Hanfor: Semantic Requirements Review at Scale

Samuel Becker, Daniel Dietsch, Nico Hauff, Elisabeth Henkel,
Vincent Langenfeld, Andreas Podelski and Bernd Westphal

Department of Computer Science, University of Freiburg, Germany

Abstract

[**Context & Motivation**] Formal analysis of requirements finds relevant problems (overlooked in reviews), but needs formal requirements. [**Question/Problem**] We address the problem of tool support for a semantical review of readily elicited requirements, that is based on formal analysis. [**Pricipal ideas/results**] Dedicated tool support for a semantical review process supports the delegation of the formalisation task to less experienced workers. [**Contribution**] We present Hanfor, a web based tool used to support formalisation in several industry projects. A video demonstration is available at struebli.informatik.uni-freiburg.de/refsq2021.

Keywords

Formal Requirements Analysis, Structured Requirements Review, Tool supported Review

1. Introduction

The specification of requirements is a critical activity in software and system development because the set of requirements can have effects on many later activities in the development process. Requirements influence the design and implementation, they define whether a product is acceptable or not during acceptance test, and they prescribe when the final software or system is adequate for its tasks. Properties of requirements specifications that increase or lower the risk for negative effects to later activities are long known and formulated, e.g., in form of ISO/IEC/IEEE standards. The IEEE standards 830 and 29148 [1, 2] name as *desired* properties, that is, properties that lower overall risks, the correctness, completeness, and consistency as well as readability and maintainability to recall only a few. The complementary *undesired* property inconsistency, for example, means that there are requirements that contradict each other so that the whole set of requirements is not realisable. If inconsistency is only recognised during the coding or implementation activity, substantial extra costs may be incurred. More recently, the weaker notion of rt-inconsistency has been proposed [3, 4, 5]. Rt-inconsistent requirements are principally realisable but there exist design decisions that inhibit the satisfaction of the requirements. An example for a complementary property to readability and maintainability is *vacuity*. A trivial example would be a requirement that occurs multiple times in a specification with slightly different wording but the same meaning.

In: F.B. Aydemir, C. Gralha, S. Abualhaija, T. Breaux, M. Daneva, N. Ernst, A. Ferrari, X. Franch, S. Ghanavati, E. Groen, R. Guizzardi, J. Guo, A. Herrmann, J. Horkoff, P. Mennig, E. Paja, A. Perini, N. Seyff, A. Susi, A. Vogelsang (eds.): Joint Proceedings of REFSQ-2021 Workshops, OpenRE, Posters and Tools Track, and Doctoral Symposium, Essen, Germany, 12-04-2021

henkele@cs.uni-freiburg.de (E. Henkel); langenfv@cs.uni-freiburg.de (V. Langenfeld)
0000-0002-8947-5373 (D. Dietsch); 0000-0002-6824-0567 (B. Westphal)

Well-known issues with the properties from IEEE 830 and 29148 are that they are characterised informally and that the standard documents do not provide a practical procedure to ensure the absence of undesired properties. Recent works have proposed formal characterisations of properties such as consistency and vacuity and provide automated analyses of sets of *formal* requirements for these properties (e.g., [6, 7, 8]). Still, one issue remains: Today's requirements are provided in natural language and the procedures named above need formal, mathematical requirements descriptions.

The Dietsch-Langenfeld-Process (DLP) [9] is a process model for a *semantic review* of requirements through their formalisation, hence addressing the latter issue. A DLP client (cf. Figure 1) provides a set of informally described requirements, the so-called *raw requirements*, and receives a report on semantic issues (including a formalisation of the raw requirements). The review is conducted by persons assuming supervisor and worker roles. The supervisor receives the raw requirements from the client, oversees the formalisation and analysis of requirements done by the workers, and delivers the report on semantic issues to the client. A worker is assigned a set of requirements and for each requirement proposes a formalisation (using a pattern language such as [10]). If all stakeholders agree on this formalisation, the formal analysis backend(s) are applied to the coded requirements.

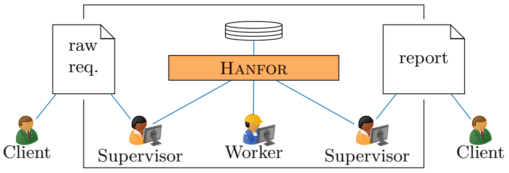


Figure 1: Dietsch-Langenfeld-Process.

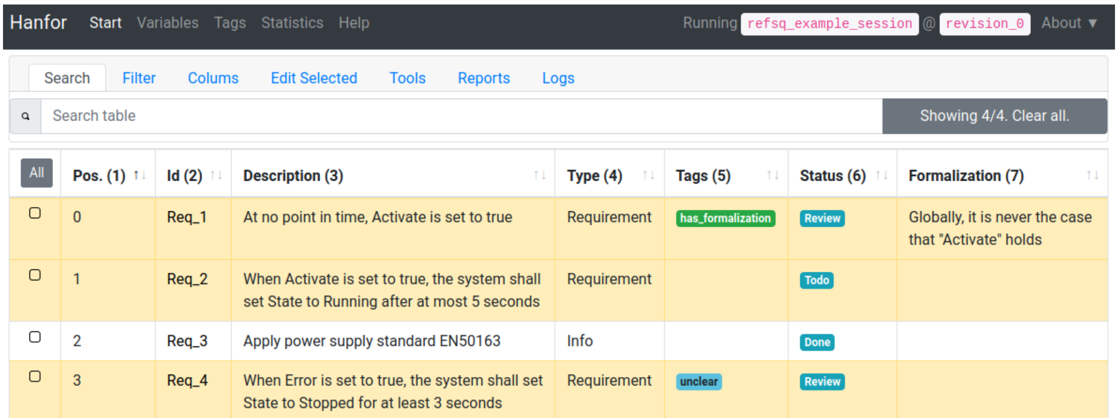
In this paper, we describe Hanfor, a web-based tool that supports the supervisor and worker role in the DLP. A main design goal of Hanfor was to ease the formalisation of large requirements sets by people who are particularly trained for their work (rather than addressing the casual user). The tool consists of a light-weight selection of requirements management features, a formalisation editor, and a powerful analysis back-end.

2. Workflow and Interaction with Hanfor

Following Figure 1, this section describes how Hanfor supports the DLP.

Pre-Processing of Raw Requirements. In the DLP context, a *raw requirement* consists of a unique *identifier*, a textual description, and a *type* as in `(Req_3, Apply power supply standard EN50163, Info)`. Form of identifiers and types are not constrained by Hanfor to support any type conventions on the client's side. The raw requirements undergo a first sight check by the supervisor supported by the client where both agree on a set of types and their meaning and obvious issues are resolved. Type 'Info' may, e.g., label requirements that are included in the requirements document but should not be subject to formalisation.

Formalisation. In DLP, the formalisation of the pre-processed raw requirements is conducted by workers. Hanfor's main page (see Figure 2) is the central workspace for them. The workspace is a tabular view on requirements, where the raw requirement triples are shown in columns 'Id', 'Description', and 'Type' ('Pos' is added as a unique Hanfor identifier to retain the original ordering).



All	Pos. (1)	Id (2)	Description (3)	Type (4)	Tags (5)	Status (6)	Formalization (7)
□	0	Req_1	At no point in time, Activate is set to true	Requirement	has_formalization	Review	Globally, it is never the case that "Activate" holds
□	1	Req_2	When Activate is set to true, the system shall set State to Running after at most 5 seconds	Requirement		Todo	
□	2	Req_3	Apply power supply standard EN50163	Info		Done	
□	3	Req_4	When Error is set to true, the system shall set State to Stopped for at least 3 seconds	Requirement	unclear	Review	

Figure 2: HANFOR starting page with tabular view of imported requirements.

The DLP assigns each requirement a *status* (cf. Figure 3). Ideally, each requirement reaches the status 'Done', indicating that the requirement has been formalised and its formalization is agreed on. Initially, all requirements have status 'Todo' and may alternate between 'Todo' and 'Review'. Workers try to propose a formalisation and change the status to 'Review'. If there are issues during formalisation (e.g. unclear mapping to variables, or properties not expressible in the considered pattern language), the worker uses HANFOR *tags* to document the issue (e.g., row 3 in Figure 2). Requirements with status 'Review' and issue tags are examined by the supervisor and, if resolved, their status changes back to 'Todo'; without issue tags they are scheduled for the formal analysis (like row 0 in Figure 2). That is, intuitively, the formalisation activity is about transforming rows like 1 and 3 in Figure 2 to the appearance of the other rows by filling in 'Tags', 'Status', and 'Formalisation'.

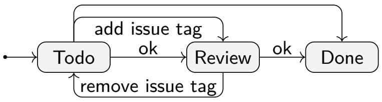
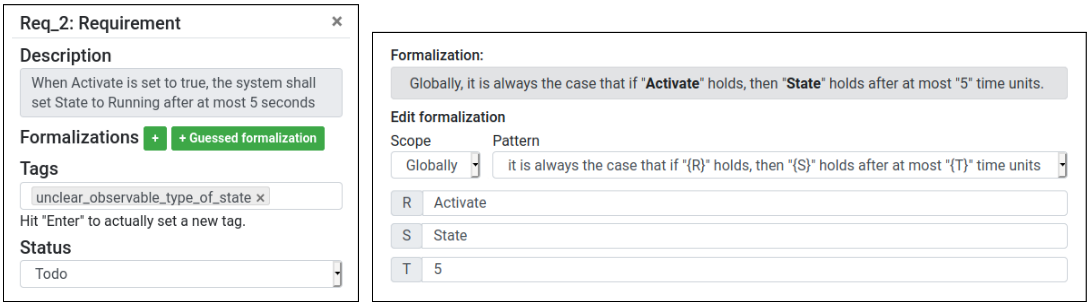


Figure 3: Requirement status model.

Pattern Editor. Formalisation (or coding) of a raw requirement with a pattern catalogue means to understand the constraints that the raw requirement expresses, to choose the corresponding pattern, and to provide values for the pattern's parameters such as expressions over variables or durations. Consider requirement *Req_2* for example. It states a constraint on the observation of particular values for (boolean) variable *Activate* and (enumeration type) variable *State* with a time bound. The matching pattern is a *bounded response* (S responds to R) with upper bound T. To formalise *Req_2*, one opens its editor by clicking on its 'Id'. Figure 4a shows the initial editor form with the description repeated. To add a formalisation (some requirements need more than one), one clicks on one of the green '+'-buttons, which open the form shown in Figure 4b. Here, the pattern is instantiated with blanks for the values R, S, and T below[1].

Analysis. Once a set of requirements has a formalisation and no issue tags, it can be passed to the formal analysis back-end REQCHECK [6], which checks for inconsistency, rt-inconsistency, and vacuity. If any of these undesired properties are detected, REQCHECK provides diagnostic input in form of a minimal core set of inconsistent requirements,

[1] Pattern in [3, 4, 5] also support*scopes* e.g., apply constraints only between two time points P and Q.



Req_2: Requirement ×

Description

When Activate is set to true, the system shall set State to Running after at most 5 seconds

Formalizations + + Guessed formalization

Tags

unclear_observable_type_of_state ×

Hit "Enter" to actually set a new tag.

Status

Todo

Formalization:

Globally, it is always the case that if "**Activate**" holds, then "**State**" holds after at most "5" time units.

Edit formalization

Scope: Globally

Pattern: it is always the case that if "{R}" holds, then "{S}" holds after at most "{T}" time units

R: Activate

S: State

T: 5

(a) Initial view. **(b)** Pattern instantiation dialogue.

Figure 4: Requirements editor.

a timing diagram, and the set of vacuous requirements, respectively. The results are considered by the supervisor and presented to the client (cf. Figure 1) in a proper document including raw requirements, formalisation, and findings. If the client is able to resolve some of the issues, the process may be iterated.

Hanfor Support for Large Requirements Sets. One design goal for Hanfor was to support workers in processing large sets of requirements (10s or 100s) conveniently, efficiently, and effectively. To this end, Hanfor offers a rich selection of efficient tools to navigate the table (sort by column, search whole table or selected columns (partial and exact match), filter by status, tag, type, etc.). Different actions can be applied to batches of requirements, e.g., to change the status of all requirements of type ‘Info’ to ‘Done’.

The pattern editor provides auto-completion, e.g., for variables and tags, on-the-fly creation of variables (including type inference), and can suggest a pattern based on matching the raw requirements (most useful if the client side states requirements in some standardised grammar). Note that suggestions are supposed to *support* the human workers rather than replace them as the DLP is about human natural language processing, not automatic NLP. Already in the editor, light-weight checks such as type checking are applied. Existing variables can be searched, sorted, and imported (tab ‘Variables’), and tags can be searched, sorted, and customised (name, colour, description; tab ‘Tags’). In addition, the Hanfor repository is fully versioned (top right corner of Figure 2).

3. Architecture and Inner Workings

Hanfor is a web application built with Python 3, the Flask web application toolkit, and JavaScript. Fig. 5 shows the architecture of Hanfor and its companion tool Ultimate ReqCheck[2]. Each Hanfor session stores raw requirements provided as `.csv` file for each revision together with specific attributes, e.g., the tags associated with a requirement or the formalisation, in flat files on disk. This session storage also contains meta information about tags, and all typed variables and their constraints. In a review session, a ‘Worker’ selects raw requirements, adds tags, and formalises it by a pattern. The pattern is selected manually or picked from the recommendations of the guesser component. Instantiation of patterns (with expressions over variables) is guarded by the syntax and type checker

[2] Both tools are available at ultimate-pa.github.io/hanfor under the LGPLv3 open source license.

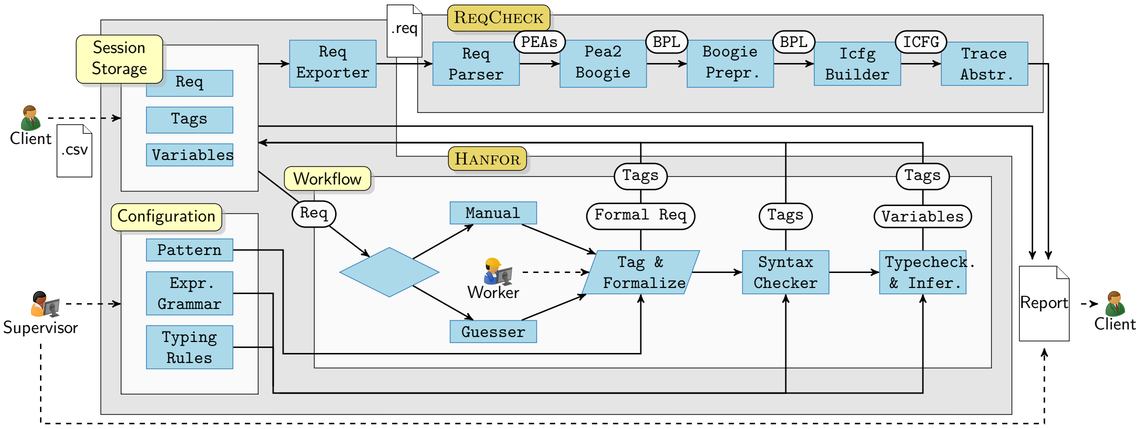


Figure 5: Hanfor and Ultimate ReqCheck architecture. (Solid lines describe the flow of internal artefacts (white rounded boxes) between components (blue boxes) in the tools (gray); dashed lines represent provision of external artefacts or decisions (to diamonds); trapezoids are manual activities.)

components that inform the worker about possible errors. The type inference component infers types for new variables from their usage. If a new version of the raw requirements is submitted by the client, Hanfor automatically identifies changed, added, or removed requirements, changes their status and tags them for re-review.

A set of formalised requirements can be exported for analysis by ReqCheck. Ultimate ReqCheck is a software model checking tool that extends Ultimate Automizer [11] with two components: `ReqParser` (cf. Fig. 5) parses formalised requirements and transforms them via Duration Calculus [12] to Phase Event Automata (PEA) [6] and `Pea2Boogie` transforms a network of PEAs into a Boogie [13] program, which encodes requirement properties as program analysis task [6]. A modified version of Automizer (represented by the remaining three components in Fig. 5) verifies the resulting Boogie program and performs post-processing steps to isolate reasons for possible property violations, which are then collated into a report. Through Hanfor's configuration, the supervisor can add or remove patterns, types, and extend the expression language (which is a subset of Boogie expressions) with new functions or operators.

4. Conclusion

We have presented Hanfor, a web-based tool for requirements formalisation and semantic analysis. A novelty of Hanfor is its co-development with a defined review process, the DLP [9]. By its architecture, Hanfor supports research into the expressiveness and applicability of pattern languages, into the ergonomics of requirements formalisation at large(r) scale, and tool-based semantic review of requirements as a service, which is the very goal of the DLP.

We have used (and continuously improved) Hanfor on industrial requirements sets of between 20 and 1000 raw requirements from the automotive and railway domain (cf. [6]). The worker role has been assumed by different workers (including PhD and graduate students with different previous knowledge) and a post-doc as supervisor. In the experiments, the tabular representation and the powerful filtering and sorting features were found adequate. Most formalisations can be done on a requirement by requirement basis, and the ones that are harder to formalise often benefit from efficient access to the neighbouring requirements and meta information. Hanfor features such as auto-completion, and type inference and checking enable effective and efficient formalisation: Auto-completion allows the workers to focus on the formalisation at hand, and type inference suggests many variable types after only few requirements have been formalised and thus gives an additional consistency check with immediate feedback to the worker. These light-weight analyses also prevent roadblocks due to careless mistakes in the subsequent formal analysis. Overall we learned, that, with a tool supporting the process an preventing careless mistakes, the formalisation of a set of requirements can be done by workers with a basic understanding of requirements engineering and the requirements patterns. Our clients were fond of the information supplied by the reports aggregated from the tags applied during the formalisation process as well as the analysis results.

Acknowledgments

Author B. Westphal was supported by the DFG under reference no. WE 6198/1-1.

References

[1] IEEE, Recomm. Practice for Software Requirements Specifications, 1998. 830:1998.

[2] IEEE, Systems and software eng. — Requirements engineering, 2018. 29148:2018.

[3] A. Post, J. Hoenicke, A. Podelski, Vacuous real-time requirements, in: RE, IEEE, 2011, pp. 153–162.

[4] A. Post, J. Hoenicke, A. Podelski, rt-inconsistency: A new property for real-time requirements, in: FASE, volume 6603 of *LNCS*, Springer, 2011, pp. 34–49.

[5] A. Post, J. Hoenicke, Formalization and analysis of real-time requirements, in: VSTTE, volume 7152 of *LNCS*, Springer, 2012, pp. 225–240.

[6] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, A. Post, Scalable analysis of real-time requirements, in: RE, IEEE, 2019, pp. 234–244.

[7] A. Moitra, K. Siu, A. W. Crapo, et al., Towards development of complete and conflict-free requirements, in: RE, IEEE, 2018, pp. 286–296.

[8] A. W. Fifarek, et al., SpeAR v2.0: Formalized past LTL specification and analysis of requirements, in: NFM, volume 10227 of *LNCS*, 2017, pp. 420–426.

[9] D. Dietsch, V. Langenfeld, B. Westphal, Formal requirements in an informal world, in: FORMREQ, IEEE, 2020, pp. 14–20.

[10] A. Post, I. Menzel, A. Podelski, Applying restricted english grammar on automotive requirements — does it work?, in: REFSQ, 2011, pp. 166—180.

[11] M. Heizmann, Y. Chen, et al., Ultimate automizer and the search for perfect interpolants, in: TACAS (2), volume 10806 of *LNCS*, Springer, 2018, pp. 447–451.

[12] Z. Chaochen, M. R. Hansen, Duration Calculus, MTCS, Springer, 2004.

[13] K. R. M. Leino, This is Boogie 2, Manuscript KRML 178 (2008).