Electronic Communications of the EASST
Volume 079 (2020)

Interactive Workshop
on the Industrial Application of Verification and Testing
ETAPS 2020 Workshop
(InterAVT 2020)

ReForm: A Tool for Rapid Requirements Formalization

Georgios Giantamidis, Georgios Papanikolaou, Marcelo Miranda,
Gonzalo Salinas-Hernando, Juan Valverde-Alcalá, Suresh Veluru,
Stylianos Basagiannis

8 pages

Guest Editors: Stylianos Basagiannis, Goetz Botterweck, Anila Mjeda
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

ReForm: A Tool for Rapid Requirements Formalization

Georgios Giantamidis[1,2], Georgios Papanikolaou[1], Marcelo Miranda[3], Gonzalo Salinas-Hernando[1], Juan Valverde-Alcalá[1], Suresh Veluru[4], Stylianos Basagiannis[1]

[1]United Technologies Research Centre Ireland, Ireland

[2]Aalto University, Finland

[3]University of Minho, Portugal

[4]ServisBOT Ltd. Arclabs Research Centre, Ireland

Abstract: Formal methods practices can sometimes be challenging to adopt in industrial environments. On the other hand, the need for formalization and verification in the design of complex systems is now more evident than ever. To the end of easing integration of formal methods in industrial model based system engineering workflows, UTRC Ireland has developed a tool aiming to render requirements formalization as effortless as possible to the industrial engineer. The developed approach is an end-to-end solution, starting with natural language requirements as input and going all the way down to auto-generated monitors in MATLAB / Simulink. We employ natural language processing and machine learning techniques for (semi-)automatic pattern extraction from requirements, which drastically reduces the required formalization workload for both legacy and new requirements. For monitor generation, we provide our own approach which outperforms existing state-of-the-art tools by orders of magnitude in some cases.

Keywords: Requirements Formalization, Formal Verification, Natural Language Processing, Clustering, Monitor Generation, MATLAB, Simulink

1 Introduction

Traditional requirements management procedure followed in industrial environments is characterized by a set of important problems that can significantly limit engineer productivity and in the worst case even have catastrophic results for the end product. One such problem is handling requirements in unstructured, natural language format which prevents early potential requirement inconsistency detection as well as analysis and tool support opportunities in general. Another issue is that typically test cases and requirement monitors are constructed manually, which is time consuming and error prone. While formalization of requirements could address these issues, it is often not performed as simply the vast volume of legacy requirements makes this prohibitively time consuming.

To address these problems at their core, in UTRC Ireland we have developed a tool for rapid requirements formalization and subsequent analysis. Our proposed approach, by employing

Natural Language Processing (NLP) and *Machine Learning* (ML) techniques for pattern identification, leads to huge acceleration in formalization for both legacy and new requirements. Once formalization is complete, consistency checking can be performed, which prevents early design error propagation. Formalized requirements also enable automatic monitor and test-case generation which rapidly accelerates the verification process. Overall, the proposed methodology can be shown to drastically reduce certification costs as well, an important consideration for industrial adoption of any technology.

In this report we outline the proposed solution, highlight important aspects that contribute to its effectiveness, present results in real world case-studies and conclude with discussion for future development goals. The presentation is structured as follows: In section 2 we outline the NLP part of the pipeline, which focuses on abstracting requirements to ease subsequent pattern identification. In section 3 we discuss grouping of abstract requirements into clusters for pattern identification and extraction. In section 4 we demonstrate how the extracted patterns speed-up the formalization process for both legacy and new requirements. In section 5 we talk about the additional capabilities of consistency checking and automatic monitor generation we provide for formalized requirements. In section 6 we present results of applying the proposed approach on real world industrial case studies. Finally, in section 7 we conclude with some ideas for future extension of the tool.

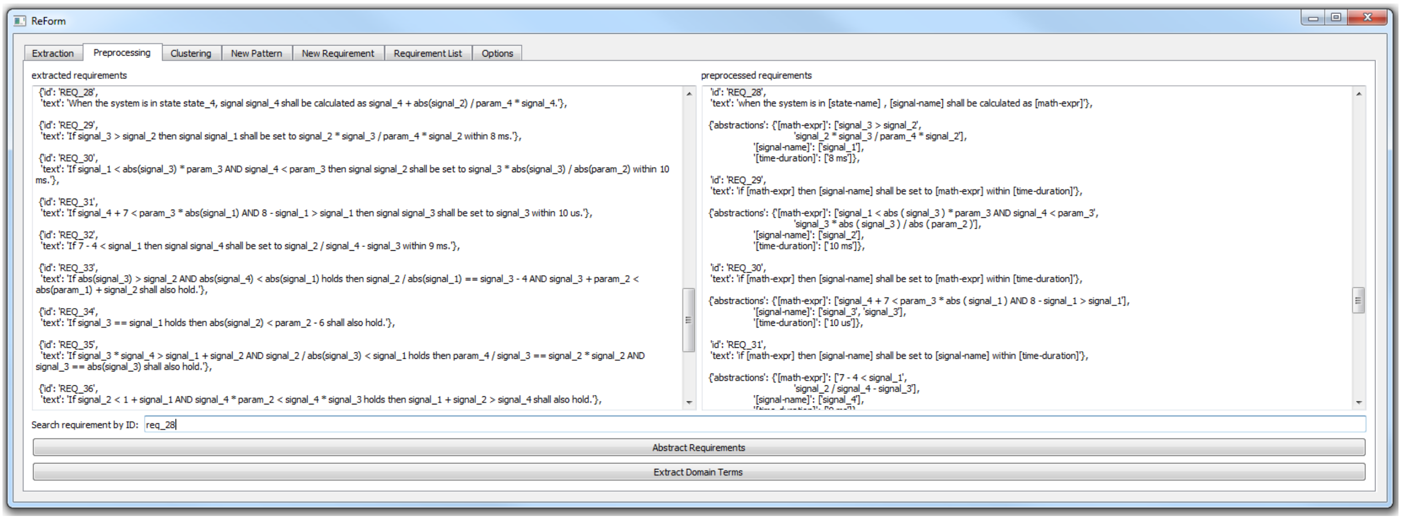
2 NLP for Domain Entity Identification

In order for pattern discovery to yield high quality results, it is imperative that irrelevant details are abstracted away from the provided input (Figure 1). To this end, we apply a range of built-in transformations to remove textual clutter that typically appears in a variety of domains, but we also provide the user with the option to insert their own abstraction rules in order to address the specific domain at hand.

The provided built-in transformations are regular expression and context free grammar based, and serve to identify and abstract away information such as signal names, quantities accompanied by units (e.g. time duration), as well as entire mathematical expressions. The user defined abstraction rules are based on NLP analysis of the requirement text. Specifically, we have developed our own heuristics for named entity recognition on top of off-the-shelf NLP libraries [Hon15, MSB$^+$14] and suggest these to the user so they can (potentially after tweaking) enable their use in the requirement abstraction procedure. Finally, the user is able to enrich the abstraction procedure by simply adding their own regular expression rules.

3 ML for Pattern Discovery

Once requirements are brought into an abstract representation, clustering for pattern discovery follows as the next step of the proposed solution (Figure 2). The aim of this step is to discover commonalities among requirements and group them based on these, so that the workload for



extracted requirements

```
 {'id': 'REQ_28',
  'text': 'When the system is in state state_4, signal signal_4 shall be calculated as signal_4 + abs(signal_2) / param_4 * signal_4.'},

 {'id': 'REQ_29',
  'text': 'If signal_3 > signal_2 then signal signal_1 shall be set to signal_2 * signal_3 / param_4 * signal_2 within 8 ms.'},

 {'id': 'REQ_30',
  'text': 'If signal_1 < abs(signal_3) * param_3 AND signal_4 < param_3 then signal signal_2 shall be set to signal_3 * abs(signal_3) / abs(param_2) within 10
ms.'},

 {'id': 'REQ_31',
  'text': 'If signal_4 + 7 < param_3 * abs(signal_1) AND 8 - signal_1 > signal_1 then signal signal_3 shall be set to signal_3 within 10 us.'},

 {'id': 'REQ_32',
  'text': 'If 7 - 4 < signal_1 then signal signal_4 shall be set to signal_2 / signal_4 - signal_3 within 9 ms.'},

 {'id': 'REQ_33',
  'text': 'If abs(signal_3) > signal_2 AND abs(signal_4) < abs(signal_1) holds then signal_2 / abs(signal_1) == signal_3 - 4 AND signal_3 + param_2 <
abs(param_1) + signal_2 shall also hold.'},

 {'id': 'REQ_34',
  'text': 'If signal_3 == signal_1 holds then abs(signal_2) < param_2 - 6 shall also hold.'},

 {'id': 'REQ_35',
  'text': 'If signal_3 * signal_4 > signal_1 + signal_2 AND signal_2 / abs(signal_3) < signal_1 holds then param_4 / signal_3 == signal_2 * signal_2 AND
signal_3 == abs(signal_3) shall also hold.'},

 {'id': 'REQ_36',
  'text': 'If signal_2 < 1 + signal_1 AND signal_4 * param_2 < signal_4 * signal_3 holds then signal_1 + signal_2 > signal_4 shall also hold.'},
```

preprocessed requirements

```
 'id': 'REQ_28',
 'text': 'when the system is in [state-name] , [signal-name] shall be calculated as [math-expr]'},

{'abstractions': {'[math-expr]': ['signal_3 > signal_2',
                                  'signal_2 * signal_3 / param_4 * signal_2'],
                  '[signal-name]': ['signal_1'],
                  '[time-duration]': ['8 ms']},

 'id': 'REQ_29',
 'text': 'if [math-expr] then [signal-name] shall be set to [math-expr] within [time-duration]'},

{'abstractions': {'[math-expr]': ['signal_1 < abs ( signal_3 ) * param_3 AND signal_4 < param_3',
                                  'signal_3 * abs ( signal_3 ) / abs ( param_2 )'],
                  '[signal-name]': ['signal_2'],
                  '[time-duration]': ['10 ms']},

 'id': 'REQ_30',
 'text': 'if [math-expr] then [signal-name] shall be set to [math-expr] within [time-duration]'},

{'abstractions': {'[math-expr]': ['signal_4 + 7 < param_3 * abs ( signal_1 ) AND 8 - signal_1 > signal_1'],
                  '[signal-name]': ['signal_3', 'signal_3'],
                  '[time-duration]': ['10 us']},

 'id': 'REQ_31',
 'text': 'if [math-expr] then [signal-name] shall be set to [signal-name] within [time-duration]'},

{'abstractions': {'[math-expr]': ['7 - 4 < signal_1',
                                  'signal_2 / signal_4 - signal_3'],
                  '[signal-name]': ['signal_4'],
```

Search requirement by ID: req_28

Figure 1: Abstraction view

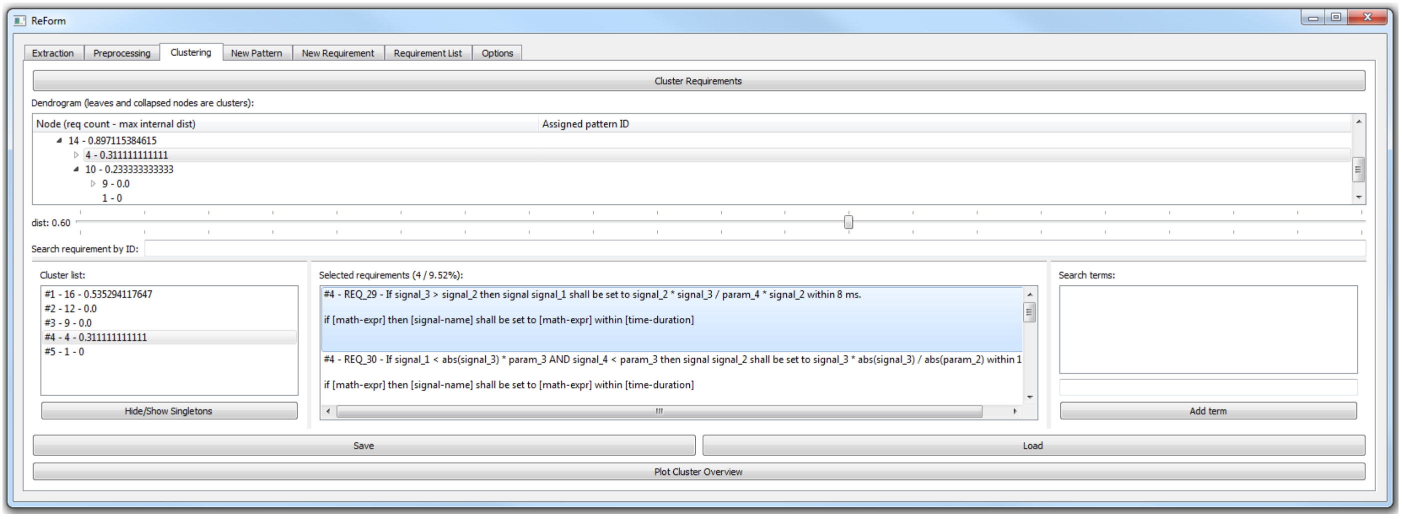
subsequent formalization is reduced (e.g. the user only has to formalize a couple dozen patterns instead of thousands of requirements).

Clustering requires defining what the distance between two requirements means. A variety of approaches have been explored here based on just syntactic information (e.g. Jaccard similarity of sentence n-grams), on just semantic information (e.g. similarity of dependency parsing results), as well as combinations of the two, along with additional heuristics.

The clustering algorithm we employ is a so called hierarchical clustering algorithm. Initially, each requirement is placed in its own group, and then an iterative phase takes place where, on each iteration, the groups that are closest to each other are merged together. This process continues until there is only one requirement group left that contains all requirements. As the algorithm executes, information about various requirement (sub)groups and their merges is kept track of and forms a binary tree data structure called a dendrogram. The final clustering is obtained by providing a distance threshold that slices the dendrogram at the corresponding height. An important property of the algorithm is that the resulting clusters are guaranteed to respect the given distance threshold, i.e. within each cluster, all pairwise requirement distances are less than that threshold. Note that while clustering results might not be perfect, the user (apart from adjusting the distance threshold) is allowed to intervene by manually expanding and collapsing dendrogram nodes in order to fine tune the obtained partitioning.

4 Formal Pattern Definition & Requirements Formalization

After partitioning of the requirements into clusters, pattern definition and formalization takes place (Figure 3). While this phase is manual, we argue that the preceding clustering significantly reduces the workload required by grouping together requirements that can potentially be



Dendrogram (leaves and collapsed nodes are clusters):

```
Node (req count - max internal dist)                Assigned pattern ID
  14 - 0.897115384615
    4 - 0.311111111111
    10 - 0.233333333333
      9 - 0.0
      1 - 0
```

dist: 0.60

Search requirement by ID:

Cluster list:

```
#1 - 16 - 0.535294117647
#2 - 12 - 0.0
#3 - 9 - 0.0
#4 - 4 - 0.311111111111
#5 - 1 - 0
```

Selected requirements (4 / 9.52%):

```
#4 - REQ_29 - If signal_3 > signal_2 then signal signal_1 shall be set to signal_2 * signal_3 / param_4 * signal_2 within 8 ms.

if [math-expr] then [signal-name] shall be set to [math-expr] within [time-duration]

#4 - REQ_30 - If signal_1 < abs(signal_3) * param_3 AND signal_4 < param_3 then signal signal_2 shall be set to signal_3 * abs(signal_3) / abs(param_2) within 1

if [math-expr] then [signal-name] shall be set to [math-expr] within [time-duration]
```

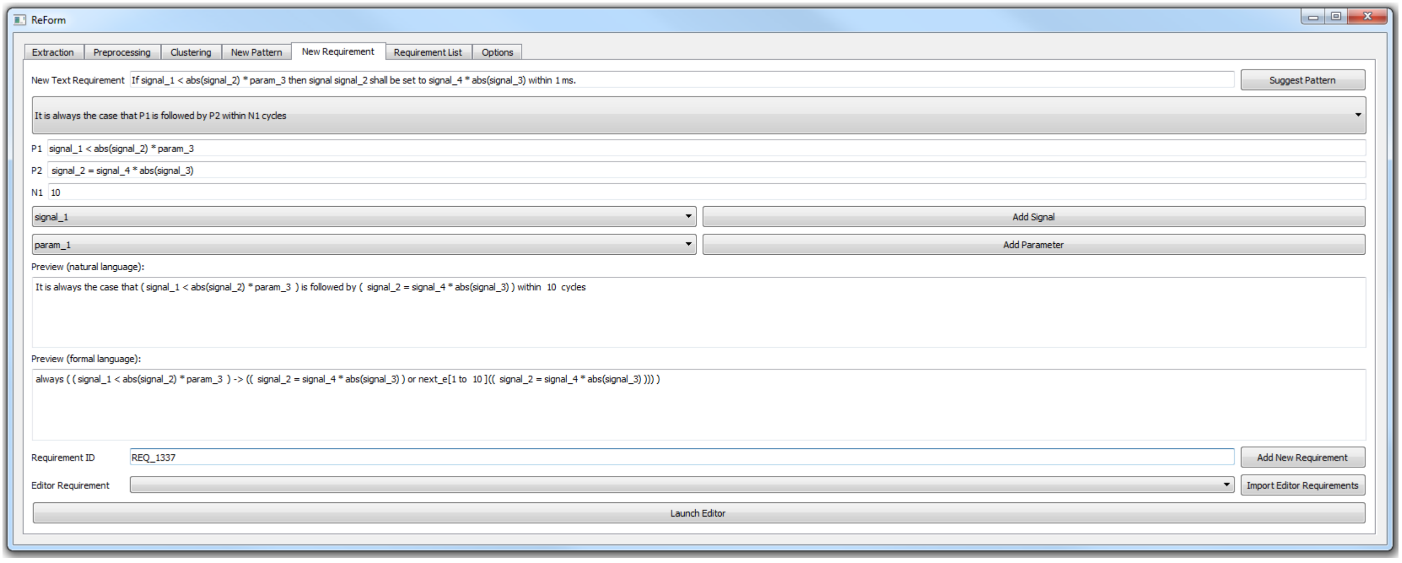
Search terms:

Figure 2: Clustering view

described by the same pattern. A pattern consists of two parts: a natural language part and a formal language part. This is done so that once the pattern is defined and a coupling between the two representations is made, the user can refer to the pattern by using the natural language part, while the tool can use the formal language part under the hood.

A variety of formal languages is supported for defining the formal part of a pattern, namely the *Property Specification Language* (PSL), an IEEE standard [PSL], the *SpeAR* domain specific language, developed by Rockwell Collins [FWH^{+}17], as well as the *Structured Assertion Language for Temporal Logic* (SALT) [BLS06]. All these languages share the characteristic that are extensions of *Linear Temporal Logic* (LTL), a widespread formalism for property specification towards verification purposes. However, in contrast to LTL, the languages we support are high-level, in the sense that they provide syntax features that facilitate expression of complex properties with minimal amount of text; which is another way our proposed approach considerably reduces the formalization workload for the user.

Given a set of patterns, formalization of legacy requirements is trivial – all the user needs to do is associate a cluster with a pattern and all requirements in that cluster are formalized according to the formal part of the corresponding pattern. For new requirements, we provide two ways formalization can be done: The user can either select a pattern to serve as a template and fill in the missing information (e.g. signal names, mathematical expressions etc.) according to the new requirement, or use the requirements editor we provide with syntax checking and autocompletion features based on a grammar automatically derived by the set of defined patterns. It might, of course, happen that none of the existing patterns are suitable for the new requirement at hand, in which case a new pattern has to be defined before proceeding. However, as more requirements are formalized using the tool and the pattern library grows, this situation becomes less likely to occur.



New Text Requirement: If signal_1 < abs(signal_2) * param_3 then signal signal_2 shall be set to signal_4 * abs(signal_3) within 1 ms.

It is always the case that P1 is followed by P2 within N1 cycles

P1: signal_1 < abs(signal_2) * param_3

P2: signal_2 = signal_4 * abs(signal_3)

N1: 10

signal_1

param_1

Preview (natural language):

It is always the case that (signal_1 < abs(signal_2) * param_3) is followed by (signal_2 = signal_4 * abs(signal_3)) within 10 cycles

Preview (formal language):

```
always ( ( signal_1 < abs(signal_2) * param_3  ) -> ((  signal_2 = signal_4 * abs(signal_3) ) or next_e[1 to  10 ]((  signal_2 = signal_4 * abs(signal_3) ))) )
```

Requirement ID: REQ_1337

Editor Requirement:

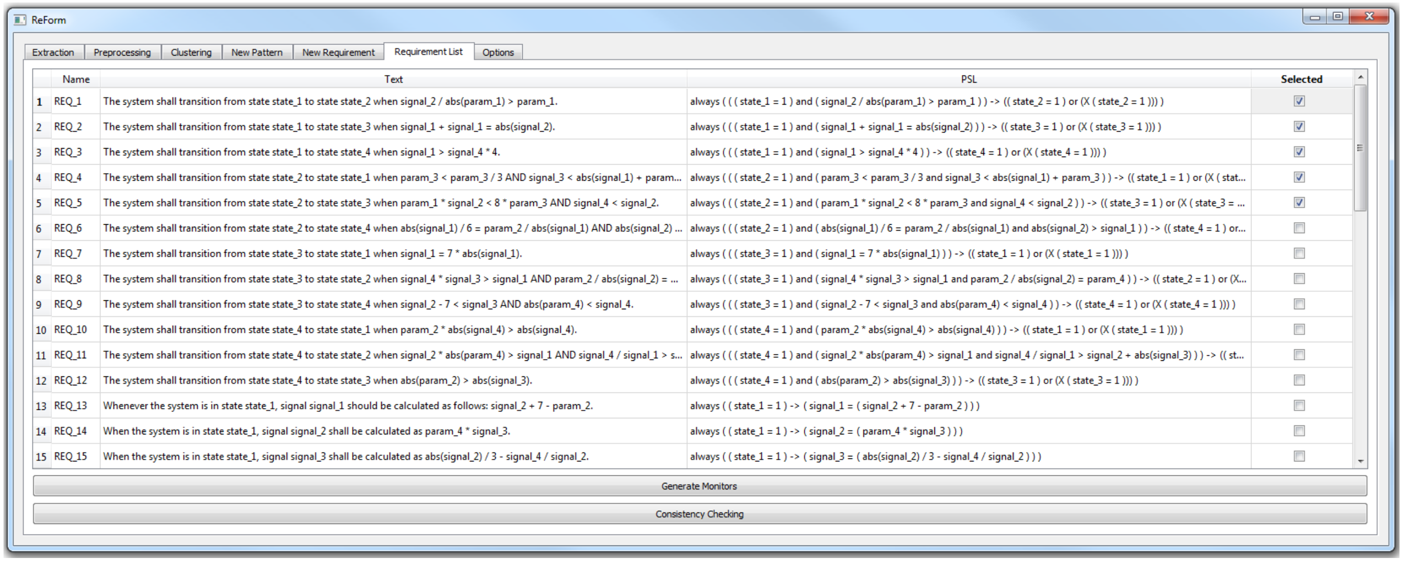
Figure 3: Formalization view

5 Consistency Checking & Monitor Generation

Having obtained a set of formalized requirements we can perform, through the tool, consistency checking as well as monitor generation (Figure 4). Note that both these procedures are supported for all available specification languages by means of translating the high-level property representations into low-level LTL formulas, the fundamental formalism our supported languages are based on. For consistency checking we extended an existing algorithm [GMR17] by integrating with the Z3 SMT solver [MB08] in order to be able to handle mathematical expressions as well. For monitor generation we use our own approach [GBT20] based on active automata learning [Ang87] which, in some cases, is able to outperform conventional translation strategies by orders of magnitude. The resulting monitors can be coupled with the system model (i) serving as assertions for run-time monitoring, (ii) for automatic test-case generation, as well as (iii) for formal verification of the design. Currently we only support Simulink as a monitor generation target, however, since we first generate a tool-agnostic intermediate representation, supporting additional targets is trivial.

6 Industrial Case Studies

The developed solution has been applied so far on two industrial case studies: (1) Low-level requirements for the FPGA specification of Airbus A350 ETRAC (electrical thrust reverser actuation controller), and (2) High-level requirements for the brake control unit of Mitsubishi Regional Jet. In the former case, the entire tool pipeline was used, from importing natural language



	Name	Text	PSL	Selected
1	REQ_1	The system shall transition from state state_1 to state state_2 when signal_2 / abs(param_1) > param_1.	always (((state_1 = 1) and (signal_2 / abs(param_1) > param_1)) -> ((state_2 = 1) or (X (state_2 = 1))))	✓
2	REQ_2	The system shall transition from state state_1 to state state_3 when signal_1 + signal_1 = abs(signal_2).	always (((state_1 = 1) and (signal_1 + signal_1 = abs(signal_2))) -> ((state_3 = 1) or (X (state_3 = 1))))	✓
3	REQ_3	The system shall transition from state state_1 to state state_4 when signal_1 > signal_4 * 4.	always (((state_1 = 1) and (signal_1 > signal_4 * 4)) -> ((state_4 = 1) or (X (state_4 = 1))))	✓
4	REQ_4	The system shall transition from state state_2 to state state_1 when param_3 < param_3 / 3 AND signal_3 < abs(signal_1) + param...	always (((state_2 = 1) and (param_3 < param_3 / 3 and signal_3 < abs(signal_1) + param_3)) -> ((state_1 = 1) or (X (stat...	✓
5	REQ_5	The system shall transition from state state_2 to state state_3 when param_1 * signal_2 < 8 * param_3 AND signal_4 < signal_2.	always (((state_2 = 1) and (param_1 * signal_2 < 8 * param_3 and signal_4 < signal_2)) -> ((state_3 = 1) or (X (state_3 = ...	✓
6	REQ_6	The system shall transition from state state_2 to state state_4 when abs(signal_1) / 6 = param_2 / abs(signal_1) AND abs(signal_2) ...	always (((state_2 = 1) and (abs(signal_1) / 6 = param_2 / abs(signal_1) and abs(signal_2) > signal_1)) -> ((state_4 = 1) or...	☐
7	REQ_7	The system shall transition from state state_3 to state state_1 when signal_1 = 7 * abs(signal_1).	always (((state_3 = 1) and (signal_1 = 7 * abs(signal_1))) -> ((state_1 = 1) or (X (state_1 = 1))))	☐
8	REQ_8	The system shall transition from state state_3 to state state_2 when signal_4 * signal_3 > signal_1 AND param_2 / abs(signal_2) = ...	always (((state_3 = 1) and (signal_4 * signal_3 > signal_1 and param_2 / abs(signal_2) = param_4)) -> ((state_2 = 1) or (X...	☐
9	REQ_9	The system shall transition from state state_3 to state state_4 when signal_2 - 7 < signal_3 AND abs(param_4) < signal_4.	always (((state_3 = 1) and (signal_2 - 7 < signal_3 and abs(param_4) < signal_4)) -> ((state_4 = 1) or (X (state_4 = 1))))	☐
10	REQ_10	The system shall transition from state state_4 to state state_1 when param_2 * abs(signal_4) > abs(signal_4).	always (((state_4 = 1) and (param_2 * abs(signal_4) > abs(signal_4))) -> ((state_1 = 1) or (X (state_1 = 1))))	☐
11	REQ_11	The system shall transition from state state_4 to state state_2 when signal_2 * abs(param_4) > signal_1 AND signal_4 / signal_1 > s...	always (((state_4 = 1) and (signal_2 * abs(param_4) > signal_1 and signal_4 / signal_1 > signal_2 + abs(signal_3))) -> ((st...	☐
12	REQ_12	The system shall transition from state state_4 to state state_3 when abs(param_2) > abs(signal_3).	always (((state_4 = 1) and (abs(param_2) > abs(signal_3))) -> ((state_3 = 1) or (X (state_3 = 1))))	☐
13	REQ_13	Whenever the system is in state state_1, signal signal_1 should be calculated as follows: signal_2 + 7 - param_2.	always ((state_1 = 1) -> (signal_1 = (signal_2 + 7 - param_2)))	☐
14	REQ_14	When the system is in state state_1, signal signal_2 shall be calculated as param_4 * signal_3.	always ((state_1 = 1) -> (signal_2 = (param_4 * signal_3)))	☐
15	REQ_15	When the system is in state state_1, signal signal_3 shall be calculated as abs(signal_2) / 3 - signal_4 / signal_2.	always ((state_1 = 1) -> (signal_3 = (abs(signal_2) / 3 - signal_4 / signal_2)))	☐

Figure 4: Consistency checking & monitor generation view

requirements all the way down to formal verification of the Space Vector Modulation (SVM) subsystem of the design using the automatically generated monitors. Specifically, we were able to fit 40% of the 750 given requirements into 25 clusters, and managed to formalize the 100 requirements for the SVM subsystem using just 6 patterns. Automatically generated monitors from these 100 requirements were coupled with the system model and successful formal verification of the design took place by employing the Simulink Design Verifier (SLDV) MATLAB toolbox. In the latter case, only the parsing and clustering part of the tool were exercised, in order to demonstrate that our solution provides benefits (e.g. better documentation and traceability by enabling easy subsequent mapping to a more structured representation) even for high-level requirements that cannot be easily mapped to Simulink model representations; in particular, we were able to fit 50% of the 700 given requirements into just 15 clusters.

7 Conclusion & Future Work

In this report we presented an effort carried out in UTRC Ireland to develop a tool for rapid requirements formalization. This is achieved by pattern identification from legacy requirements with NLP and ML methods, and subsequent use of the extracted patterns to drive formalization of both legacy and new requirements. A variety of formal languages is supported by the tool and, once requirements are formalized, consistency checking and automatic monitor generation can be performed as well. The approach has been tested on industrial case studies with several hundreds of requirements in each case and the results have been very promising so far.

One limitation of the developed solution is that it currently only focuses on functional requirements (i.e. system behavior). Therefore, a direction we plan to explore in the future is handling non-functional requirements as well (e.g. timing and architectural constraints). Another direction for future development is extending the tool with more specification languages and monitor generation targets in order to enable further interoperability with other tools and ease adoption from industrial users.

Bibliography

[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75(2):87–106, Nov. 1987.
doi:10.1016/0890-5401(87)90052-6

[BLS06] A. Bauer, M. Leucker, J. Streit. SALT—Structured Assertion Language for Temporal Logic. In Liu and He (eds.), *Formal Methods and Software Engineering*. Pp. 757–775. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[FWH$^+$17] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, J. A. Davis. SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In Barrett et al. (eds.), *NASA Formal Methods*. Pp. 420–426. Springer International Publishing, Cham, 2017.

[GBT20] G. Giantamidis, S. Basagiannis, S. Tripakis. Efficient Translation of Safety LTL to DFA Using Symbolic Automata Learning and Inductive Inference. In *Computer Safety, Reliability, and Security - 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 15-18, 2020, Proceedings*. Springer Nature Switzerland AG, 2020.
https://doi.org/10.1007/978-3-030-54549-9_8

[GMR17] N. Gigante, A. Montanari, M. Reynolds. A One-Pass Tree-Shaped Tableau for LTL+Past. In Eiter and Sands (eds.), *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*. EPiC Series in Computing 46, pp. 456–473. EasyChair, 2017.
http://www.easychair.org/publications/paper/340363

[Hon15] M. Honnibal. spaCy: Industrial-Strength Natural Language Processing. 2015.
https://spacy.io/

[MB08] L. de Moura, N. Bjørner. Z3: An Efficient SMT Solver. In Ramakrishnan and Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[MSB$^+$14] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. Pp. 55–60. 2014.
http://www.aclweb.org/anthology/P/P14/P14-5010

[PSL] IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL), in IEC 62531:2012(E) (IEEE Std 1850-2010), vol., no., pp.1-184, 28 June 2012.