Poster: Automatic Consistency Checking of Requirements with ReqV

Simone Vuotto[*], Massimo Narizzano[†], Luca Pulina[*], Armando Tacchella[†]
[*]Chemistry and Farmacy Dept., University of Sassari, Italy
[†]DIBRIS, University of Genoa, Italy
{svuotto,lpulina}@uniss.it, {massimo.narizzano,armando.tacchella}@unige.it

***Abstract*—In the context of Requirements Engineering, checking the consistency of functional requirements is an important and still mostly open problem. In case of requirements written in natural language, the corresponding manual review is time consuming and error prone. On the other hand, automated consistency checking most often requires overburdening formalizations. In this paper we introduce REQV, a tool for formal consistency checking of requirements. The main goal of the tool is to provide an easy-to-use environment for the verification of requirements in Cyber-Physical Systems (CPS). REQV takes as input a set of requirements expressed in a structured natural language, translates them in a formal language and it checks their inner consistency. In case of failure, REQV can also extracts a minimal set of conflicting requirements to help designers in correcting the specification.**

***Index Terms*—Requirements Engineering, Verification, Consistency, CPS**

I. Introduction

In the context of safety- and security-critical Cyber-Physical Systems (CPSs), checking the sanity of functional requirements is an important, yet challenging task. While it is largely recognized that a flaw in the requirements specification can lead to delays, additional expenses and, possibly, the failure of the project, the knife-edge between increasing automation, at the expense of increasing formalization cost, and increasing usability, at the expense of increasing the review effort, is still difficult to walk in most cases. Given the growing demand for complex, safe and secure CPSs, and the need to reduce time-to-market and costs, usable solutions to speed-up the review process are in order. Formal methods provide a viable solution, but they require precise specifications and a high degree of expertise. As a trade-off between formalization and usability, a recurrent solution in the literature is the use of Property Specification Patterns (PSPs), i.e., English-like structured natural sentences that provide a direct mapping to one or more logics [1].

In this paper, we present REQV, a tool developed in the context of the H2020 EU CERBERO [2] project[1]. REQV leverages on PSPs to provide the user with a friendly, yet formally grounded specification environment. REQV incorporates our contribution [3], whereby PSPs are extended by considering Boolean as well as atomic numerical assertions of the form $x \bowtie c$, where x is a variable of the system, $c \in \mathbb{R}$ is a constant real number and the operator $\bowtie \in \{<, <=, =, >=, >\}$ has the usual interpretation. Furthemore, we presented an encoding to reduce the *inner consistency* of extended PSPs, *i.e.*, logical errors in the specification that prevent any possible system to satisfy all requirements, to the Linear Temporal Logic (LTL) [4] satisfiability problem. We also extended previous works with a new algorithm to find a minimum set of conflicting requirements in case of inconsistency. We collected all these functionalities in a Java library called SPECPRO[2]. REQV exploits the capabilities of SPECPRO to provide an easy-to-use interface for the verification of requirements. Its goal is to enable users with no background knowledge of formal methods and logic languages to write requirements as PSPs and check their consistency. REQV also aims at minimizing the setup process for the user, and therefore it is developed as a web application that can easily be accessed within a browser.

The rest of the paper is organized as follows; in Section II we present the architecture of REQV, and in Section III we describe the implementation details. We conclude the paper in Section V with some final remarks and ideas for future extensions.

II. Architecture

The architecture of REQV is outlined in Figure 1. REQV can be accessed by multiple users through the internet. It is comprised of the following components:

- The *Front-end*, a web application that provides a graphical user interface for the user and performs asynchronous calls to the back-end.
- the *Back-end*, a server application that provides services as REST APIs. It builds on top of the SPECPRO library to execute consistency checking and inconsistency explanation tasks in background. Services are accessible with the JWT authentication mechanism over HTTPS.
- SPECPRO [3], a Java library containing a parser for the extended PSPs language and algorithms to check the consistency and to find a minimal unsatisfiable core (MUC) of requirements, employing on existing model checkers.
- An off-the-shelf model checker capable to check the satisfiability of LTL formulas. Currently, AALTA [5] and NUSMV [6] are supported.

[1]http://www.cerbero-h2020.eu/

[2]https://gitlab.sagelab.it/sage/SpecPro

- *A database* to store requirements, user data and other information such as tasks execution logs.

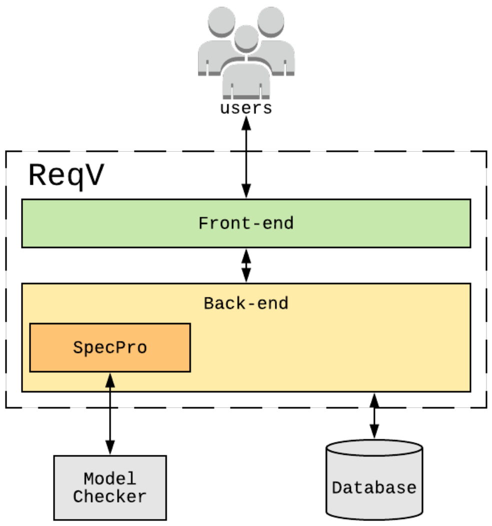


Fig. 1. REQV architecture diagram.

III. IMPLEMENTATION

REQV's implementation relies on different open-source tools and frameworks. In the following, we describe the implementation details of the main two components, namely the front-end and the back-end of ReqV.

A. Front-End

The front-end is a web application implemented in Type-script, using the ANGULAR[3] framework. To use the application, the user must authenticate first, and then she can create a new project or select an existing one. A project is a collection of requirements, along with some supplemental information (*e.g.*, the title and the description). Selecting a project displays in a table the list of requirements which are already contained in the project, as showed in Figure 3. In REQV, requirements have to be written as Property Specification Patterns [7]. PSPs are meant to describe the essential structure of system behaviours in form of structured English sentences [8] and to provide expressions of such behaviors in a range of common formalisms. An example of a PSP is given in Figure 2 — with some part omitted for sake of readability.[4] In more detail, a PSP is composed of two parts: (*i*) the *scope*, and (*ii*) the *body*. The *scope* is the extent of the program execution over which the pattern must hold, and there are five scopes allowed: *Globally*, *Before*, *After*,*Between*, *After-until*. The *body* of a pattern describes the behavior that we want to specify.

REQV automatically performs a syntactic check on every requirement, coloring in red the ones containing an error. The

[3]https://angular.io/

[4]The full list of PSPs considered in this paper and their mapping to LTL and other logics is available at http://ps-patterns.wikidot.com/.

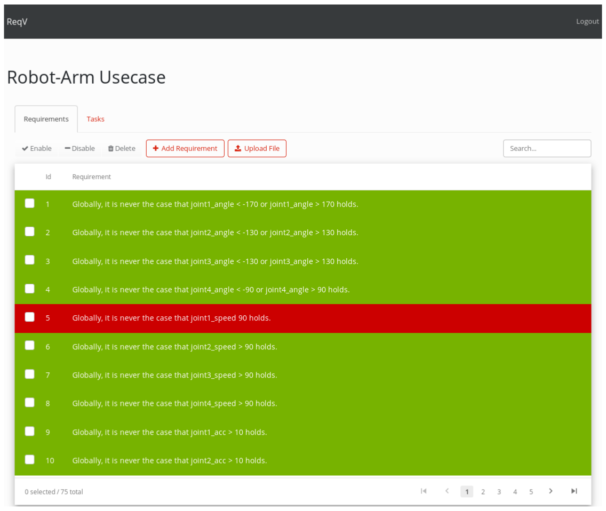
Response

Describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to.

Structured English Grammar
It is always the case that if P holds, then S eventually holds.

Example
It is always the case that if object_detected holds, *then* moving_to_target *eventually holds.*

Fig. 2. Response Pattern. A pattern is comprised of a *Name*, an (informal) statement describing the behaviour captured by the pattern, and a (structured English) statement that should be used to express requirements.



Robot-Arm Usecase

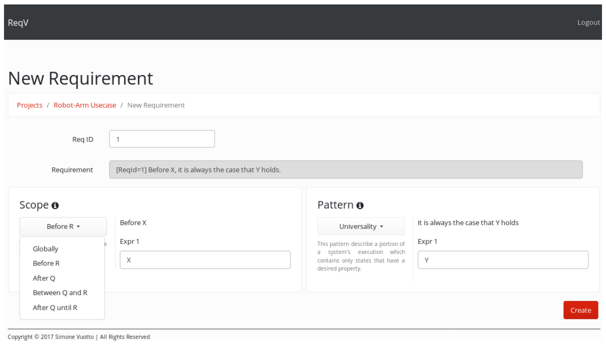
	Id	Requirement
	1	Globally, it is never the case that joint1_angle < -170 or joint1_angle > 170 holds.
	2	Globally, it is never the case that joint2_angle < -130 or joint2_angle > 130 holds.
	3	Globally, it is never the case that joint3_angle < -130 or joint3_angle > 130 holds.
	4	Globally, it is never the case that joint4_angle < -90 or joint4_angle > 90 holds.
	5	Globally, it is never the case that joint1_speed 90 holds.
	6	Globally, it is never the case that joint2_speed > 90 holds.
	7	Globally, it is never the case that joint3_speed > 90 holds.
	8	Globally, it is never the case that joint4_speed > 90 holds.
	9	Globally, it is never the case that joint1_acc > 10 holds.
	10	Globally, it is never the case that joint2_acc > 10 holds.

0 selected / 75 total

1 2 3 4 5

Fig. 3. REQV front-end interface showing a list of requirements inserted by the user. Requirements colored in green are syntactically correct, the ones in red are not. A grey requirement indicates it has been disabled and therefore not considering during the analysis.

syntactic check consists in parsing the textual requirement with a context-free grammar and reporting the position of any error, i.e., failure to comply with the structure of the grammar. Compared with the grammar presented in [1], we make the scope optional (considering *Globally* as the default option), we add the possibility to specify an ID directly in the pattern, and we extend the grammar to support atomic numerical constraints, as described in [3]. The user can add, edit, delete or disable a requirement. A disabling requirement is kept in the database, but it is not considered during during the consistency checking analysis. To insert new requirements, the user can either upload a text file containing a list of requirements – one per line – or use the user interface to help build correct PSPs, as illustrated in Figure 4. The interface is similar to PSPWizard [9]: it allows to select a scope and a pattern, showing a short description of the intended meaning, and displaying some text fields to complete the pattern. The main difference with respect to PSPWizard is that our interface can handle the extended language with numerical signals, and it is tightly integrated in the rest of the framework, while PSPWizard is a standalone application that only shows the resulting mapping into a target logic. By contrast, PSPWizard also handles other PSPs, such as real-time and probabilistic properties, and an encoding to many different logics, which at the moment we do not support.



New Requirement

Req ID: 1

Requirement: [ReqId=1] Before X, it is always the case that Y holds.

Before X

Expr 1: X

it is always the case that Y holds

This pattern describe a portion of a system's execution which contains only states that have a desired property.

Expr 1: Y

Copyright © 2017 Simone Vuotto | All Rights Reserved

Fig. 4. ReqV PSP composition interface.

Finally, ReqV allows for *push-button verification*: moving to the Tasks tab, the user can simply press a button and launch a new *consistencyCheck* task on the server, described in the following section. The task will run in background and the interface will update automatically once the task is completed. In case of inconsistency, the user can also run an inconsistency explanation task that will find a minimum unsatatisfiable set of requirements, helping the user to debug the specification.

B. Back-End

The back-end is a Java server application based on the Spring Boot[5] framework and PostgreSQL[6] database engine. It provides a set of endpoints REST APIs with JSON format for data exchange. In order to access services and user's own data, ReqV employ the JWT [10] open standard for authentication over HTTPS. Services provide access to four main resources:

- **User:** contains basic information about the logged user.d;
- **Projects:** list of saved projects. Each project has a title, a description and some configuration data, and it is associated with one user.
- **Requirements:** list of requirements saved in a project. Each requirement contains its textual representation, its state (after the syntax check is executed) and other application dependent information.
- **Tasks:** list of completed and executing tasks for a given project. A task contains information about its state (*i.e.* if it is still running, or succeeded/failed), log information and type.

[5]https://spring.io/projects/spring-boot
[6]https://www.postgresql.org/

Each resource can be accessed, modified or deleted with the usual HTTP methods calls. In particular, there are three types of tasks that can be created for each project:

- **Translate**: the requirement specification is translated into a LTL satisfiability problem and a file with the encoded specification is returned.
- **ConsistencyCheck**: a consistency check of the specification is executed in background. It consists of variable type checking, *i.e*, ensuring that no variable is used both as a Boolean and numerical value, and of an LTL satisfiability check of the encoded requirements;
- **FindInconsistency**: it executes in background a research of a minimal set of requirements that can help explain the inconsistency, if any. The algorithm iteratively removes some requirements and performs the satisfiability check of the remaining set, keeping only a minimal subset of them that maintain the inconsistency.

Only one task per project at a time is allowed: if a task is still running, further requests to instantiate a new task will be aborted. Finally, as anticipated in Section II, all the tasks are based on SpecPro capabilities. For a full list of APIs, the reader is redirected to https://reqv.sagelab.it/api/swagger-ui.html.

C. SpecPro

As mentioned before, SpecPro is an open-source Java library that implements a parser of the extended PSPs, translators for various LTL model-checkers and the algorithms to performs the tasks mentioned before, namely the consistency checking and the inconsistency explanation tasks. In particular, it takes as input a string stream of the requirements in textual form, it builds an intermediate representation of them, maintaining a symbol table for variables and constants, and it provides some utility classes to handle the execution of the model-checking process in background, interpreting the produced output. Finally, SpecPro also provides a minimal command-line interface that enables the user to performs the same tasks described in III-B from a shell.

IV. Example

Id	PSP Requirements
r_1	Globally, it is always the case that A holds.
r_2	It is never the case that A holds.
r_3	Globally, it is always the case that B holds.
r_4	Globally, it is always the case that if B holds, then C eventually holds.
r_5	Globally, it is never the case that $C < 10$ holds.
r_6	Globally, it is always the case that A and B holds.
r_7	After B, D eventually holds.

Fig. 5. Set R of inconsistent PSPs.

Here we illustrate the use of ReqV. We assume that, in the process of writing the requirements about a system, a user has assembled the set in Figure 5. Now the goals are: (i) check if the requirements are written properly as PSPs, and (ii) check if they are consistent; in case they are not, enable the user to selectively review them. These goals are achieved by following the steps below:

1) Login and create a new Project.
2) Load the set of requirements listed in Figure 5 as text file and then push the button "Check".
3) REQV executes syntax checking and partial type checking, reporting two requirements as "red": In r_2 the scope is missing while r_5 redefine a boolean variable, C, as numerical signal (C $<$ 10).
4) Modify r_2 by adding *Globally* as scope and r_5 by replacing the numerical constraint C $<$ 10 with boolean signal C.
5) Te subsequent syntax checking does not report any error; Check the requirements for consistency. The system Back-End calls SPECPRO, and it reports that the requirements are inconsistent.
6) Run inconsistency explanation; SPECPRO returns that r_1 and r_2 are inconsistent (it returns the first one found). REQV highlights r_1 and r_2.
7) At this point, we can either modify one of the requirement or delete one; Delete r_1, and run the consistency check again.
8) SPECPRO reports that r_2 and r_6 are still conflicting.
9) The process can be iterated until the requirements become consistent.

Notice how REQV can be used to "trim" the specification from inconsistent requirements in an iterative fashion. At each step, a small set of inconsistent requirements is pointed out, and the user is guided in the process of composing a consistent specification through removing or rewriting all constraints that make it. This is a precondition towards obtaining realizable specifications.

V. CONCLUSION AND FUTURE WORK

In this paper we presented REQV, a tool for the management and verification of functional requirements. In [3] we showed that the encoding used for consistency checking can scale up to thousands of requirements and it can check in few seconds the specification of a robotic-arm case study composed of 76 requirements. In case of inconsistency, the tool can help the user in identifying a minimal set of conflicting requirements, and in debugging faulty specifications. The tool has undergone some alpha-testing, but it is still under activedevelopment. REQV is available online at https://reqv.sagelab.it [7] and the source code can be consulted at https://gitlab.sagelab.it/sage/ReqV. A video tutorial is currently available at https://youtu.be/2WKSxh64Z2k.

REFERENCES

[1] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[7]The tool is currently accessible to a restricted audience only. To get access to the tool please contact Simone Vuotto. Reviewers are invited to try the online tool with the temporary credentials username: "test" password: "test".

[2] M. Masin, F. Palumbo, H. Myrhaug, J. de Oliveira Filho, M. Pastena, M. Pelcat, L. Raffo, F. Regazzoni, A. Sanchez, A. Toffetti *et al.*, "Cross-layer design of reconfigurable cyber-physical systems," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 740–745.
[3] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto, "Consistency of property specification patterns with boolean and constrained numerical signals," in *NASA Formal Methods: 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, vol. 10811. Springer, 2018, pp. 383–398.
[4] A. Pnueli and Z. Manna, "The temporal logic of reactive and concurrent systems," *Springer*, vol. 16, p. 12, 1992.
[5] J. Li, Y. Yao, G. Pu, L. Zhang, and J. He, "Aalta: an LTL satisfiability checker over infinite/finite traces," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 731–734.
[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *14th International Conference on Computer Aided Verification (CAV 2002)*, 2002, pp. 359–364.
[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International conference on Software engineering*, 1999, pp. 411–420.
[8] S. Konrad and B. H. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 372–381.
[9] M. Lumpe, I. Meedeniya, and L. Grunske, "Pspwizard: machine-assisted definition of temporal logical properties with specification patterns," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 468–471.
[10] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," Tech. Rep., 2015.