Tracing of Informal and Formal Requirements through Model Variables
SKY 2015 Challenge

Martin Böschen and Christian Rudat
OFFIS - Institute for Information Technology, Escherweg 2, 26121 Oldenburg, Germany

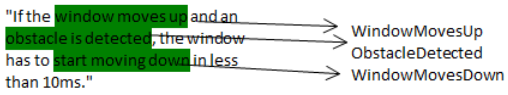
Keywords: Requirements Engineering, Traceability, Requirement Formalization.

Abstract: We describe a particular notion of traceability, namely the traceability between model variables and text passages in an informal requirement and discuss the usefulness of this concept, as it establishes a natural connection between formal and informal requirements. The traceability can be established in an semi-automated way by using the algorithm presented in this paper, which combines two metrics to match parts of the informal requirements to model variables. The first metric is purely text based and uses the Levenshtein distance, the second metric measures the semantical distance of concepts by using the Wu and Palmer measure and the WordNet ontology. The algorithm makes suggestions to the requirements engineer based on a combined score of these two measures. The suggested approach has been implemented in a tool and is studied in a small example.

1 INTRODUCTION

Traceability is a well-known concept in system engineering. It helps to manage the creation and changes of different assets within the development process. Safety critical requirements are often subject to a formal analysis and therefore need to be refined to a degree in which the formal semantic of a requirement is stated precisely. The ability to efficiently trace between the different levels of abstractions becomes even more important as standards like the ISO 26262 (International Standard Organization, 2011) require the industry to have a fully traceable development process. Requirement management tools like IBM DOORS (IBM, 2015) support these kinds of trace links between requirements. The CESAR project (Rajan and Wahl, 2013) described several requirement languages (varying in their level of formality) and the corresponding analysis techniques.

In this article, we focus on a specific notion of traceability, namely the traceability between text passages of natural language requirements to either variables of formalized versions of the requirement or variables of a model which shall implement the requirement. Technically, we assume requirements to be defined by a single or by multiple sentences written in natural language. Model variables are a list of variable names, defined in a system model. They can originate from a Matlab Simulink (Mathworks, 2015) model or from more abstract model, like an automaton. It is also possible that these variables are only defined within some ontological knowledge database. In any of these cases a clearly defined traceability is of importance. Figure 1 shows an example of a linked text passages to model variables. Below the natural language description of the requirement is the corresponding formal requirement, which describes the temporal-logic relationships between the model variables and can be processed by programs for testing or verification.



cyclic_P_implies_finally_Q_B__immediate
P: `$WindowMovesUp ∧ $ObstacleDetected`
Q: `$WindowMovesDown`
B: `10ms`

Figure 1: Example requirement (top) with traces to model variables. Formalization of the requirement (bottom) using a BTC ES pattern.

The requirement is taken of the motor driven power window system in one of the Matlab Simulink tutorials (Prabhu and Mosterman, 2004). The concept of traceability to variables has been implemented in

Böschen, M. and Rudat, C..
Tracing of Informal and Formal Requirements through Model Variables - SKY 2015 Challenge.
In *Proceedings of the 6th International Workshop on Software Knowledge (SKY 2015)*, pages 58-63
ISBN: 978-989-758-162-5

industrial tools, such as the Embedded Specifier (BTC Embedded Systems, 2015), where the manual identification and linking of variables to requirements is used as a first step to formalize a requirement. Several discussions (Sexton, 2013; Sebastian Siegl, 2014) emphasize the value of this step to structure the process of deriving formal requirements from informal ones.

A disadvantage of manually establishing traceability is that the process becomes increasingly error-prone with larger projects. The requirements engineer often has to process long lists of similar sounding words and has to be very carefully not to make mistakes, such as to link the wrong variables or create duplicates of variables.

Our approach support such a identification process by using a combination of different similarity measures between the model variables and the text passages, and to suggest the most similar terms to be linked with a trace. In this article, we measure the similarity by a textual difference (Levenshtein, 1966) as well as by a semantic similarity (Wu and Palmer, 1994), derived from the WordNet ontology (Miller, 1995). Knowledge stored in other kinds of ontologies could also be used to improve the results. Our goal is to support a requirements engineer by providing suggestions, since a fully automatic approach is unreasonable in most cases due to subtle differences in the meaning of natural language texts.

The theoretical background of our approach is elaborated in Section 2 and its implementation in Section 3. In Section 4 metrics to evaluate the provided suggestions are discussed.

2 ALGORITHM

This section describes our algorithm for generating suggestions for links to model variables. As a running example we use the following natural language requirement which was the basis for the formal requirement depicted in Figure 1:

Requirement: *"If the window moves up and an obstacle is detected, the window has to start moving down in less than 10ms."*

Model variables: `ObstacleObserved`, `PassengerDownSwitchIsPressed`, `WindowMovesDown`,`WinMovUp`

In this simple case, we would like to establish a trace between the text passage *"window moves up"* and the variable `WinMovUp`, a trace between the text passage *"obstacle is detected"* and the variable `ObstacleObserved`, as well as between the text passage *"window has to start moving down"* and `WindowMovesDown`. The variable `PassengerDownSwitchIsPressed` is not textually mentioned in this requirement and therefore shall not get a link. Instead of defining these traces manually, our algorithm generates suggestions, which can be accepted to establish the trace. Suggestions are proposed if a text passage within the requirement is similar to any of the variables. In this example, we illustrated several case which should be handled by the algorithm. The text passage *"obstacle is detected"* and the variable `ObstacleObserved` have the same meaning, but use a different verb. This semantic similarity should be perceived. The variable `WinMovUp` should be referenced to the text passage *"window moves up"*, although it is in an abbreviated variable style. The text passage *"window has to start moving down"* contains other information (*"has to start"*, but should be referenced to the model variable `WindowMovesDown`.

To measure the similarity illustrated in the examples above, our algorithm uses a textual similarity measure as well as a semantical similarity by using knowledge stored in an ontology. The top suggestions with their respective scores are finally presented to the engineer.

A suggestion is a pair of a text passage and a model variable, accompanied by a score. The algorithm for generating suggestions for a given requirement is outlined in Algorithm 1.

Algorithm 1: Generate Suggestion List for Requirement.

```
 1: function GETSUGGESTIONS(text)
 2:     for each passage ∈ text do
 3:         for each var ∈ model do
 4:             score ← SIMILARITY(passage, var)
 5:             list.append((score,passage,var))
 6:         end for
 7:     end for
 8:     SORT(list)
 9:     for each suggestion ∈ list do
10:         if NOOVERLAP(suggestion, sugList) then
11:             sugList ← suggestion
12:         end if
13:     end for
14:     sugList ← sugList[0..maxSuggestions]
15:     return sugList
16: end function
```

Every possible combination of text passages and variables is checked and rated with a score (line 4). We don't suggest overlapping text passages to avoid cluttering the suggestion list with slight modifications of the same text passage. Therefore, after identification of the individual scores, the list is sorted by de-

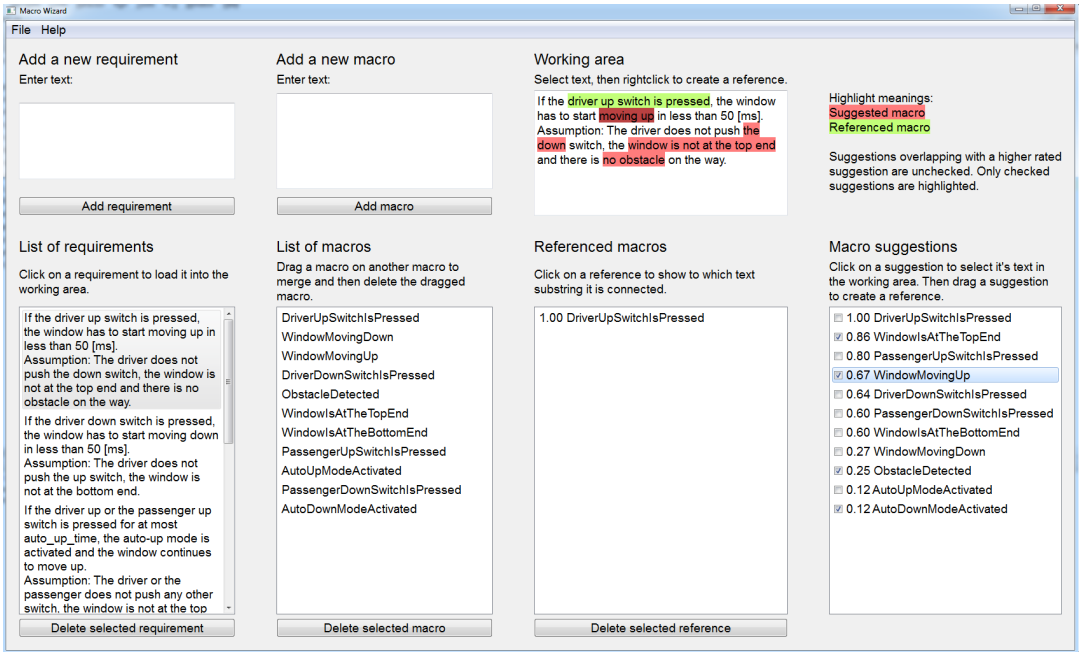


Figure 2: Screen shot of the prototype implementation.

scending score (line 8) and only the text passage with the best score will be used as a suggestion (line 11). We further limit the number of suggestions to a maximal value (line 14).

The similarity score is calculated based on a combination of two different similarity measures: The first score $score_{lvs}$ is a textual similarity measure called the Levenshtein (Levenshtein, 1966) distance. It is a very common score to measure how similar two strings are and is available in many programming libraries. The idea is to measure the number of character edits needed to transform one string into the other. We use the Levenshtein distance of the whole text passage and the model variable as our first score.

Our second score $score_{wpd}$is a semantic similarity measure, called the Wu-Palmer (Wu and Palmer, 1994) similarity measure. It measures the distance in terms of the number of semantic links necessary to reach the closest common ancestor of two concepts in an ontology. In this scenario, we used the semantic definition and relationship of concepts within the WordNet Ontology (Miller, 1995), which is a general purpose ontology of the English language. This measure does not work on arbitrary strings but on ontological concepts. Therefore it can not be directly applied to the text passage and the variable names. They need to be split into identified WordNet concepts, before the Wu-Palmer similarity measure can be applied. Therefore, the algorithm splits the text passages and the variable names into tokens and tries to match them to concepts defined in WordNet. The function *hit* for a text passage token is defined as:

$$hit(token, variable) = \begin{cases} 1 \text{ if } \exists t \in \mathcal{T}_{\text{variable}} : \\ \quad wpd(token, t) < treshold \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

,where $\mathcal{T}_{variable}$ is the set of the tokens extracted from the variable name, $wpd(\cdot)$ is the Wu-Palmer distance, and *treshold* is a constant. The score of this measure is then defined as:

$$score_{wpd}(\text{textpassage}, \text{variable}) = \frac{\sum hit(token)}{\sqrt{|\text{text passage}|}} \tag{2}$$

,where the sum in the numerator is computed over all tokens extracted from the text passage. In this paper, we assume that the names of the model variables are written using a CamelCase convention, so that they can be split into individual tokens. The square root in the denominator has the effect, that the score prefers further token hits than a shorter text passage.

The combined suggestion score is a logistic regression classifier. The two values $score_{lvs}$ and $score_{pwd}$ are used as features of the classifier. The model is learned on a set of training requirements, where the traces have been drawn manually. The learning is done offline and the identified model is used as combined score of the algorithm. The final score estimate is normalized to be within the interval $[0,1]$, such that they can be easily compared by the requirements engineer.

3 IMPLEMENTATION

We implemented the methodology and the algorithm described above within a prototypical tool. Currently, it is a stand-alone tool independent of commercial solutions, but we also cover integration scenarios into commercials solutions within this section. Figure 2 depicts a screen shot of the user interface.

The figure shows the natural language requirements on the far left and the model variables right next to it. The established traces are shown on the right side, together with the generated suggestions for traces. The currently selected requirement is shown within the text field working area. Already established traces and suggestion for traces are marked by a background color and suggestions can be accepted by drag-and-drop into the traces section. This organization of the GUI was chosen to give the engineer an overview over the current state of the data and suggestions, and facilitate the interaction.

The tool was implemented in Python using the NLTK[1] library for natural language processing. For the machine learing step the scikit-learn[2] library was used. The graphical user interface was implemented using the Qt framework[3].

For the methodology to be applicable in an industry process, it has to be integrated into their tool landscape. In the following, we sketch a possible scenario. We start by identifying which data has to be stored and which functionality must be provided. There are three different data artifacts, which must be stored:

- Natural Language Requirements
- Model Variables
- Traces

Furthermore, the following functionality must be provided:

- Calculation of suggestions
- Presentation of suggestion
- Establishing traces

Figure 3 sketches an overview of a possible tool landscape which realizes these requirements. In this scenario, the Requirements are stored in a requirement Repository, for example in DOORS. To edit and analyze the requirements, one could use the Requirement Quality Suite from the ReUse Company (The Reuse Company, 2015). It is integrated with a

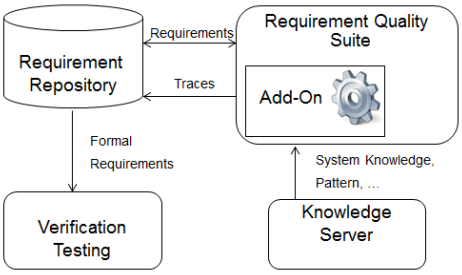


Figure 3: Possible Tool Landscape.

Knowledge Server, where one could define the structure of the formal requirements and the model variables. With its support, the requirements can be interpreted as patterns, and its formal structure can be accessed. The Requirement Quality Suite also allows to define add-ons with custom code, which can be used to implement our suggested algorithm. The results of this calculation can be shown using a web interface, which in turn could be used to create traces in the requirement repository by using an OSLC(OSLC Community, 2013) connection. Finally, the formal requirements can be subject for further analysis, such as verification and testing procedures. The full traceability to the natural language requirements to their refined formal requirements and even to the individual model variables can still be maintained.

4 EVALUATION METRIC

We propose an evaluation metric for our method. First, we need to make sure that the algorithm produces suggestions that have the same quality as traces drawn manually by an expert. This can be done quantitatively by defining an appropriate measure and compare traces established by an expert with suggestion made by the algorithm.

Before this metric is described, we show the data we used during development and testing. We made extensive use of a case study developed by MathWorks, where it is used as a tutorial for Matlab/Simulink. A variant of their requirements is also been used as a tutorial example for the Embedded Specifier, to show how requirements are formalized. The following list shows an excerpt of the requirements:

- If the driver up switch is pressed, the window has to start moving up in less than 50 [ms]. Assumption: The driver does not push the down switch, the window is not at the top end and there is no obstacle in the way.

[1] http://www.nltk.org
[2] http://scikit-learn.org
[3] http://www.qt.io

- If the driver down switch is pressed, the window has to start moving down in less than 50 [ms]. Assumption: The driver does not push the up switch, the window is not at the bottom end.
- If the passenger up switch is pressed while the window is at the top end, the passenger up request shall have no effect. Assumption: N/A
- .... and six more requirements.

The following model variables have been identified within theses requirements:

- DriverUpSwitchedIsPressed
- WindowMovingDown
- DriverDownSwitchIsPressed
- ObstacleDetected
- WindowIsAtTheTopEnd
- WindowIsAtTheBottomEnd
- PassengerUpSwitchIsPressed
- AutoUpModeActivated
- PassengerDownSwitchIsPressed
- AutoDownModeActivated

We produced several examples of sets of requirements (5 to 10 requirements and model variables) of similar complexity for our testing and evaluation purpose, for example a beverage vending machine, a traffic light or a home alarm system.

We suggest the following metric for measuring the quality of the suggestion. For a single requirement, let R be the set of references (which have been established manually by an expert) and let S be the ordered list of suggestions. Let S_n be the set of the best n suggestion. If $|R| = n$, then we calculate the quality score q of a requirements as followed:

$$q = \frac{|R \cap S_n|}{n} \tag{3}$$

The quality score is hence a number between 0 and 1. Note that we don't reduce the quality score if a suggestion is made for a different text passage, our evaluation shows that such an issue can be easily corrected by the user. We also don't consider the quality beyond the top suggestions. The formula expresses our experience, that the top suggestion must be useful for the user and small adjustments are acceptable. For our simple example we achieved very often a score above 50% and up to 100 %, but we deliberately don't publish any numbers here, as our example lacks the noise and the size of real-world examples. We plan to further evaluate our method with more realistic data sets in the near future.

As our method proposes a semi automatic approach, we also need to evaluate the user acceptance. This suggests a classical user experience research study. Within an experiment, different persons should establish the traceability on different set of requirements manually and with the help of our suggestion based method. The following measures should be used:

- Time to complete the task
- Perceived complexity of the task
- Accuracy achieved

5 CONCLUSION

We presented our algorithm and a prototypical implementation to establish traceability between requirements and model variables in a semi-automatic way. Our algorithm makes suggestions based on a similarity measure between development artifacts. For our approach, we got positive feedback from our colleagues and industrial partners, which encourages us to head further into the described direction. We would like to emphasize two main ideas, which are underlying our work:

- Model variables, which are explicitly stated in formal requirement and at least implicitly mentioned in informal requirements can be use to "glue" the requirement on different levels of abstraction.
- Traceability can be established in a semi-automatic, suggestion based process. An automatic process seems unfeasible due to the subtleness natural language can have, but a computer can support the engineer by pointing into the right direction.

In our future research, we plan to perform a more rigorous evaluation on real-world examples from industry and use more domain knowledge to improve the suggestion algorithm. For example, instead of only using the general purpose WordNet ontology, we could use domain specific ontologies which may lead to better results.

Instead of pure natural language requirement, one could start with requirement templates or patterns. In the CRYSTAL project we developed examples using requirements patterns (Génova et al., 2013). These requirements are more structured than natural language requirements, and their structure can be directly accessed by a computer. This closes the gap between the high-level requirements and the formulas used for verification and testing and could make it easier to draw the connection.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under Grant Agreement N°332830 (CRYSTAL) and German national funding from BMBF N°01*IS*13001*A*.

REFERENCES

BTC Embedded Systems (2015). BTC Embedded Specifier. http://www.btc-es.de/index.php?idcatside=52 (last visited 09/09/2015).

Génova, G., Fuentes, J. M., Llorens, J., Hurtado, O., and Moreno, V. (2013). A framework to measure and improve the quality of textual requirements. *Requirements Engineering*, 18(1):25–41.

IBM (2015). IBM Rational DOORS. http://www-03.ibm.com/software/products/en/ratidoor (last visited 09/09/2015).

International Standard Organization (2011). *Road Vehicles - Functional Safety*.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

Mathworks (2015). Simulink. http://de.mathworks.com/products/simulink (last visited 09/09/2015).

Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.

OSLC Community (2013). Open Services for Lifecycle Collaboration. http://open-services.net/.

Prabhu, S. M. and Mosterman, P. J. (2004). Model-based design of a power window system: Modeling, simulation and validation. In *Proceedings of IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics, Inc., Dearborn, MI*.

Rajan, A. and Wahl, T., editors (2013). *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Number 978-3709113868. Springer.

Sebastian Siegl, T. W. (2014). Von natrlichsprachlichen zu formalen anforderungen zwei werkzeuge im praxistest. *OBJEKTspektrum*.

Sexton, D. (2013). An outline workflow for practical formal verification from software requirements to object code. In *Formal Methods for Industrial Critical Systems*, pages 32–47. Springer.

The Reuse Company (2015). Requirements Quality Suite. http://www.reusecompany.com/requirements-quality-suite (last visited 05/27/2014).

Wu, Z. and Palmer, M. (1994). Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics.