e.g. *greater than*, *exceeds*, etc.) in the requirement, we utilize the exhaustive synonym hyperlinks from Roget's Thesaurus [22] and map them to the corresponding equality (=), inequality (!=), inferiority (<, <=) and superiority (>, >=) symbols.

Example: From the sub-requirements "[if the temperature of the battery is below Tmin] *or* [if the temperature of the battery exceeds Tmax], [charging approval has to be withdrawn]", the system identifies *Battery_Temperature* and *Charging_Approval* as Subjects, *Tmin* and *Tmax* as Objects and *withdrawn* as a Boolean Action. Also, the comparison term *below* is mapped as < and *exceeds* is mapped as >.

3.3. Semantic entity identification

Semantic entities are tightly coupled with the end application which translates the parsed syntactic information to sequence diagrams and then to abstract test cases. The semantic

Table 1
Mapping of syntactic to semantic entities

Syntactic entities	Semantic entities
Action	Signal
Action constraints	Attributes
Subject / Object	Actor / Component

entities are defined from the perspective of interactions in a sequence diagram and are outlined below. The algorithm derives these entities from their syntactic counterparts[1].

1) Actor or Component: The participants involved in an interaction are defined as actors and components. To differentiate other participants from the system under test (SUT), component is always considered as the SUT.
2) Signal: The interaction between different participants is defined as a signal.
3) Attributes: The variables holding the status at different points of interaction are defined as attributes.
4) State: This refers to the initial, intermediate and final states of an interaction.

Semantic entities demand additional details for completeness. For example, if the value of an attribute is not given, it can not be initialized in its corresponding test case. Likewise, for each signal the corresponding actor needs to be identified. For each requirement, the direction of communication (*incoming*: towards the system under test or *outgoing*: from the system under test) should be identified. In cases where the algorithm lacks the desired ontology information, user input is demanded to update these values.

It is worth noting that the separation of the entities as syntactic (application independent but grammar dependent) and semantic (application dependent but grammar independent) gives more flexibility to the algorithm to be used in parts also in a different environment than the description language considered here. However, the mapping from the syntactic entities to their semantic counterparts can be completely automated with stricter rules or can be accomplished with user intervention and validation.

Example: From the sub-requirements "[if the temperature of the battery is below Tmin] *or* [if the temperature of the battery exceeds Tmax], [charging approval has to be withdrawn]", the identified Subjects (*Battery_Temperature* and *Charging_Approval*) are mapped as Signals and the identified Objects (*Tmin* and *Tmax*) are mapped as Attributes. Here, the identified Action *withdrawn* is also considered as an Attribute owing to the semantics of its corresponding Boolean Signal. Additionally, we can arrive at the Actor for *Battery_Temperature* as *battery*. However, the Actor of *Charging_Approval* is ambiguous (or rather unknown). Likewise, Attribute values should either be passed by the user or they remain uninitialized in the resulting test case.

[1]Note that the algorithm maps syntactic to semantic entities with more complex rules (including action types and verb types). In Table 1, we have presented only the most primitive ones for brevity. This difference is also detailed in the example where an Action is considered as an Attribute and a Subject is mapped to a Signal.

3.4. Transformation to requirement model

For the description of the formal requirements a simple text-based domain-specific language (DSL) is used, the ifak requirements description language (IRDL) [23]. This notation for requirement models was developed on the basis of UML sequence diagrams and is specially adapted to the needs of describing requirements as sequences. The IRDL defines a series of model elements (e.g. components, messages) with associated attributes (name, description, recipient, sender, etc.) and special model structures (behavior based on logical operators or time). Functional, behavior-based requirements are described textually using IRDL and can then be visualized graphically as sequence diagrams (Fig. 2).

Once the entities are mapped and validated, the algorithm forms IRDL relations for each clause and then combines them together to form relations for the whole requirement. IRDL defines mainly two types of relations:

1) Incoming messages: SUT receives these messages provided the guard expression evaluates to be true and then continues to the next sequence in an interaction. IRDL defines this class of messages as 'Check'.
2) Outgoing messages: SUT sends these messages to other interaction participants with the content defined in the signal. In IRDL, these messages are denoted as 'Message'.

```
Check(Actor->Component): Signal[guard expression];
Message(Component->Actor): Signal(signal content);
```

As an intermediate result, the user is shown the formulated IRDL relations along with the sequence diagram corresponding to the requirement and is asked if the IRDL and the corresponding sequence diagram are correct. In case the user wants to further modify the relation formulation, the algorithm repeats from the mapping of syntactic entities to semantic entities. This continues until the user confirms the model is satisfactory.

Example: IRDL relations for the example requirement "If the temperature of the battery is below Tmin or it exceeds Tmax, charging approval has to be withdrawn", after the above-mentioned steps is shown in Fig. 2.

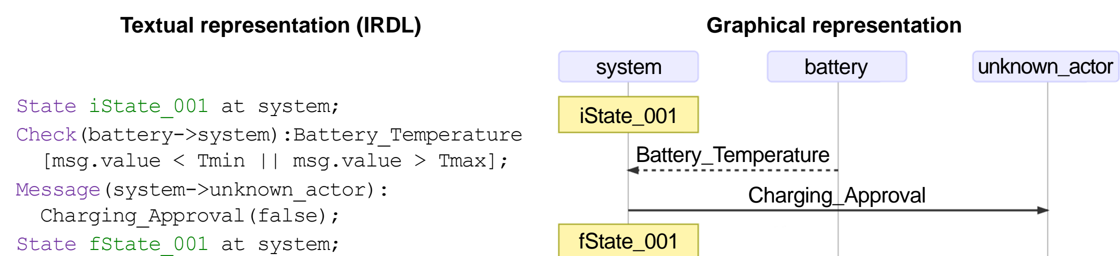

Figure 2: Visualization of a requirement model in IRDL and as sequence diagram.

3.5. Model synthesis and test generation

The formalized requirements of the SUT can be combined to a specification model using existing methods for model synthesis [23]. The sequence elements described before, are transformed using a rule-based algorithm into equivalent elements of a UML state machine.

After model synthesis, test cases can be automatically generated from the state machine using an existing method for model-based test generation [24]. Selecting a specific graph-based coverage criteria such as ”all paths”, ”all decisions”, ”all places” or ”all transitions”, the state machine is transformed into a special form of a Petri net from which abstract test cases in the form of sequence diagrams can be generated. In this way, the approach allows modeling of even complex system behavior and applying graph-based coverage criteria to the entire system model.

4. Application

The toolchain for requirements-based model and test case generation presented in the previous section is applied to an industrial use case from the e-mobility domain. The use case describes a system for charging approval of an electric vehicle in interaction with a charging station. The industrial use case was defined by AKKA and aims to provide a typical basic scenario and development workflow in software development for an automotive electronic control unit (ECU). It does so by defining requirements, using model-based software development and deploying the functionality on an ECU.

The use case has to be seen in the context of an electric car battery that is supposed to be charged. The function “charging approval” implements a simple function, which decides upon specific input signals, if the charging process of the battery is allowed or not. For example, charging approval is given or withdrawn depending on the battery temperature, voltage or state of charge, the requested current is adjusted according to the battery temperature, and error behavior is handled for certain conditions. This is a continuous process, i.e. the signal values may change over time. A more detailed technical description of the industrial use case can be found in [25]. To fulfill the requirement of model-based software development, the module is implemented in Matlab Simulink. Matlab Simulink Coder is used to generate C/C++ code that can be compiled and deployed to the target. A Raspberry Pi is used to simulate some but not all aspects of an ECU. A basic overview of the charging approval system and its interfaces to the environment is given in Fig. 3.

Figure 3: Process overview of charging approval system.

5. Results

The battery charging approval system described in the former section is used to evaluate the proposed method. We first define the used evaluation metrics and then demonstrate the results. To our knowledge, there are no available tools with similar input and output properties as our tool that enable a direct comparison.

5.1. Evaluation metrics

Let R be the set of textual requirements. For a requirement $r \in R$, let X_r be the set of expected artifacts and Y_r be the set of generated artifacts. Here, *artifacts* refer to all the semantic entities including the relation indicators. Let $X = \bigcup_{r \in R} X_r$ denote the set of expected artifacts in all requirements and $Y = \bigcup_{r \in R} Y_r$ the set of generated artifacts in all requirements. Then we define the following metrics to measure the performance of the method.

1) **Completeness:** For an individual requirement, this metric denotes the number of expected artifacts $x \in X_r$ for which a corresponding (not necessarily identical) generated artifact $y \in Y_r$ exists, in relation to the total number of expected artifacts $|X_r|$.
2) **Correctness:** For an individual requirement, this metric denotes the number of generated artifacts $y \in Y_r$ for which a corresponding, semantically identical (up to naming conventions) expected artifact $x \in X_r$ exists, in relation to the total number of generated artifacts $|Y_r|$.
3) **Consistency:** This metric denotes the number of generated artifacts $y \in Y$ for which a corresponding expected artifact $x \in X$ exists and is used identically in all requirements $r \in R$, in relation to the total number of generated artifacts $|Y|$.

The *macro average* for completeness and correctness, respectively, is then given by the mean value of all individual percentage values for all $r \in R$. The *micro average* is given by the sum of all values in the numerator divided by the sum of all values in the denominator for all $r \in R$.

Example: In order to assert the evaluation metrics in detail, consider the requirement clause *'if the SoC of the battery is below SoC_max'*.

Expected: `Check(charging_management->system): Battery_SoC[msg.Ialue < SoC_max];`
Generated: `Check(battery->system): battery_SoC[msg.value < SoC_max];`

For the metric *completeness*, we check if all the expected artifacts (i.e. *Check*, *charging_management*, *Battery_SoC*, etc.) are generated by the algorithm. In this case, we can see that all of them were generated. For obtaining the *correctness*, we check if those generated artifacts are semantically correct. In this case, though we expect the actor *charging_management*, the algorithm generates *battery*. This reduces the value of correctness. If the algorithm generates *battery_SoC* for every occurrence of 'SoC of battery' across all the requirements, it is considered *consistent* for this artifact.

Table 2
Evaluation of the algorithm on the charging approval system

	without domain knowledge		with domain knowledge	
	macro avg.	**micro avg.**	**macro avg.**	**micro avg.**
Completeness	78.2%	79.8%	81.4%	84.1%
Correctness	74.9%	78.8%	78.3%	82.1%
Consistency	94.1%		96.4%	

5.2. Requirements formalization

As part of the demonstrated use case, AKKA has provided functional requirement documents describing the expected behavior for the relevant SUT. To apply the NLP-based requirements formalization method, each statement is treated as a separate entity for which a well-defined requirement model is created. Overall, the charging approval SUT is described by 14 separate requirement statements.

The results of our evaluation are shown in Table 2. We have determined the individual correctness and completeness values and calculated the macro and micro average for them. We avoided double counting of identical entity detections as not to skew the results. As mentioned above, if an actor or value of an attribute is not explicitly mentioned in the textual requirement, it cannot be detected by the algorithm. Therefore we also show the results using domain knowledge, which could be in the form of a predefined list of signals, attributes, etc. or integrated by direct user interaction from an expert with knowledge about the system.

As one can observe, the method shows good results, most of the signals and other artifacts were detected correctly and completely. Having a list of artifact declarations in advance produces even more accurate predictions. Thus, our NLP-based approach shows a good quality and supports the generation of the formal requirement model to a significant extent.

A comparison of the time for its creation, both with and without the provided tool is not measured directly. However, from our experience of former and the presented use case it takes a lot of time for a requirements engineer to get into the description language for sequence diagrams by reading documentations and having discussions, to create the logical structure and to add all the details to the model manually. The new semi-automated approach supports the user in a great manner. It gives a first proposal of the requirement model in a textual and graphical view and provides options for handeling unclear points. This should therefore save a lot of time, even though a manual review of the created model is still required.

5.3. Model synthesis and test generation

For the next step, the requirement models of the charging approval system were used as the input for model synthesis using ifak's prototypical tool ModGen. Since the NLP-based approach treats every requirement as a separate entity, it is also necessary to connect each requirement by modelling the boundaries explicitly. As a result, a graph-based representation of the functionality as described by the requirements was generated in the form of a UML state machine (Fig. 4). The generated model contains 6 states and 20 transitions with appropriate signals, guards and actions. The semantic as well as syntactic validity of the generated UML state machine could

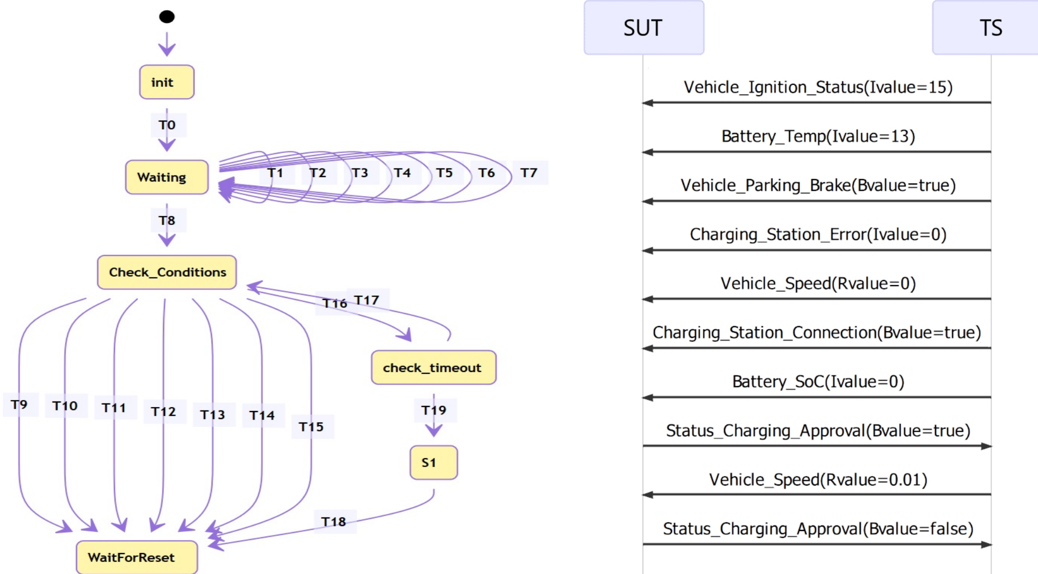

Figure 4: UML state machine of charging approval (left) and a test case visualized as a sequence diagram (right).

be confirmed by a thorough evaluation based on the initial requirement documents and by checking for deadlocks and livelocks. It could be shown that no further manual editing of the model is required for a full description of the behavior of the system.

In this evaluation, ifak's prototypical tool TCG with the coverage criteria "all-paths" was selected, which ensures that each possible path in the utilized model is covered by at least one test case. By utilizing the existing algorithm for test generation, a total of 73 test cases were generated. In Fig. 4, one of the generated test cases is visualized in the form of a sequence diagram. Here, a test system (TS) interacts with the SUT (charging approval) and provides a number of parameters, upon which the system decides if charging approval is given.

Overall, it can be shown that valid abstract test cases are generated based on the specification model. Using an appropriate framework for test case execution and a suitable test adapter, the generated test cases could be used for validation of the functional behavior of the SUT.

6. Conclusion

In this work, an NLP-based method for machine-aided model generation from textual requirements is presented. The method is designed to cover a wide range of requirements formulations without being restricted to a specific domain or format. Further, the generated requirement models are given in a user-friendly, comprehensible textual and graphical representation in the form of sequence diagrams.

We evaluated our approach on the industrial use case of a battery charging approval system and showed that the algorithm can produce complete, correct and consistent artifacts to a high degree. We have also shown how these artifacts are then used to create sequence diagrams

for each requirement and transformed into a state machine for the entire specification model to finally generate abstract test cases. With the proposed semi-automated approach, we aim to reduce the human effort of creating test cases from textual requirements to validating the generated requirement models. In future versions of this prototypical implementation, we intend to refine the rule-based approach further, thus reducing the need for manual modifications. One possible solution to this regard could be training a Named Entity Recognition (NER) algorithm to identify the semantic entities, however at the cost of intensive labelling work. Another solution could be to rely on (pre-trained) Semantic Role Labels (SRL).

This study is still research-in-progress, since even more complex textual requirements have to be considered for future applications. The use of the methodology is also conceivable in other domains, such as in rail, industrial communication and automotive. In future work, we therefore intend to analyze how we can improve the method to cover more application domains.

Acknowledgments

This research was funded by the German Federal Ministry of Education and Research (BMBF) within the ITEA 3 projects TESTOMAT under grant no. 01IS17026G and XIVT under grant no. 01IS18059E. We thank our former colleague Martin Reider and our research assistant Libin Kutty from ifak for the valuable contributions to this paper. We also thank AKKA Germany GmbH for providing an industrial use case for the evaluation of the presented method.

References

[1] P. Ammann, J. Offutt, Introduction to Software Testing, 2nd ed., Cambridge University Press, 2017.

[2] M. J. Escalona, J. J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres, F. J. Domínguez, An overview on test generation from functional requirements, Journal of Systems and Software 84 (2011) 1379–1393.

[3] I. Ahsan, W. H. Butt, M. A. Ahmed, M. W. Anwar, A comprehensive investigation of natural language processing techniques and tools to generate automated test cases, in: ICC, 2017, pp. 1–10.

[4] V. Garousi, S. Bauer, M. Felderer, NLP-assisted software testing: A systematic mapping of the literature, Information and Software Technology 126 (2020).

[5] M. Riebisch, M. Hubner, Traceability-Driven Model Refinement for Test Case Generation, in: ECBS, 2005, pp. 113–120.

[6] C. Rupp, Requirements-Engineering und -Management: Aus der Praxis von klassisch bis agil, 6th ed., Hanser, 2014.

[7] A. Mavin, P. Wilkinson, A. Harwood, M. Novak, Easy Approach to Requirements Syntax (EARS), in: RE, 2009, pp. 317–322.

[8] G. Carvalho, F. Barros, A. Carvalho, A. Cavalcanti, A. Mota, A. Sampaio, NAT2TEST Tool: From Natural Language Requirements to Test Cases Based on CSP, in: SEFM, 2015, pp. 283–290.

[9] S. C. Allala, J. P. Sotomayor, D. Santiago, T. M. King, P. J. Clarke, Towards Transforming User Requirements to Test Cases Using MDE and NLP, in: COMPSAC, 2019, pp. 350–355.
[10] C. Nebut, F. Fleurey, Y. Le Traon, J.-M. Jezequel, Automatic test generation: A use case driven approach, IEEE Transactions on Software Engineering 32 (2006) 140–155.
[11] A. Goffi, A. Gorla, M. D. Ernst, M. Pezzè, Automatic generation of oracles for exceptional behaviors, in: ISSTA, 2016, pp. 213–224.
[12] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, S. D. Castellanos, Translating code comments to procedure specifications, in: ISSTA, 2018, pp. 242–253.
[13] C. Wang, F. Pastore, A. Goknil, L. Briand, Automatic Generation of Acceptance Test Cases from Use Case Specifications: an NLP-based Approach, IEEE Transactions on Software Engineering (2020) 1–38.
[14] T. Yue, S. Ali, M. Zhang, RTCM: a natural language based, automated, and practical test case generation framework, in: ISSTA, 2015, pp. 397–408.
[15] B. C. F. Silva, G. Carvalho, A. Sampaio, Test Case Generation from Natural Language Requirements Using CPN Simulation, in: SBMF, 2015, pp. 178–193.
[16] J. Fischbach, A. Vogelsang, D. Spies, A. Wehrle, M. Junker, D. Freudenstein, Specmate: Automated creation of test cases from acceptance criteria, in: ICST, 2020, pp. 321–331.
[17] spaCy, Industrial-strength Natural Language Processing in Python, 2020. URL: https://spacy.io/.
[18] H. Yang, A. de Roeck, V. Gervasi, A. Willis, B. Nuseibeh, Analysing anaphoric ambiguity in natural language requirements, Requirements Engineering 16 (2011) 163–189.
[19] S. Lappin, H. J. Leass, An algorithm for pronominal anaphora resolution, Computational Linguistics 20 (1994) 535–561.
[20] L. Qiu, M.-Y. Kan, T.-S. Chua, A Public Reference Implementation of the RAP Anaphora Resolution Algorithm, in: LREC, 2004, pp. 291–294.
[21] D. K. Deeptimahanti, R. Sanyal, An Innovative Approach for Generating Static UML Models from Natural Language Requirements, in: ASEA, 2008, pp. 147–163.
[22] Roget's Hyperlinked Thesaurus, Categories of notions, 2020. URL: http://www.roget.org/scripts/hier.php/?class=I&division=0§ion=III.
[23] S. Magnus, T. Ruß, J. Krause, C. Diedrich, Modellsynthese für die Testfallgenerierung sowie Testdurchführung unter Nutzung von Methoden zur Netzwerkanalyse, at - Automatisierungstechnik 65 (2017) 73–86.
[24] J. Krause, Testfallgenerierung aus modellbasierten Systemspezifikationen auf der Basis von Petrinetzentfaltungen, Ph.D. thesis, Otto-von-Guericke-Universität Magdeburg, 2012.
[25] D. Grujic, T. Henning, E. J. C. García, A. Bergmann, Testing a Battery Management System via Criticality-based Rare Event Simulation, preprint, arXiv:2107.00530 [cs.SE], 2021.