Requirements-Driven Design of Cyber-Physical Systems

Simone Vuotto

[1] Università degli Studi di Genova
simone.vuotto@edu.unige.it
[2] Università degli Studi di Sassari
svuotto@uniss.it

Abstract. The demand for more and more complex Cyber-Physical Systems (CPS), *i.e.* tightly coupled hardware and software components that operate in a physical (unsupervised) environment, is increasing. More often than not, they require guarantees in terms of performance, safety, security, sustainability, etc. The design of CPS is therefore a fundamental, yet difficult and challenging task. The Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments (CERBERO) EU project [9] aims at developing a set of tools, cooperating at different layers of abstraction, to ease the design process and increase adaptivity of such systems.
In this context, our research focuses on the analysis of the first item produced in the design process: software and system requirements. It is well known that a flaw in the requirements specification can lead to delays, additional costs and, possibly, the failure of the project. Nonetheless, due to the intrinsic difficulty of dealing with natural language sentences, requirements are often checked manually, an error-prone and time-consuming activity.
Our goal is to design a tool to automatize this task and, starting from the formalization of requirements, generate artifacts that can be used to validate and verify the items produced along the design process.

1 Introduction

The definition and elicitation of requirements is usually the first step in the design of software and cyber-pysical systems. They are used as a guideline for the implementation and as a reference for the verification and validation of the final product (see, for example, the V-Model depicted in Figure 1 on the left, often used in the design of CPS). Despite their importance, the assessment of requirements is still largely carried out manually. The Requirements Engineering (RE)[12] research field aims at developing tools and techniques to analyze and handle requirements in a more efficient and automatic way. The goal of our research is to build a tool that can assist the CPS designer along all the design process, exploiting the information available in the requirements specification document to test and monitor the behavior of the final implementation.

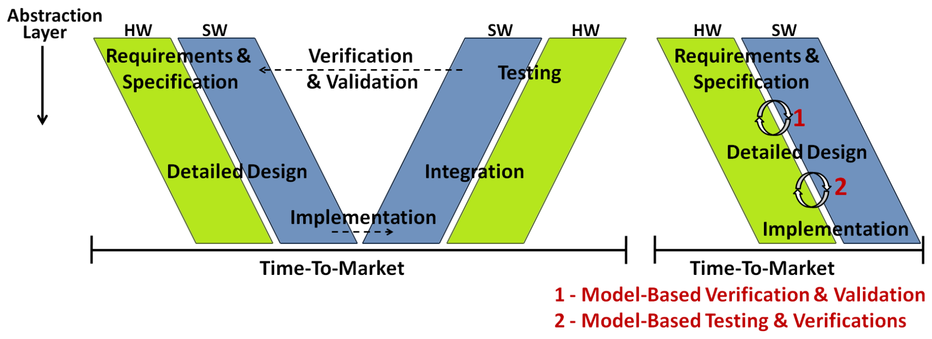
Abstraction Layer
HW
SW
Requirements & Specification
Verification & Validation
Detailed Design
Implementation
SW
HW
Testing
Integration
Time-To-Market
HW
SW
Requirements & Specification
1
Detailed Design
2
Implementation
Time-To-Market
1 - Model-Based Verification & Validation
2 - Model-Based Testing & Verifications

Fig. 1. CERBERO design approach.

This objective matches well with one of the main CERBERO's missions, namely reducing the time-to-market of new designed CPSs by means of an integrated toolchain for comprehensive model-based design of CPS (Figure 1 on the right).

The first main challenge of our research is to evaluate requirements *consistency*: informally, it means detecting errors, missing information and deficiencies that can compromise the interpretation and implementation of the intended system behavior. At a syntactic level, this may involve the check for compliance with standards and guidelines, such as the use of a restricted grammar and vocabulary. We call this task *Compliance Checking*.

However, most of the inconsistencies reside at a semantic level, *i.e.* in their intended meaning. Therefore, it is necessary to interpret the requirements semantics and represent them with a formalism that allows for reasoning. How to formalize and translate requirements into a formal representation is an open research question. A recurrent solution in the literature is the use of Property Specification Patterns (PSPs), first introduced by [4]. PSPs provide a direct mapping from English-like structured natural languages to one or more logics. A survey of all available patterns and their translations has been conducted by [1]. Other approaches, like [6], employ Natural Language Processing techniques to extract the representation directly from fully natural language requirements. In both cases, the integration of an external Knowledge Base is often desirable to capture the semantics of domain-specific language. We call *NL2FL* the component that performs this formalization.

Given the set of requirements represented in a formal logic, it becomes possible to implement more sophisticated checks and exhaustive reasoning. We formally define this task *Consistency Checking* analysis [7]. Consistency Checking can range from simple variables type and domain checks to more complex activities, like the evaluation of the intended system behavior over time.

Moreover, the formalization of requirements and the consistency checking are enablers for other tasks we would like to tackle, namely the automatic generation of test suites [14] and runtime monitors [8]. The former is used to verify the compliance of the code with the requirement specification, while the latter

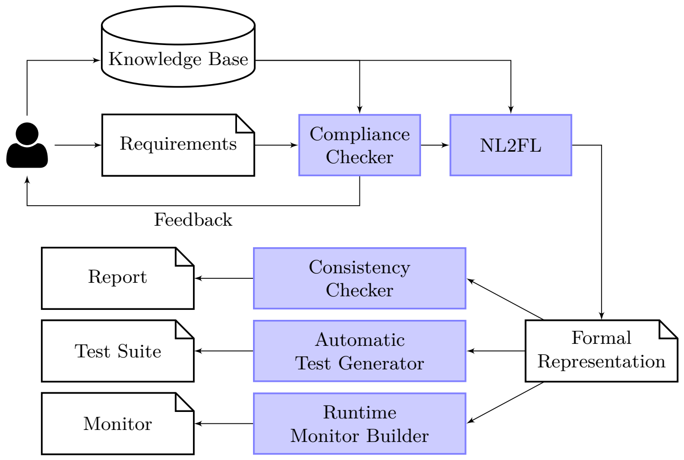

Fig. 2. General framework of the requirement analysis tool

is useful to ensure that no unexpected behavior, which may have escaped the test phase, is executed at runtime on the target CPS. The full overview of the framework we intend to realize is depicted in Figure 2.

Finally, the choice of which logic to use largely affects the reasoning power and the kind of requirements that can be formalized: qualitative, real-time and/or probabilistic. For example, in some logics it is only possible to specify that an event e will eventually happen in the future, while others can also constraint the time frame (*e.g.* e will happen within 5s) or its likelihood (*e.g.* e will happen with probability $p \geq 0.5$). We started from Linear Temporal Logic (LTL)[13] because it has a good balance between expressiveness and complexity, and it is widely used in the literature, but we also plan to explore more expressive logics.

2 Consistency of Property Specification Patterns

Our first contribution [11], presented a tool for the consistency checking of qualitative requirements expressed in form of PSPs with constrained numerical signals. An example of requirement that we can handle is:

Globally, it is always the case that if $proximity\_sensor < 20$ holds, then arm_idle eventually holds.

We first translate every requirement $r_i \in R$ in LTL($\mathcal{D}_C$), an extension of LTL over a constraint system $D_C = (\mathbb{R},<,=)$, with atomic constraints of the form $x < c$ and $x = c$ (where $c \in \mathbb{R}$ is a constant real number and '$<$" and "$=$" have the usual interpretation). We then show how the new problem can be reduced to LTL satisfiability. Let $X(\phi)$ be the set of numerical variables and $C(\phi)$ be the set of constants that occur in ϕ. We compute:

- the LTL($\mathcal{D}_C$) formula ϕ_i for every requirement $r_i \in R$;
- the conjunctive formula $\phi = \phi_1 \wedge ... \wedge \phi_n$;
- a set $M_x(\phi)$ of boolean propositions representing possible values of $x \in X(\phi)$;
- the formula Q_M encoding the constraints over $M_x(\phi)$ $\forall x \in X(\phi)$;
- the formula ϕ' that substitute all $x \in X(\phi)$ in ϕ with a set of boolean propositions from $M_x(\phi)$;

Given the LTL($\mathcal{D}_C$) formula ϕ over the set of Boolean atoms *Prop* and the terms $C(\phi) \cup X(\phi)$ we have that ϕ is satisfiable if and only if the LTL formula $\phi_M \wedge \phi'$ is satisfiable. This result is important because it shows that LTL($\mathcal{D}_C$) is decidable and that we can exploit state-of-the-art LTL model checkers.

However, LTL satisfiability is not sufficient to guarantee the correctness and completeness of the requirements set, and a more extensive feedback is necessary in case of inconsistency. In order to answer these needs, we extended the work in [11] in two directions.

2.1 Connected Requirements Check

Given a set of requirements $R = r_1, \ldots, r_n$, we want to check if one or more of such requirements are completely unrelated to the others, meaning that they describe some behaviors that do not interact with the main bulk of the system. This may happen in an underspecified requirements set or for some spelling errors. To find these faults, we first build the undirected graph $G = (V, E)$ representing the connections in R, such that:

- $v_i \in V \ \forall r_i \in R$;
- $(i, j) \in E$ if $X(r_i) \cap X(r_j) \neq \emptyset \ \forall r_i, r_j \in R, i \neq j$

where $X(r_i)$ is the set of variables, boolean or numerical, that appear in r_i. We then perform a static analysis on the graph, checking if:

- a variable is never used to connect two requirements;
- the graph is composed of two or more connected components.

Both cases suggest that the system is underspecified, and a corresponding warning message is generated.

Consider, for instance, the set of requirements:

R_1 It is always the case that **x >= 0 and x <= 10** holds.
R_2 After **x > 5**, **y or z** eventually holds.
R_3 It is always the case that if **w > 10** holds, then **y and z** previously held.
R_4 Globally, it is always the case that if **j** holds, then **k** eventually holds.
R_5 Globally, it is always the case that if **k** holds, then **j** eventually holds.

The corresponding graph is showed in Figure 3. In this case two warnings are generated: (1) The variable **w** is used only in R_3; and (2) requirements R_4 and R_5 are part of a separated component, meaning that they are describing the behavior of a subsystem that does not interact with the main system, or some requirement is missing.

In the current implementation boolean and numerical variables are handled in the same way, but a more fine grained check could take into account every single range (in the example before, what happen when **x** is between 0 and 5?).

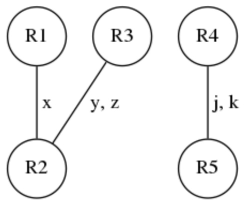

Fig. 3. Connected Graph Example

2.2 Inconsistency Requirements Explanation

In case of inconsistent requirements, we are interested in finding a small subset of them that explain the problem. This can help the designer in identifying and fixing mistakes in a (possibly very large) set of requirements. We perform this task, called *Inconsistent Requirements Explanation*, using the deletion-based algorithm represented in Algorithm 1. The algorithm finds an high-level minimal unsatisfiable core (HLMUC) [10], *i.e.*, an irreducible subset of requirements that maintain the inconsistency. It exploits the simple observation that if the set $R' \leftarrow \{R \backslash r : r \in R\}$ is still inconsistent, then r is not necessary for the HLMUC. If, on the other hand, R' is consistent, r is part of one HLMUC and can not be removed. Therefore, we iteratively remove a specification r_i and check for the consistency of the new set. The algorithm terminate after all r_i with $i \in 1 \dots n$ have been checked.

To better illustrate how the algorithm works, consider the following example:

R_1 Globally, it is always the case that A holds.
R_2 Globally, 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 holds as well.
R_5 Globally, it is never the case that C holds.
R_6 Globally, it is always the case that A and B holds.
R_7 After B, D eventually holds.

In this case it is simple to see that there are 4 *HLMUC* in R: $\{R_1, R_2\}$, $\{R_2, R_6\}$, $\{R_3, R_4, R_5\}$, $\{R_4, R_5, R_6\}$. Algorithm 1 finds only one of these sets, depending on the evaluation order. A possible step-by-step execution is illustrated in the following table.

r_i	R'	$isConsistent(R')$
R_1	$\{R_2, R_3, R_4, R_5, R_6, R_7\}$	$False$
R_2	$\{R_3, R_4, R_5, R_6, R_7\}$	$False$
R_3	$\{R_4, R_5, R_6, R_7\}$	$False$
R_4	$\{R_5, R_6, R_7\}$	$True$
R_5	$\{R_4, R_6, R_7\}$	$True$
R_6	$\{R_4, R_5, R_7\}$	$True$
R_7	$\{R_4, R_5, R_6\}$	$False$

The final result is $R' = \{R_4, R_5, R_6\}$, but if we used the inverse order, we would have obtained $R' = \{R_1, R_2\}$.

Algorithm 1 is pretty simple although inefficient, because it performs $|R|$ satisfiability checks. We also implemented a faster algorithm that employs a recursive strategy, but it has been omitted due to space constraints.

Algorithm 1 Linear Deletion-Based Inconsistency Finder Algorithm

```
1: function findInconsistency($R$)
2:     $R' \leftarrow R$
3:     for $r_i \in R$ do
4:         $R' \leftarrow R' \setminus r_i$
5:         if isConsistent($R'$) then
6:             $R' \leftarrow R' \cup \{r_i\}$
7:         end if
8:     end for
9:     return $R'$
10: end function
```

3 Future work

We are now working on the problem of automatic tests generation from LTL specifications. This is a widely studied problem [5], but most of the proposed solutions assume the availability of the model that the specification refer to. They often exploit the model checkers capability to produce a counterexample in order to generate a finite trace that can be interpreted as a test case. However, in our case we don't have such a model, and therefore an alternative approach is required. Using the requirement-based coverage metric described in [14], our idea is to extract finite paths from the Büchi Automata representation of the specification. We are currently developing an algorithm to generate these paths, but scalability is an important issue to be taken into account.

Moreover, the consistency checking of requirements has been improved in the current paper, but more work is necessary in order to validate the specification. We plan to add more checks in parallel with the current ones, like the Realizability Check implemented in [2]. We also believe that the performance of HLMUC extraction can be improved, exploiting the internal information produced by the model checker to drive the research of the unsatisfiable core in a more efficient way.

Finally, for future works we would also like to both extend the natural language interface with less restrictive constraints and explore more expressive formalisms. In this regard, two research directions are possible: extend the current work with probabilistic or real-time logics, or introduce the contract-based design paradigm. This paradigm is an emerging solution to deal with complex systems by decomposing them in components and defining their properties in term of contracts [3]. Contract-based design is particularly interesting in our context because it presents an appealing parallelism with how requirements are usually specified and, at the same time, it can reduce the verification complexity.

Acknowledgments The research of Simone Vuotto is funded by the EU Commission H2020 Programme under grant agreement N.732105 (CERBERO project).

References

1. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering 41(7), 620–638 (2015)
2. Bloem, R., Cavada, R., Pill, I., Roveri, M., Tchaltsev, A.: Rat: A tool for the formal analysis of requirements. In: International Conference on Computer Aided Verification. pp. 263–267. Springer (2007)
3. Cimatti, A., Demasi, R., Tonetta, S.: Tightening the contract refinements of a system architecture. Formal Methods in System Design 52(1), 88–116 (2018)
4. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International conference on Software engineering. pp. 411–420 (1999)
5. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Software Testing, Verification and Reliability 19(3), 215–261 (2009)
6. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: Arsenal: automatic requirements specification extraction from natural language. In: NASA Formal Methods Symposium. pp. 41–46. Springer (2016)
7. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) 5(3), 231–261 (1996)
8. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
9. Masin, M., Palumbo, F., Myrhaug, H., de Oliveira Filho, J., Pastena, M., Pelcat, M., Raffo, L., Regazzoni, F., Sanchez, A., Toffetti, A., et al.: Cross-layer design of reconfigurable cyber-physical systems. In: Proceedings of the Conference on Design, Automation & Test in Europe. pp. 740–745. European Design and Automation Association (2017)
10. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated deletion-based extraction of minimal unsatisfiable cores. Journal on Satisfiability, Boolean Modeling and Computation 9, 27–51 (2014)
11. Narizzano, M., Pulina, L., Tacchella, A., Vuotto, S.: Consistency of property specification patterns with boolean and constrained numerical signals. In: NASA Formal Methods Symposium. pp. 383–398. Springer (2018)
12. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. pp. 35–46. ACM (2000)
13. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57. IEEE (1977)
14. Whalen, M.W., Rajan, A., Heimdahl, M.P., Miller, S.P.: Coverage metrics for requirements-based testing. In: Proceedings of the 2006 international symposium on Software testing and analysis. pp. 25–36. ACM (2006)