

FORMAL REQUIREMENTS ANALYSIS AND
SPECIFICATION-BASED TESTING IN
CYBER-PHYSICAL SYSTEMS

Simone Vuotto

A thesis submitted in fulfillment
of the requirements of the degree of
Doctor of Philosophy

Advisors:

Professor Armando Tacchella, University of Genoa, Italy

Professor Luca Pulina, University of Sassari, Italy

University of Genoa, Italy

Acknowledgements

This thesis is the result of three challenging years of work that I shared with the people of AIMS Lab (University of Genoa), and IDEA Lab (University of Sassari). First of all, I would like to acknowledge and thank my advisors Armando and Luca for their scientific and personal support that they gave me along this difficult journey, especially when firmness swayed. Similarly, I am very much indebted with my coauthors Massimo Narizzano and Laura Pandolfo for their contribution and precious advices. I would also like to acknowledge the CERBERO H2020 and the FitOptivis ECSEL EU projects, that sustained my work, and the great people that participated and contributed to them.

However, aside the scientific aspects there are so many other people that I would like to thank. First, I have been very lucky to spend lot of this time in the beautiful Sardinia island and I felt very welcomed by sardinians. A special mention goes to the IDEA Lab people: Claudio, Francesca, Laura, Luca, Maria Grazia, Monica, Tiziana and Valentina. Outside the university I also met fabulous people. In particular, I would like to thank my flatmate Giulia, with whom I shared the apartment, many delicious dinners and many funny moments along the way, and the gym group that, although I wasn't really constant with my exercises, welcomed me as one of them. Among them, there is one person that I cannot thank enough: Giovanni. He took me under his protective wing and he was always there to listen and to give helpful advices on any matter. Another important person that I met in Sassari is Imèn. The time we shared was special and she managed to be a constant presence regardless the distance.

I also would like to thank my Genoese friends, and especially Nicolò, Marta, Pinka and Diego that helped me feel the past lockdown less harsh. The same applies for my Erasmus friends: Giulia, Margherita, Martina, and Miguel. Many

years passed from our meeting, but distance didn't affect our friendship and when we talk it is like we never left.

Another big thanks goes to my historic group of friends in Brescia: Alice, Andrea, Caterina, Chiara, Corrado, Erica, Fabio, Federica, Francesco and Vanessa. Despite the many years I spent far from my hometown, they have always been a fix point, a certainty in a world of chaos.

Finally, my gratitude goes to my parents and all my family for their love and support. I feel lucky to be part of such a large and cohesive family, where everyone is ready to help each other. Above all, I would like to dedicate this thesis to my grandma Annunziata, who unfortunately left us at the begging of this troubled 2020, and my grandpa Bonaventura. They are the two pillars of the family and, with more than 60 years of marriage, they taught me the real meaning of love. I hope to always make you proud.

Abstract

Formal requirements analysis plays an important role in the design of safety- and security-critical complex systems such as, *e.g.*, Cyber-Physical Systems (CPS). It can help in detecting problems early in the system development life-cycle, reducing time and cost to completion. Moreover, its results can be employed at the end of the process to validate the implemented system, guiding the testing phase. Despite its importance, requirements analysis is still largely carried out manually due to the intrinsic difficulty of dealing with natural language requirements, the most common way to represent them. However, manual reviews are time-consuming and error-prone, reducing the potential benefit of the requirement engineering process. Automation can be achieved with the employment of formal methods, but their application is still limited by their complexity and lack of specialized tools.

In this work we focus on the analysis of requirements for the design of CPSs, and on how to automatize some activities related to such analysis. We first study how to formalize requirements expressed in a structured English language, encode them in linear temporal logic, check their consistency with off-the-shelf model checkers, and find minimal set of conflicting requirements in case of inconsistency. We then present a new methodology to automatically generate tests from requirements and execute them on a given system, without requiring knowledge of its internal structure. Finally, we provide a set of tools that implement the studied algorithms and provide easy-to-use interfaces to help their adoption from the users.

Contents

1	Introduction	2
1.1	Research Area, Motivations and Goals	2
1.2	Thesis outline	5
1.3	Relevant Publications	6
1.3.1	In preparation	7
2	Background	8
2.1	ω -languages and Automata	8
2.2	Linear Temporal Logic	10
2.2.1	Syntax	10
2.2.2	Semantics	11
2.3	Sanity Checking	12
2.3.1	LTL satisfiability	13
2.4	Property Specification Patterns	14
2.5	Minimal Unsatisfiable Cores	17
2.6	Conformance Testing	18
3	State of the Art and Related Work	20
3.1	Requirements Formalization and Analysis	20
3.1.1	Formalization and Consistency Checking	20
3.1.2	Inconsistency Explanation	22
3.2	Automatic Testing from LTL specification	23
4	Consistency Checking	26
4.1	Constraint Property Specification Patterns	29

4.2	Inconsistency Explanation	37
4.2.1	Linear Deletion-Based MUC Extraction	38
4.2.2	Dichotomic MUC Extraction	39
4.3	Analysis with Probabilistic Requirement Generation	42
4.3.1	Evaluation of LTL(\mathcal{D}_c) Satisfiability	43
4.3.1.1	Evaluation of LTL satisfiability solvers.	43
4.3.1.2	Evaluation of scalability.	44
4.3.2	Evaluation of MUC Extraction	47
4.4	Analysis with a Controller for a Robotic Manipulator	50
5	Requirements-Based Black-Box Testing	54
5.1	Automatic Test Case Generation from LTL specification	56
5.1.1	Requirements and Automata Processing	57
5.1.2	Test Oracle	57
5.1.3	Input Generator	59
5.2	Experimental Analysis	61
5.2.1	Syntcomp Benchmarks	61
5.2.2	Adaptive Cruise Control	63
5.2.3	Robotic Manipulator	66
6	Tools	68
6.1	SpecPro	69
6.1.1	Parse And Translate Requirements	69
6.1.2	Consistency Checking	70
6.1.3	Testing	71
6.1.3.1	SUT	73
6.2	ReqV	74
6.2.1	Architecture	74
6.2.2	Workflow	77
6.3	ReqT	80
6.3.1	Workflow	81
7	Conclusion	84
7.1	Summary of Contributions	84

CONTENTS

ix

7.2 Open challenges and future work 86

Chapter 1

Introduction

1.1 Research Area, Motivations and Goals

Requirements Engineering (RE) is the branch of software and system engineering concerned with real-world goals for, functions of, and constraints on systems [Lap17]. Requirements play an important role in the development life-cycle of a system; they usually are defined and collected at the beginning of the design process and influence all the subsequent steps. They are used and shared among many different stakeholders, namely the set of individuals that have some interest in the realization of the system, and they can range from high-level abstract statements to formal and mathematically rigorous specifications. For this reason, requirements are sometimes categorized into three (or more) levels of abstraction: user requirements, system requirements and design specifications. Other kinds of taxonomies are also possible, *e.g.*, based on their content (functional vs non-functional requirements). The definition of a requirements specification document raises many challenges that have to be undertaken. The RE process involves a large variety of activities to tackle such problems, such, for example, requirements elicitation and discovery; management and traceability; analysis; modeling; verification and validation; etc. Hence, the RE research field aims at developing tools and techniques to address these activities in a more efficient and automatic way. Formal methods proved to be a powerful ally in tackling many of such activities, providing precise formalism and reasoning

capabilities [WLB09].

In this thesis we focus on some of these challenges, addressed in the context of the “*Cross-layer model-based framework for multi-objective design of Reconfigurable systems in uncertain hybrid environments*” (*Cerbero*) H2020 EU project [MPM⁺17, PFS⁺19] first, and the “*From the cloud to the edge – smart Integration and Optimisation Technologies for highly efficient Image and Video processing Systems*” (*Fitptivis*) ECSEL EU Project [AABdB⁺19] later. In particular, we deal with system level functional requirements of Cyber-Physical Systems (CPS), *i.e.*, systems with tightly coupled hardware and software components that operate in a physical (unsupervised) environment.

Our first research question is how to represent, formalize and check the consistency of requirements. To deal with this problem, two different strategies are proposed in the literature: the former involves the application of Natural Language Processing (NLP) techniques to understand arbitrary requirements written in unrestricted natural language; the latter define a restricted and controlled language to eliminate ambiguity and to maintain a clear semantics. Examples of the former strategy are ARSENAL [GEL⁺16], that performs consistency checking and generates state-machine implementations for consistent sets of requirements, and [FB16], a general architecture and an evaluation tool that parses natural-language requirements, interacts with the users for clarifications, and create initial partial implementations. However, as stated in [FB16], the state of the art in natural-language processing is still far from what would be required to fully analyze system requirements. Moreover, [BGST12] argues against the use of NLP-based tools in Requirement Engineering tasks because they cannot provide guarantees of completeness and correctness, essential in safety- and security-critical system, and they could be counter-productive in practice. In spite of these limitations, they can still be effectively deployed in early stages of the design process and for non-critical systems, because they do not require any prior knowledge or restriction from the system engineer point of view; and they can address the large body of existing unformalized requirements. Examples of the latter strategy are Attempto Controlled English (ACE) [FKK08], that defines a controlled subset of natural language and can unambiguously translate text into discourse representation structures, a syntactic

variant of first-order logic, and Property Specification Patterns (PSPs), first introduced in [DAC99]. PSPs are a collection of parameterizable, high-level, formalism-independent specification abstractions usually based on a restricted English grammar. Since the original work of Dwyer [DAC99], a considerable number of property specification pattern systems have been proposed, grounding on different logics. A unified catalog that collects and combines all the proposed patterns is presented in [AGL⁺15]. In our work, we embrace PSPs backed with Linear Temporal Logic (LTL) [Pnu77], and extend them with the addition of constraint numerical signal. We formally present the proposed encoding and we show how to automate the consistency check of requirements using state-of-the-art tools available in the literature. Moreover, since general purpose model checkers and satisfiability checkers do not provide useful information for debugging in case of inconsistency, we propose two algorithms devoted to extract minimal subsets of inconsistent requirements.

The second research goal is to determine how to use the formalized and verified requirements to validate the implemented system. In theory, formal verification techniques can be used to automatically check the system against a given specification, giving strong correctness guarantees. However, these techniques suffer of known scalability issues and the complete verification of the specification becomes impractical or even impossible for complex systems. Moreover, it is often the case that a complete, explicit and formal model of the system is not available, making model checking unfeasible. For these reasons, testing is the preferred technique for hardware and software verification in industry, although it provides less guarantees; testing can only detect the presence of errors, not their absence. Nonetheless, a formal specification can still be of great practical use to automatically generate test suites to show conformance of the model and the actual implementation, or, just to derive “interesting” test cases to check the developed system [BJK⁺05]. A large body of work studied how to automatically generate tests from LTL specifications, exploiting the model checkers capability to generate counter examples for violated formulas [FWA09]. However, in our work we assume that a formal model of the system is not available, and therefore we need to look for an alternative strategy. Techniques aimed at automated test generation for black-box reactive systems relying on formal models of the specifi-

cations have been explored — see, e.g., [KT04, BBD19, KGHS98, SEG00, JJ05] — relying on the concept of specification coverage. Following this stream of research, in [AGR13] the authors describe a methodology for online testing of Java classes by exploiting a monitor derived from LTL specifications to check conformance of the system to stated requirements. In our work, we aim at generalizing and extending the approach proposed in [AGR13], addressing a more general class of properties. To validate our approach, we evaluate it in three different experimental settings, comparing it with other state-of-the-art techniques.

Finally, our third research objective is to make these technologies accessible to practitioners. To this end, we implemented three different tools:

- **SpecPro**: it is a Java library containing the implementation of all the algorithms discussed in this thesis. It also provides utilities to interact with external tools and it provides simple APIs for the developers. It is also the core upon which the other two tools are developed.
- **ReqV**: it is a web application that helps the user to write, manage and verify the consistency of requirements expressed as PSPs. The user can interact with the REQV front-end with any commercial browser and all the computationally demanding tasks are executed in background on the back-end.
- **ReqT**: it is a desktop application that is designed to automatically generate and execute tests on a given system. It takes in input a formal specification written as a list of PSPs or LTL requirements and produce a report with the executed tests and the result of their evaluation.

All of them have been applied in Cerbero [PFS⁺19] and partially in Fitoptivis [AABdB⁺19] projects.

1.2 Thesis outline

The remainder of this document is organized as follows. Chapter 2 introduces the necessary background and definitions on the topics we touch in this thesis. Chapter 4 discusses how to formalize a set of requirements, check their consistency and find minimal set of conflicting ones in case of inconsistency. Chapter 5

present a testing framework for black-box reactive systems that automatically generate tests from a set of consistent requirements. In Chapter 6 we present three different tools that implement the algorithms studied in this thesis and provide easy-to-use interfaces for non-expert users. Finally, in Chapter 7 we conclude the thesis with a summary of the achieved results and outlining some possible directions for future research.

1.3 Relevant Publications

Below I list the articles that, together with the fruitful collaboration of my co-authors, we published in different conferences and journals and that contain the contributions presented in this thesis.

- [NPTV18] Massimo Narizzano, Luca Pulina, Armando Tacchella, and Simone 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*, volume 10811, pages 383–398. Springer, Springer Verlag, 2018. https://doi.org/10.1007/978-3-319-77935-5_26
- [Vuo18] S. Vuotto. Requirements-driven design of cyber-physical systems. In *Proceedings of the Cyber-Physical Systems PhD & Postdoc Workshop 2018*, volume 2208 of *CEUR Workshop Proceedings*, pages 38–44. CEUR-WS, 2018. <http://ceur-ws.org/Vol-2208/6.pdf>
- [NPTV19] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto. Property specification patterns at work: verification and inconsistency explanation. *Innovations in Systems and Software Engineering*, 15(3-4):307–323, 2019. <https://doi.org/10.1007/s11334-019-00339-1>
- [VNPT19a] S. Vuotto, M. Narizzano, L. Pulina, and A. Tacchella. Automata based test generation with specpro. In *2019 IEEE/ACM 6th International Workshop on Requirements Engineering and Testing (RET)*, pages 13–16. IEEE, 2019. <https://doi.org/10.1109/RET.2019.00010>
- [VNPT19b] S. Vuotto, M. Narizzano, L. Pulina, and A. Tacchella. Poster: Automatic consistency checking of requirements with reqv. In *2019 12th*

IEEE Conference on Software Testing, Validation and Verification (ICST), pages 363–366. IEEE, 2019. <https://doi.org/10.1109/ICST.2019.00043>

- [Vuo19] S. Vuotto. Automata-based generation of test cases for reactive systems. In *Proceedings of the Cyber-Physical Systems PhD & Postdoc Workshop 2019*, volume 2457 of *CEUR Workshop Proceedings*, pages 96–106. CEUR-WS, 2019. <http://ceur-ws.org/Vol-2457/10.pdf>
- [NPTV20] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto. Automated requirements-based testing of black-box reactive systems. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings*, volume 12229, pages 153–169. Springer Nature, 2020. https://doi.org/10.1007/978-3-030-55754-6_9

1.3.1 In preparation

The following publication is currently under submission and is part of my research activity in the Fitoptivis EU project.

- [PPVon] L. Pandolfo, L. Pulina, and S. Vuotto. Smt-based consistency checking of configuration-based components specifications. Under submission

Chapter 2

Background

The goal of this chapter is to give basic definitions and terminology that will serve as a basis for the concepts presented in later chapters. The content of this chapter is intended mainly as a reference and it is not meant to be complete. For a more extensive treatment of the mentioned arguments, the reader can consult [BK08] and [BJK⁺05].

2.1 ω -languages and Automata

Given a set of symbols Σ (also called alphabet), a word over Σ is a sequence $A_0A_1\dots$ of symbols, where $A_i \in \Sigma, \forall i \geq 0$. A word is *finite* if it is a sequence of finite length, or it is *infinite* otherwise. By convention, we use lowercase Latin letters w, v, u to denote finite words and the Greek letter σ to denote infinite words. The Greek letter ε is used to indicate the special case of the empty word. We also use the notation $\sigma[i] = A_i$ for the $(i + 1)$ -th element of σ and $\sigma[j\dots] = A_jA_{j+1}\dots$ to denote the suffix of σ starting in the $(j + 1)$ -th symbol A_j .

Σ^* denotes the set of all finite words over Σ and a subset $\mathcal{L} \subseteq \Sigma^*$ is a finite language over Σ . Similarly, Σ^ω is the set of all infinite words over Σ and any subset $\mathcal{L}_\omega \subseteq \Sigma^\omega$ is a language of infinite words, also called ω -language. A finite language \mathcal{L}_1 and an infinite language \mathcal{L}_2 can be combined using the concatenation operator $\mathcal{L}_1.\mathcal{L}_2$ to create a new language defined by $\mathcal{L}_1.\mathcal{L}_2 = \{w\sigma \mid w \in \mathcal{L}_1, \sigma \in \mathcal{L}_2\}$.

In particular, we are interested in a class of ω -languages called ω -regular

languages, defined below. ω -regular languages are important for verification because many relevant linear temporal properties fall into this category.

Definition 2.1.1 (ω -Regular Expression). An ω -regular expression G over the alphabet Σ has the form

$$G = E_1.F_1^\omega + \dots + E_n.F_n^\omega$$

where $n \geq 1$ and $E_1, \dots, E_n, F_1, \dots, F_n$ are regular expressions over Σ such that $\varepsilon \notin \mathcal{L}(F_i)$, for all $1 \leq i \leq n$ and $+$ is the union operator.

The semantics of the ω -regular expression G is a language of infinite words, defined by

$$\mathcal{L}_\omega(G) = \mathcal{L}(E_1).\mathcal{L}(F_1)^\omega \cup \dots \cup \mathcal{L}(E_n).\mathcal{L}(F_n)^\omega$$

where $\mathcal{L}(E) \subseteq \Sigma^*$ denotes the language (of finite words) induced by the regular expression E . For a more detailed presentation of regular expressions we refer to [HMU01].

Definition 2.1.2 (ω -Regular Language). A language $\mathcal{L} \subseteq \Sigma^\omega$ is called ω -regular if $\mathcal{L} = \mathcal{L}_\omega(G)$ for some ω -regular expression G over Σ .

Recognizing ω -regular languages, *i.e.*, deciding if a word σ is part of the ω -regular language \mathcal{L} , requires to check all the infinite symbols of the input word. A way to achieve this goal is with ω -automata, namely variants of nondeterministic finite-state automata with a special acceptance criteria for infinite words. In the literature, different kinds of ω -automata have been proposed. Here we focus on a specific type of ω -automata called Nondeterministic Büchi Automata.

Definition 2.1.3 (Nondeterministic Büchi Automata). A non deterministic Büchi Automata (NBA) \mathcal{A} is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where:

- Q is a finite set of states,
- Σ is a finite set of symbols,
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function
- $Q_0 \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is a set of accepting states, called acceptance set.

Definition 2.1.4 (Run). A run for an infinite word $\sigma = A_0A_1A_2\dots \in \Sigma^\omega$ denotes an infinite sequence $\varrho = q_0q_1q_2\dots$ of states in \mathcal{A} such that $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, A_i)$ for $i \geq 0$, with $q_i, q_{i+1} \in Q$.

Notice that each run ϱ in a NBA *induces* a corresponding word $\sigma \in \Sigma^\omega$.

Definition 2.1.5 (Accepting run). A run ϱ is accepting if there exist $q_i \in F$ such that q_i occurs infinitely many times in ϱ .

A different kind of automaton, called monitor, is designed to follow the execution of a system and move accordingly. An error is detected when the monitor cannot move, i.e., the system has performed some action, or reached some state that it was not meant to be. In other words, the monitor reports an error whenever a bad prefix of the language occurs.

Definition 2.1.6 (Monitor). A monitor \mathcal{M} is a tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0)$ where:

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function
- $q_0 \in Q$ is the initial state

2.2 Linear Temporal Logic

In this section we introduce the syntax and semantics of Linear Temporal Logic (LTL), a logical formalism that extend the standard propositional logic with temporal operators, and we present the concept of LTL satisfiability.

2.2.1 Syntax

Linear temporal logic (LTL) [Pnu77] formulas are built on a finite set $Prop$ of atomic propositions as follows:

$$\phi = p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\phi_1 \mid \phi_1 \mathcal{U}\phi_2 \mid (\phi)$$

where $p \in Prop$, ϕ, ϕ_1, ϕ_2 are LTL formulas, \mathcal{X} is the “next” operator and \mathcal{U} is the “until” operator. In the following, unless specified otherwise using

parentheses, unary operators have higher precedence than binary operators. We consider other Boolean connectives like “ \wedge ” and “ \rightarrow ” with the usual meaning, and we abbreviate $p \vee \neg p$ as \top , $p \wedge \neg p$ as \perp . We also take into account additional temporal operators that can be derived as follow: $\diamond \phi$ (“eventually”) to denote $\top \mathcal{U} \phi$, $\square \phi$ (“always”) to denote $\neg \diamond \neg \phi$, and $\alpha \mathcal{W} \beta$ (“weak until”) defined as $\square \alpha \vee (\alpha \mathcal{U} \beta)$.

2.2.2 Semantics

An LTL formula can be interpreted either over words or over a computation. This lead to two equivalent definitions of the LTL semantics, that we report below.

Definition 2.2.1 (Semantics Over Words). Let ϕ be an LTL formula over the set AP and let $\sigma = A_0 A_1 A_2 \dots$ be an infinite word over (2^{AP}) . We define the relation “ \models ” between σ and ϕ as the smallest relation with the following properties:

1. $\sigma \models true$
2. $\sigma \models a$ iff $a \in \sigma[0]$
3. $\sigma \models \phi_1 \wedge \phi_2$ iff $\sigma \models \phi_1$ and $\sigma \models \phi_2$
4. $\sigma \models \neg \phi$ iff $\sigma \not\models \phi$
5. $\sigma \models \mathcal{X} \phi$ iff $\sigma[1\dots] \models \phi$
6. $\sigma \models \phi_1 \mathcal{U} \phi_2$ iff $\exists j \geq 0$ such that $\sigma[j\dots] = A_j A_{j+1} \dots \models \phi_2$ and $\sigma[i\dots] \models \phi_1$
 $\forall 0 \leq i < j$

Definition 2.2.1 allows to interpret the semantics of LTL formula ϕ as the language $Words(\phi)$ that contains all infinite words over the alphabet 2^{AP} that satisfy ϕ .

Definition 2.2.2 (Semantics Over Computations). a *computation*, i.e., a function $\pi : \mathbb{N} \rightarrow 2^{AP}$ which assigns truth values to the elements of AP at each time instant (natural number). For a computation π and a time instant $i \in \mathbb{N}$:

- $\pi, i \models p$ for $p \in AP$ iff $p \in \pi(i)$

- $\pi, i \models \neg\alpha$ iff $\pi, i \not\models \alpha$
- $\pi, i \models (\alpha \wedge \beta)$ iff $\pi, i \models \alpha$ and $\pi, i \models \beta$
- $\pi, i \models \mathcal{X}\alpha$ iff $\pi, i + 1 \models \alpha$
- $\pi, i \models \mathcal{U}\alpha\beta$ iff for some $j \geq i$, we have $\pi, j \models \beta$ and for all $k, i \leq k < j$ we have $\pi, k \models \alpha$

We say that π *satisfies* a formula ϕ , denoted $\pi \models \phi$, iff $\pi, 0 \models \phi$. If $\pi \models \phi$ for every π , then ϕ is *valid* and we write $\models \phi$.

Definition 2.2.3 (Semantics Over Transition Systems). A transition system \mathcal{M} satisfy a formula ϕ iff all the computations $\Pi_{\mathcal{M}}$ generated from \mathcal{M} satisfy ϕ . Formally:

$$\mathcal{M} \models \phi \stackrel{def}{=} \forall \pi \in \Pi_{\mathcal{M}} : \pi \models \phi$$

2.3 Sanity Checking

Writing formal specifications is a difficult task, which is prone to errors just as developing the system. However, some automatic activities can be performed to check the sanity of requirements. Some of these activities are, for example, vacuity checking, completeness checking and consistency checking [BBB⁺16]. A specification is satisfied vacuously in a model if it is satisfied in some non-interesting way; borrowing the example from [RV10], the LTL specification “every request is eventually followed by a grant” is satisfied vacuously in a model with no requests. Vacuity checking can also be performed without the need of a model, and in this case it is known as *inherent* vacuity checking [FKSFV08, RV11]. Completeness checking is equivalent to verify if the set of requirements covers all reasonable behaviors of a system. Completeness can be checked in combination with a system model, but in [BBB⁺16] a proposal for model-free completeness checking is also presented. Finally, requirements consistency is about checking whether a *real* system can be implemented from a given set of requirements. Therefore, two types of check [RV11] are possible: (i) *realizability*, i.e., testing whether there is an *open* system that satisfies all the properties in the set [PR89], and (ii) *satisfiability*, i.e., testing whether there is a *closed*

system that satisfies all the properties in the set. Satisfiability checking ensures that the behavioral description of a system is internally consistent and neither over- or under-constrained. If a formal property is valid, namely always true, or unsatisfiable, *i.e.* always false, than this is certainly due to an error. Even if the satisfiability test is weaker than the realizability test, its importance is widely recognized [RV11].

2.3.1 LTL satisfiability

Recall that a logical formula ϕ is valid iff its negation $\neg\phi$ is not satisfiable. LTL satisfiability checking, for a given LTL formula ϕ , consists in determining if there exists at least a model for which ϕ holds. In other words, do we have $Words(\phi) \neq \emptyset$? Among various approaches to decide LTL satisfiability, reduction to model checking was proposed in [RV07] to check the consistency of requirements expressed as LTL formulas. Given a formula ϕ over a set AP of atomic propositions, a *universal* model M can be constructed, namely a model that generates all possible computations over its atomic propositions. Intuitively, a universal model encodes all the possible computations over AP as (infinite) traces, and therefore ϕ is satisfiable precisely when M does not satisfy $\neg\phi$ (see Definition 2.2.3).

In [RV11] a first improvement over this basic strategy is presented together with the tool PANDA¹. Similarly, [CRST07] employs a model checker on propositional abstractions of the problem in order to determine the satisfiability of temporal properties. In [LZP⁺13] an algorithm based on automata construction is proposed to enhance performances even further — the approach is implemented in a tool called AALTA. Further studies along this direction include [LYP⁺14] and [LPZ⁺13]. In the latter, a portfolio LTL satisfiability solvers called POLSAT is proposed to run different techniques in parallel and return the result of the first one terminating successfully.

¹<https://ti.arc.nasa.gov/m/profile/kyrozier/PANDA/PANDA.html>

2.4 Property Specification Patterns

To deal with the problem of requirements formalization, a common solution adopted is the use of Property Specification Patterns (PSPs for short), first introduced by [DAC99]. PSPs are a collection of parameterizable, high-level, formalism-independent specification abstractions usually based on a restricted English grammar.

They provide an easy way to express properties of a system with an English-like syntax, while preserving a well-defined semantic and provide expressions of such behaviors in a range of common formalisms. Since the original work of [DAC99], a considerable number of property specification pattern systems have been proposed, grounding on different logics. PSPs have successfully been applied in many domains, such as automotive [PMHP12], aviation [EKN⁺12] and banking [BGPS12].

There are three general types of PSPs: Qualitative, Real-Time and Probabilistic specification patterns. These types aim at representing different aspects of the system and are based on logics with different properties. [AGL⁺15] presented a unified catalog that combines all the qualitative, real-time and probabilistic specification patterns in a single framework, aligning the English grammar and identifying new patterns. They gathered together the translation of PSPs in different logics on a dedicated website² and provide a tool to guide system engineers in the translation process.

In this work, we only focus on qualitative patterns with LTL translation and we invite the reader to refer to [AGL⁺15] for a more extensive discussion.

An example of a PSP is given in Figure 2.1 — with some parts omitted for sake of readability.³ A pattern is comprised of a *Name* (Response in Figure 2.1), an (informal) statement describing the behavior captured by the pattern, and a (structured English) statement. The context-free grammar introduced in [KC05] to express qualitative requirements is depicted in Figure 2.2.

The LTL mappings corresponding to different declinations of the pattern are also given, where capital letters (P , S , T , etc.) stands for Boolean states/events.

²<http://ps-patterns.wikidot.com>

³We omitted aspects which are not relevant for our work, e.g., translations to other logics like CTL [DAC99].

Response
<p>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.</p> <hr/>
<p>Structured English Grammar <i>⟨scope⟩, it is always the case that if P holds, then S eventually holds.</i></p> <hr/>
<p>LTL Mappings</p> <p style="padding-left: 40px;">Globally, it is always the case that if P holds, then S eventually holds.</p> $\square (P \rightarrow \diamond S)$ <hr style="border-top: 3px double #000;"/> <p style="padding-left: 40px;">Before R, it is always the case that if P holds, then S eventually holds.</p> $\diamond R \rightarrow (P \rightarrow (\bar{R} \mathcal{U} (S \wedge \bar{R}))) \mathcal{U} R$ <hr style="border-top: 3px double #000;"/> <p style="padding-left: 40px;">After Q, it is always the case that if P holds, then S eventually holds.</p> $\square (Q \rightarrow \square (P \rightarrow \diamond S))$ <hr style="border-top: 3px double #000;"/> <p style="padding-left: 40px;">Between Q and R, it is always the case that if P holds, then S eventually holds.</p> $\square ((Q \wedge \bar{R} \wedge \diamond R) \rightarrow (P \rightarrow (\bar{R} \mathcal{U} (S \wedge \bar{R})))) \mathcal{U} R$ <hr style="border-top: 3px double #000;"/> <p style="padding-left: 40px;">After Q until R, it is always the case that if P holds, then S eventually holds.</p> $\square (Q \wedge \bar{R} \rightarrow ((P \rightarrow (\bar{R} \mathcal{U} (S \wedge \bar{R})))) \mathcal{W} R)$ <hr/>
<p>Example</p> <p><i>Globally, it is always the case that if object_detected holds, then moving_to_target eventually holds.</i></p>

Figure 2.1: Response Pattern ($\bar{\alpha}$ stands for $\neg\alpha$).

Start	1: property	::= <i>scope</i> “,” <i>specification</i> “.”
Scope	2: scope	::= “Globally” “Before ” R “After ” Q “Between ” Q “ and ” R “After ” Q “ until ” R
General	3: specification	::= <i>qualitativeType</i> <i>realtimeType</i>
Qualitative	4: qualitativeType	::= <i>occurrenceCategory</i> <i>orderCategory</i>
	5: occurrenceCategory	::= <i>absencePattern</i> <i>universalityPattern</i> <i>existencePattern</i> <i>boundedExistencePattern</i>
	6: absencePattern	::= “it is never the case that ” P “ holds”
	7: universalityPattern	::= “it is always the case that ” P “ holds”
	8: existencePattern	::= P “ eventually holds”
	9: boundedExistencePattern	::= “transitions to states in which ” P “ holds occur at most twice”
	10: orderCategory	::= “it is always the case that if ” P “ holds” (<i>precedencePattern</i> <i>precedenceChainPattern1-2</i> <i>precedenceChainPattern2-1</i> <i>responsePattern</i> <i>responseChainPattern1-2</i> <i>responseChainPattern2-1</i> <i>constrainedChainPattern1-2</i>)
	11: precedencePattern	::= “, then ” S “ previously held”
	12: precedenceChainPattern1-2	::= “ and is succeeded by ” S “, then ” T “ previously held”
	13: precedenceChainPattern2-1	::= “, then ” S “ previously held and was preceded by ” T
	14: responsePattern	::= “, then ” S “ eventually holds”
	15: responseChainPattern1-2	::= “, then ” S “ eventually holds and is succeeded by ” T
	16: responseChainPattern2-1	::= “ and is succeeded by ” S “, then ” T “ eventually holds after ” S
	17: constrainedChainPattern1-2	::= “, then ” S “ eventually holds and is succeeded by ” T “, where ” Z “ does not hold between ” S “ and ” T

Figure 2.2: Structured Natural Language Specification

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*, to span the entire scope execution; *Before*, to span execution up to a state/event; *After*, to span execution after a state/event; *Between*, to cover the part of execution from one state/event to another one; *After-until*, where the first part of the pattern continues even if the second state/event never happens. For state-delimited scopes, the interval in which the property is evaluated is closed at the left and open at the right end. The *body* of a pattern, describes the behavior that we want to specify. In [DAC99], bodies are categorized in *occurrence* and *order* patterns. Occurrence patterns require states/events to occur or not to occur. Examples of such bodies are *Absence*, where a given state/event must not occur within a scope, and its opposite *Existence*. Order patterns constrain the order of the states/events. Examples of such patterns are *Precedence*, where a state/event must always precede another state/event, and *Response*, where a state/event must always be followed by another state/event within the scope. Moreover, we included the *Invariant* pattern introduced in [PH12], and dictating that a state/event must occur whenever another state/event occurs. Combining scopes and bodies we can construct 55 different types of patterns.

2.5 Minimal Unsatisfiable Cores

Usually, inconsistency in a set of requirements is best explained in terms of minimal subsets of requirements exposing the core issues within the specification. The literature does not provide a consistent naming of such cores, and the terms *minimal inconsistency subset* (*MIS*) [Ben17], *minimal unsatisfiable subset* [BMS12] (*MUS*), *minimal unsatisfiable core* [LS08] (*MUC*), and also High-Level MUC (HLMUC) [Nad10] are introduced to refer to the same concept — in the following, and throughout the paper, we denote with MUC a minimal set of inconsistent requirements. Algorithms for finding MUCs can be divided in two basic groups: (i) those focusing on the extraction of a single MUC, and (ii) those focusing on the extraction of all MUCs. These techniques can be further divided into domain specific, i.e., targeting specific domains such as propositional satisfiability [BMS11], and general purpose, i.e., high level algorithms that can be applied to any domain provided that a consistency checking procedure exists for that domain [Dra89]. The most basic general purpose solution for computing a single MUC out of a set of logical constraints, consists of iteratively removing constraints from an initial set. At each step, the set of constraints represents an over-approximation of the MUC. This solution is referred to as the *deletion-based* approach [Dra89, CD91, BDTW93, DGHP09]. Given a set R of n constraints, the deletion-based approach calls the consistency checker exactly n times. When examining the i -th constraint, if $R \setminus \{r_i\}$ remains inconsistent, then there is a MUC that does not include r_i , and r_i can be removed; otherwise r_i must be part of the MUC. This approach is guaranteed to produce a set $M \subseteq R$ such that, if a single requirement is eliminated from M , then M becomes consistent. However, the approach does not guarantee that another MUC $M' \subseteq R$ such that $|M'| \leq |M|$ may not exist. Extraction of all MUCs has received some attention, also because retrieving MUCs of minimal size can be done simply by enumerating all MUCs. Finding all the MUCs of a set of constraints R in a naive way amounts to check the consistency of all the elements of the power set 2^R , but this is clearly untenable in real world applications. In [LM13], the power set of requirements is implicitly considered as follows. Given a set of requirements R , if $R' \subseteq R$ is inconsistent, every $R'' \supset R'$ and $R'' \subset R$ is also inconsistent. Furthermore if $R' \subseteq R$ is consistent,

every $R'' \subset R'$ is consistent too. This algorithm can be modified to find a single MUC by stopping it to the first MUC extracted. In [BBCB16], the algorithm of [LM13] is improved by constructing some chains between elements of the power set of requirements. The chains implement the notion of super/subset between set of requirements. The main difference is that the search for MUCs is done in a depth first fashion in [BBCB16], whereas it is done in breadth first way in [LM13].

2.6 Conformance Testing

Conformance is a relation between the observable behavior of a System Under Test (SUT) and that of its specification, or model. Therefore, *conformance testing* consists in testing the implementation of a system against that system's specification. When the specification is given as a precise formal model of the system being developed, we use the more specific term *model-based testing*. A model is an abstraction of a SUT or of its environment, or both. In model-based testing, a model of the SUT is, among other things, used to determine the expected output. An important distinction that it is usually made is between white- and black-box testing. The former takes into account knowledge of the inner structure of the SUT, the latter does not.

In this work, we focus our attention to systems that can be modeled as Mealy machines. Mealy machines are finite state machines and allow to model both inputs and outputs as part of their behavior. Therefore, they are a suitable abstraction to model reactive systems, *i.e.*, systems that maintain an ongoing interaction with the environment and react to external stimuli.

Definition 2.6.1 (Mealy machine). A Mealy machine is a tuple $\mathcal{M} = (S, s_0, I, O, \tau)$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- I is a finite set of symbols called input alphabet,
- O is a finite set of symbols called output alphabet,

- $\tau : S \times I \rightarrow S \times O$ is a transition function mapping pairs of states and input symbols to the corresponding pairs of states and output symbols.

Given an infinite word $i_0 i_1 \dots \in (2^I)^\omega$ over the inputs, \mathcal{M} can be traversed applying the transition function $\tau(s_j, i_j) = (o_j, s_{j+1})$ for every $j \geq 0$, with $s_j, s_{j+1} \in S$, $i_j \in I$, $o_j \in O$ and s_0 being the initial state. The application of τ for every input i_j , starting from s_0 , produces an infinite trace $(s_0 \cup i_0 \cup o_0)(s_1 \cup i_1 \cup o_1) \dots \in (2^{S \cup I \cup O})^\omega$. The projection of a trace to the atomic propositions is a path $w \in (2^{I \cup O})^\omega$. We denote the set of all paths generated by a Mealy machine \mathcal{M} as $Paths(\mathcal{M})$. A Mealy machine \mathcal{M} realizes an LTL formula φ if $Paths(\mathcal{M}) \subseteq Words(\varphi)$.

Chapter 3

State of the Art and Related Work

In this Chapter we summarize the state of the art and we discuss the research works that are most closely related to the contributions presented in this thesis. In particular, in Section 3.1 we review the literature regarding the formalization and analysis of requirements for Cyber-Physical Systems, while in Section 3.2 we discuss how state-of-the-art techniques for automatic testing from LTL specifications compare with our work.

3.1 Requirements Formalization and Analysis

3.1.1 Formalization and Consistency Checking

Regarding the automatic formalization and analysis of requirements, several approaches have been proposed. In [LMG11] the framework *Property Specification Pattern Wizard (PSP-Wizard)* is presented. Its purpose is the machine-assisted definition of temporal formulas capturing pattern-based system properties. PSP-Wizard offers a translation into LTL of the patterns encoded in the tool, but it is meant to aid specification, rather than support consistency checking, and it cannot deal with numerical signals.

In [KC05], an extension is presented to deal with real-time specifications, together with mappings to different real-time logics. Even if this work is not

directly connected with ours, it is worth mentioning it since their structured English grammar for patterns is at the basis of our formalism.

The work in [KC05] also provided inspiration to a recent set of works [DHF16, DHF15] about a tool, called VI-Spec, to assist the analyst in the elicitation and debugging of formal specifications. VI-Spec lets the user specify requirements through a graphical user interface, translates them to MITL formulas and then supports debugging of the specification using run-time verification techniques. VI-Spec embodies an approach similar to ours to deal with numerical signals by translating inequalities to sets of Boolean variables. However, VI-Spec differs from our work in several aspects, most notably the fact that it performs debugging rather than consistency checking, so the behavior of each signal over time must be known. Also, VI-Spec handles only inequalities and does not deal with sets of requirements written using PSPs.

In [FLM⁺04], the authors present a framework that supports the formal verification of early requirements expressed in Formal Tropos, a specification language that consists of a sequence of class declarations such as actors, goals, and dependencies. Similarly to our approach, they map their high-level specifications into LTL constraints that are checked with NuSMV[CCG⁺02]. Likewise, the work presented in [CRST11] aims at formalizing and validate requirements represented in a domain specific formalism applied in an industrial project. The formalism includes class diagrams with fragments of first order logic and temporal logic operators, which allows to reason about object models and their temporal evolution.

Alike the approach we describe in Chapter 4, where we define $LTL(\mathcal{D}_C)$ as an extension of the LTL that supports atomic numerical constraints and provides an encoding to reduce $LTL(\mathcal{D}_C)$ formulas to standard LTL, many recent works extended the LTL expressiveness in different directions. In [CRT09, CRT15], in order to deal with requirements of hybrid systems, where continuous and discrete variables are combined, the authors propose the HRELTL logic, a combination of temporal logic with regular expressions and both discrete and continuous variables, and demonstrate how to check the satisfiability of requirements expressed in the polynomial fragment of HRELTL. In [CGM⁺20], instead, they propose a first-order LTL logic called LTL-EF, with the “at next” and “at last” operators.

They also provide a reduction to equisatisfiable discrete-time formulas and an encoding for SMT-based model checking.

3.1.2 Inconsistency Explanation

The problem of finding minimal unsatisfiable subsets, or *inconsistency explanations*, has been the subject of some attention, e.g., in propositional satisfiability and constraint programming. The algorithms to be found in the literature can be either domain specific — see, e.g., [BMS12, MSL11, LS08] — or domain independent — see, e.g., [Jun01]. They can be further divided into algorithms that find only one inconsistent subset or all inconsistent subsets.

In particular, we are interested in searching only one minimal unsatisfiable subset of LTL formulas. To this end, some special purpose algorithms have been recently proposed:

- in [CRST07], the authors perform extraction of UCs for PSL to accelerate a PSL satisfiability solver by performing Boolean abstraction;
- [CRST08] introduces the notion of unrealizable cores that have been proposed to help debugging unrealizable specifications. The algorithm is based on a deletion-based strategy for Generalized Reactivity [PPS06] specifications, a subset of LTL;
- in [AGTW11], the PROCUNE tool is presented, which uses a tableau-based solver to obtain an initial subset from an unsatisfiable set of LTL and then applies deletion-based minimization to that subset.
- PLTL-MUP [GHST13], built upon the PLTL model checker, uses a method based on Ordered Binary Decision Diagrams to find inconsistent subsets;
- finally, more recently a new algorithm based on resolution graphs has been implemented in TRP++UC [Sch12, Sch16a, Sch16b] to extract minimal unsatisfiable subsets of requirements.

However, our work differ from all these contributions because it is independent from any specific model checker or satisfiability checker implementation.

In fact, our algorithm only needs to know if a given set of requirements is satisfiable or not. In this way, we can easily exploits the most recent state-of-the-art satisfiability solvers.

3.2 Automatic Testing from LTL specification

In the literature, many techniques for test generation with LTL specifications follow a model-based strategy, as depicted in Figure 3.1. The general idea is to generate a set of trap properties, *i.e.*, LTL formulae designed to expose a specific behavior of the model, and use a model checker to find counterexamples, which are then interpreted as test cases. The main challenge is to force the model checker to systematically create sets of such counterexamples (see [FWA09] for a survey of such techniques). The main disadvantage of this strategy is that it

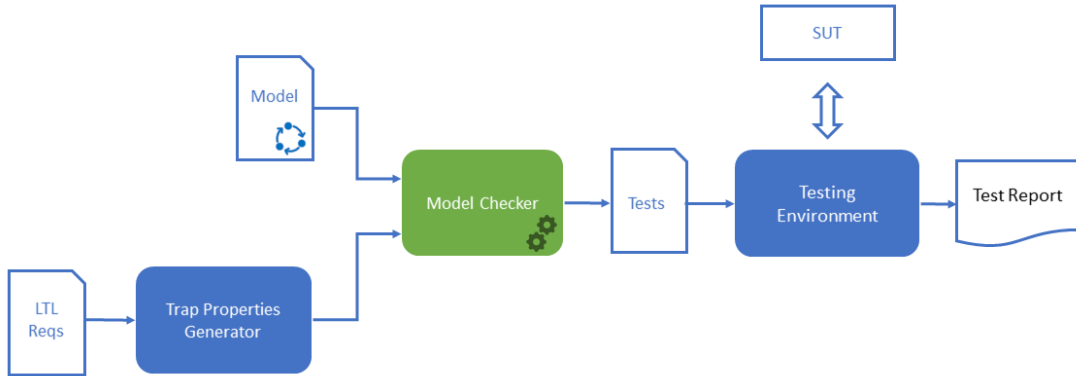


Figure 3.1: Model-Based test generation with LTL requirements

requires a formal (abstract) model of the system under test (SUT), which is not always available.

Techniques aimed at automated test generation for black-box reactive systems relying on formal models of the specifications have been explored — see, e.g., [KT04, BBD19, KGHS98, SEG00, JJ05] — and they seem more promising than classical techniques when both efficiency of test generation and effectiveness in covering the specification are considered.

Runtime verification [BLS11] techniques can be seen as a form of *oracle-based testing* [BGM91]: each test is executed on the system implementation and the test oracle, *i.e.*, the monitor in runtime verification jargon, observes the system

and checks whether its executions are behaviors allowed by the specification or not. Following this stream of research, a technique based on the use of monitors as test oracles is proposed in [AGR13] for online testing of Java classes. The key idea is to exploit a monitor derived from LTL specifications to check conformance of the system to stated requirements, with a focus on safety properties. Their approach can test for safety properties (“something bad will never happen”), but it does not deal with liveness properties (“something good will happen infinitely often”). While liveness properties are not amenable to monitoring on finite executions, their proper subclass of co-safety properties (“something good will happen”) consists of formulas that can be monitored on finite traces and that we wish to consider in our work when testing a system for conformance.

Another work related to ours is presented in [KT04] where the authors describe a methodology for specification based testing of black-box systems. They assume that the specification of the system is given as a non-blocking input/output timed automaton, and the system itself — whose model need not to be known — is also a timed automaton. The two main differences between their methodology and ours are (i) the capability of dealing with real-time requirements and (ii) the form of the specification: ours is “declarative”, in the form of a set of LTL requirements, whereas theirs is “operational” in the form of an automaton. We thus incur into one additional step, i.e., extracting an automaton from the requirements, after which the two methodologies proceed in a similar way. However, given the different form and expressivity of the requirements, a direct comparison is not easily feasible, and might be even misleading.

More recently in [BBD19], another approach based on timed automata to specify input signals constraints has been proposed. Also this approach bears some similarity with ours and with that of [KT04], but in our opinion it is not directly comparable, at least in the settings that we consider for our experimental analysis.

Other research which is closely related to ours appears in a series of papers [TSL04, ZT16, ZT15] where the authors present a test-case generation methodology that (i) translates LTL requirements into Generalized Büchi Automata, (ii) builds trap properties from them — using different criteria — and (iii) performs model checking of negated trap properties against the system

model in order to extract test cases. The main difference with our work is that such methodology relies on a model of the system under testing, a model that must be verified against the system specification. Failing to do so, may generate conflicting tests, i.e., a test which fulfills a requirement, and violates another.

To the extent of our knowledge there is no other recent work on formally-grounded methods for requirement based testing, while there is some not-so-recent work mentioning conformance testing to specification, such as, for example [KGHS98, SEG00, JJ05]. However, in these works specifications are mostly “operational” in the form, e.g., of finite state machines and thus a direct comparison with our methodology is not possible.

Chapter 4

Consistency Checking

In the context of safety- and security-critical Cyber-Physical Systems (CPSs), checking the sanity of functional requirements is an important, yet challenging task. Requirements written in natural language call for time-consuming and error-prone manual reviews, whereas enabling automated sanity verification often requires overburdening formalizations. Given the increasing pervasiveness of CPSs, their stringent time-to-market and product budget constraints, practical solutions to enable automated verification of requirements are in order. Property Specification Patterns (PSPs) [DAC99] offer a viable path towards this target. PSPs are a collection of parameterizable, high-level, formalism-independent specification abstractions, originally developed to capture recurring solutions to the needs of requirement engineering. Each pattern can be directly encoded in a formal specification language, such as Linear Temporal Logic (LTL) [PM92], Computational Tree Logic (CTL) [CES86], or Graphical Interval Logic (GIL) [DKM⁺94]. Because of their features, PSPs may ease the burden of formalizing requirements, yet enable verification of their sanity using current state-of-the-art automated reasoning tools — see, e.g., [LPZ⁺13, LZPV15, Sch98, CCG⁺02, HK03].

In this work, we restrict our attention to sanity checking as satisfiability checking. We speak of (internal) consistency of requirements written using PSPs having in mind that PSPs can be translated to LTL formulas whose satisfiability can be checked using methods and tools available in the literature.

The original formulation of PSPs caters for temporal structure over Boolean variables, but for most practical applications such expressiveness is too restricted. This is the case of the embedded controller for robotic manipulators that is under development in the context of the EU project CERBERO [MPM⁺17]¹ and provides the main motivation for this work. As an example, consider the following statement: “*The angle of joint1 shall never be greater than 170 degrees*”. This requirement imposes a safety threshold related to some joint of the manipulator (*joint1*) with respect to physically-realizable poses, yet it cannot be expressed as a PSP unless we add atomic numerical assertions in some constraint system \mathcal{D} . We call Constraint PSP, or PSP(\mathcal{D}) for short, a pattern which has the same structure of a PSP, but contains atomic propositions from \mathcal{D} . For instance, using PSP($\mathbb{R}, <, =$) we can rewrite the above requirement as a *universality* pattern: “*Globally, it is always the case that $\theta_1 < 170$ holds*”, where θ_1 is the numerical signal (variable) for the angle of *joint1*. In principle, automated reasoning about Constraint PSPs can be performed in Constraint Linear Temporal Logic, i.e., LTL extended with atomic assertions from a constraint system [DD07]: in our example above, the encoding would be simply $\Box(\theta_1 < 170)$. Unfortunately, this approach does not always lend itself to a practical solution, because Constraint Linear Temporal Logic is undecidable in general [CC00]. Restrictions on \mathcal{D} may restore decidability [DD07], but they introduce limitations in the expressiveness of the corresponding PSPs. We propose a solution which ensures that automated verification of consistency is feasible, yet enables PSPs mixing both Boolean variables and (constrained) numerical signals. Our approach enables us to capture many specifications of practical interest, and to pick a verification procedure from the relatively large pool of automated reasoning systems currently available for LTL. In particular, we restrict our attention to a constraint systems of the form $(\mathbb{R}, <, =)$, and atomic propositions of the form $x < c$ or $x = c$, where $x \in \mathbb{R}$ is a variable and $c \in \mathbb{R}$ is a constant value. In the following, we write \mathcal{D}_C to denote such restriction.

Knowing that a set of requirements written with PSPs(\mathcal{D}_C) is (in)consistent is only the first step in writing a correct specification. In case of inconsistent

¹Cross-layer model-based framework for multi-objective design of reconfigurable systems in uncertain hybrid environments — <http://www.cerbero-h2020.eu/>

requirements, obtaining a *minimal* set of such requirements would be desirable to help designers avoid manual checks to pinpoint problems in a specification.

Since for practical reasons in requirement engineering it is better to have a quick turnaround time rather than a complete answer, we present a method to look for inconsistencies in an incremental fashion, i.e., stopping the search once at least one (minimal) inconsistency subset is found. In particular, given a set of inconsistent requirements, we extract a minimal (irreducible) subset from them that it is still inconsistent. The set is guaranteed to be minimal in the sense that, if we remove one of the elements, the remaining set becomes consistent.

Overall, our contribution can be summarized as follows:

- We extend basic PSPs over the constraint system \mathcal{D}_C .
- We provide an encoding from any $\text{PSP}(\mathcal{D}_C)$ into a corresponding LTL formula.
- We propose algorithms devoted to extract minimal subsets of inconsistent requirements, and we implement them in the tool mentioned above.
- We provide an open-source tool, described in details in Chapter 6, that implements the encoding and algorithms proposed to automatically analyze requirements expressed as $\text{PSPs}(\mathcal{D}_C)$.
- We implement a generator of artificial requirements expressed as $\text{PSPs}(\mathcal{D}_C)$; the generator takes a set of parameters in input and emits a collection of PSPs according to a parameterized probability model.
- Using our generator, we run an extensive experimental evaluation aimed at understanding (i) which automated reasoning tool is best at handling set of requirements as $\text{PSPs}(\mathcal{D}_C)$, and (ii) whether our approach is scalable.
- Finally, we analyze the specification of the embedded controller to be dealt with in the context of CERBERO project, experimenting also with the addition of faulty requirements.

Verification and inconsistency explanation of requirements written in $\text{PSP}(\mathcal{D}_C)$ are carried out using tools and techniques available in the literature [RV10, RV11, LPZ⁺13]. With those, we demonstrate the scalability of our approach

by checking the consistency of up to 1920 requirements, featuring 160 variables and up to 8 different constant values appearing in atomic assertions, within less than 500 CPU seconds. A total of 75 requirements about the embedded controller for the CERBERO project is checked in a matter of seconds, even without resorting to the best tool among those we consider.

4.1 Constraint Property Specification Patterns

Let us start by defining a *constraint system* \mathcal{D} as a tuple $\mathcal{D} = (D, R_1, \dots, R_n, \mathcal{I})$, where D is a non-empty set called *domain*, and each R_i is a predicate symbol of arity a_i , with $\mathcal{I}(R_i) \subseteq D^{a_i}$ being its interpretation. Given a finite set of variables X and a finite set of constants C such that $C \cap X = \emptyset$, a *term* is a member of the set $T = C \cup X$; an (atomic) \mathcal{D} -*constraint* over a set of terms is of the form $R_i(t_1, \dots, t_{a_i})$ for some $1 \leq i \leq n$ and $t_j \in T$ for all $1 \leq j \leq a_i$ which we call *constraint* when \mathcal{D} is understood from the context. We define *linear temporal logic modulo constraints* — $\text{LTL}(\mathcal{D})$ for short — as an extension of LTL with additional atomic constraints. Given a set of Boolean propositions AP , a constraint system $\mathcal{D} = (D, R_1, \dots, R_n, \mathcal{I})$, and a set of terms $T = C \cup X$, an $\text{LTL}(\mathcal{D})$ formula is defined as:

$$\phi = p \mid R_i(t_1, \dots, t_{a_i}) \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\phi_1 \mid \phi_1 \mathcal{U}\phi_2 \mid (\phi)$$

where $p \in AP$, ϕ, ϕ_1, ϕ_2 are $\text{LTL}(\mathcal{D})$ formulas, and $R_i(\cdot)$ with $1 \leq i \leq n$ is an atomic \mathcal{D} -constraint. Additional Boolean and temporal operators are defined as in LTL with the same intended meaning. Notice that the set of $\text{LTL}(\mathcal{D})$ formulas is a (strict) subset of those in constraint linear temporal logic — $\text{CLTL}(\mathcal{D})$ for short — as defined, e.g., in [DD07]. $\text{LTL}(\mathcal{D})$ formulas are interpreted over computations of the form $\pi : \mathbb{N} \rightarrow 2^{AP}$ plus additional *evaluations* of the form $\nu : T \times \mathbb{N} \rightarrow D$ such that, for all $i \in \mathbb{N}$, $\nu(c, i) = \nu(c) \in D$ for all $c \in C$, whereas $\nu(x, i) \in D$ for all $x \in X$. In words, the function ν associates to constants $c \in C$ a value $\nu(c)$ that does not change in time, and to variables $x \in X$ a value $\nu(x, i)$ that possibly changes at each time instant $i \in \mathbb{N}$. LTL semantics is extended to $\text{LTL}(\mathcal{D})$ by handling constraints:

$$\pi, \nu, j \models_{\mathcal{D}} R_i(t_1, \dots, t_{a_i}) \text{ iff } (\nu(t_1, j), \dots, \nu(t_{a_i}, j)) \in \mathcal{I}(R_i)$$

We say that π and ν *satisfy* a formula ϕ , denoted $\pi, \nu \models_{\mathcal{D}} \phi$, iff $\pi, \nu, 0 \models \phi$. A formula ϕ is *satisfiable* as long as there exist a computation π and a valuation ν such that $\pi, \nu \models_{\mathcal{D}} \phi$. We further restrict our attention to the constraint system $D_C = (\mathbb{R}, <, =, \mathcal{I})$, with atomic constraints of the form $x < c$ and $x = c$, where c is a constant corresponding to some real number — hereafter we abuse notation and write $c \in \mathbb{R}$ instead of $\nu(c) \in \mathbb{R}$ — and the interpretation \mathcal{I} of the predicates “<” and “=” is the usual one. For example, $\diamond(x < 100)$ is a valid LTL(\mathcal{D}_C) formula, while $\diamond(x < y)$ can be expressed in LTL(\mathcal{D}) but not in LTL(\mathcal{D}_C). Similarly, the formula $\diamond(x < \mathcal{X}y)$ can be expressed in CLTL(\mathcal{D}) but not in LTL(\mathcal{D}).

While CLTL(\mathcal{D}) is undecidable in general [DD07, CC00], LTL(\mathcal{D}_C) is decidable since, as we show in this paper, it can be reduced to LTL satisfiability.

We introduce the concept of *constraint property specification pattern*, denoted PSP(\mathcal{D}), to deal with specifications containing Boolean variables as well as atoms from a constraint system \mathcal{D} . In particular, a PSP(\mathcal{D}_C) features only Boolean atoms and atomic constraints of the form $x < c$ or $x = c$ ($c \in \mathbb{R}$). For example, the requirement:

The angle of joint1 shall never be greater than 170 degrees

can be re-written as a PSP(\mathcal{D}_C):

Globally, it is always the case that $\theta_1 < 170$

where $\theta_1 \in \mathbb{R}$ is the variable associated to the angle of *joint1* and 170 is the limiting threshold. While basic PSPs only allow for Boolean states/events in their description, PSPs(\mathcal{D}_C) also allow for atomic numerical constraints. It is straightforward to extend the translation of [DAC99] from basic PSPs to LTL in order to encode every PSP(\mathcal{D}_C) to a formula in LTL(\mathcal{D}_C). Consider, for instance, the set of requirements:

R_1 Globally, it is always the case that $\mathbf{v} \leq \mathbf{5.0}$ holds.

R_2 After \mathbf{a} , $\mathbf{v} \leq \mathbf{8.5}$ eventually holds.

R_3 After \mathbf{a} , it is always the case that if $\mathbf{v} \geq \mathbf{3.2}$ holds, then \mathbf{z} eventually holds.

where \mathbf{a} and \mathbf{z} are Boolean states/events, whereas \mathbf{v} is a numeric signal. These PSPs(\mathcal{D}_C)² can be rewritten as the following LTL(\mathcal{D}_C) formula:

$$\begin{aligned} & \Box(v < 5.0 \vee v = 5.0) && \wedge \\ & \Box(a \rightarrow \Diamond(v < 8.5) \vee (v = 8.5)) && \wedge \\ & \Box(a \rightarrow \Box(\neg(v < 3.2) \rightarrow \Diamond z)) \end{aligned} \quad (4.1)$$

Therefore, to reason about the consistency of sets of requirements written using PSPs(\mathcal{D}_C) it is sufficient to provide an algorithm for deciding the satisfiability of LTL(\mathcal{D}_C) formulas.

To this end, consider an LTL(\mathcal{D}_C) formula ϕ , and let $Var(\phi)$ be the set of variables and $C(\phi)$ be the set of constants that occur in ϕ . We define the *set of thresholds* $S_x(\phi) \subseteq C(\phi)$ as the set of constant values against which some variable $x \in Var(\phi)$ is compared to; more precisely, for every variable $x \in Var(\phi)$ we construct a set $S_x(\phi) = \{c_1, \dots, c_n\}$ such that, for all $c_k \in \mathbb{R}$ with $1 \leq k \leq n$, ϕ contains a constraint of the form $x < c_k$ or $x = c_k$. For convenience, we always consider each threshold set $S_x(\phi)$ ordered in ascending order, i.e., $c_k < c_{k+1}$ for all $1 \leq k < n$. For instance, in example (4.1), we have $Var = \{v\}$ and the corresponding set of threshold is $S_v = \{3.2, 5.0, 8.5\}$. Given an LTL(\mathcal{D}_C) formula ϕ , and some variable $x \in Var(\phi)$, let $S_x(\phi) = \{c_1, \dots, c_n\}$ be the set of thresholds for which we define the corresponding sets of *inequality propositions* $Q_x(\phi) = \{q_1, \dots, q_n\}$ and *equality propositions* $E_x(\phi) = \{e_1, \dots, e_n\}$. Informally, inequality propositions should be true exactly when a variable $x \in Var(\phi)$ is below or between some value in the threshold set $S_x(\phi)$, whereas equality propositions should be true exactly when x is equal to some value in $S_x(\phi)$. Because of this, in our encoding we must ensure that for every computation π and time instant $i \in \mathbb{N}$ exactly one of the following cases is true ($1 \leq j \leq n$):

- $q_j \in \pi(i)$ for some j , $q_l \notin \pi(i)$ for all $l \neq j$ and $e_j \notin \pi(i)$ for all j ;
- $e_j \in \pi(i)$ for some j , $e_l \notin \pi(i)$ for all $l \neq j$ and $q_j \notin \pi(i)$ for all j ;
- $q_j \notin \pi(i)$ and $e_j \notin \pi(i)$ for all j .

²Strictly speaking, the syntax used is not that of \mathcal{D}_C , but a statement like $v \leq 5.0$ can be thought as syntactic sugar for the expression $(v < 5.0) \vee (v = 5.0)$.

The first case above corresponds to a value of x that lies between some threshold value in $S_x(\phi)$ or before its smallest value; the second case occurs when a threshold value is equal to x , and the third case is when x exceeds the highest threshold value in $S_x(\phi)$.

Given the definitions above, an LTL(\mathcal{D}_C) formula ϕ over the set of Boolean propositions AP and the set of terms $T = C \cup Var$, can be converted to an LTL formula ϕ' over the set of Boolean propositions $AP \cup \bigcup_{x \in Var(\phi)} (Q_x(\phi) \cup E_x(\phi))$. We obtain this by considering, for each variable $x \in Var(\phi)$ and associated threshold set $S_x(\phi)$, the corresponding propositions $Q_x(\phi) = \{q_1, \dots, q_n\}$ and $E_x = \{e_1, \dots, e_n\}$; then, for each $c_k \in S_x(\phi)$, we perform the following substitutions:

$$x < c_k \rightsquigarrow \bigvee_{j=1}^k q_j \vee \bigvee_{j=1}^{k-1} e_j \quad \text{and} \quad x = c_k \rightsquigarrow e_k. \quad (4.2)$$

Replacing atomic numerical constraints is not enough to ensure equisatisfiability of ϕ' with respect to ϕ . In particular, for every $x \in Var(\phi)$, we must encode the informal observation made above about “mutually exclusive” Boolean valuations for propositions in $Q_x(\phi)$ and $E_x(\phi)$ as corresponding constraints:

$$\phi_M = \bigwedge_{x \in Var(\phi)} \left(\bigwedge_{a, b \in M_x(\phi), a \neq b} \Box \neg (a \wedge b) \right) \quad (4.3)$$

where $M_x(\phi) = Q_x(\phi) \cup E_x(\phi)$.

For instance, given example (4.1), we have $Q_v = \{q_1, q_2, q_3\}$ and $E_v = \{e_1, e_2, e_3\}$ and the mutual exclusion constraints are written as:

$$\begin{aligned} \phi_M = & \Box \neg (q_1 \wedge q_2) \wedge \Box \neg (q_1 \wedge q_3) \wedge \Box \neg (q_1 \wedge e_1) \wedge \Box \neg (q_1 \wedge e_2) \wedge \\ & \Box \neg (q_1 \wedge e_3) \wedge \Box \neg (q_2 \wedge q_3) \wedge \Box \neg (q_2 \wedge e_1) \wedge \Box \neg (q_2 \wedge e_2) \wedge \\ & \Box \neg (q_2 \wedge e_3) \wedge \Box \neg (q_3 \wedge e_1) \wedge \Box \neg (q_3 \wedge e_2) \wedge \Box \neg (q_3 \wedge e_3) \wedge \\ & \Box \neg (e_1 \wedge e_2) \wedge \Box \neg (e_1 \wedge e_3) \wedge \Box \neg (e_2 \wedge e_3). \end{aligned} \quad (4.4)$$

Therefore, the LTL formula to be tested for assessing the consistency of the requirements is

$$\begin{aligned} \phi_M \wedge (& \Box (q_1 \vee q_2 \vee e_1 \vee e_2) \wedge \\ & \Box (a \rightarrow \Diamond (\bigvee_{i=1}^3 q_i \vee e_i)) \wedge \\ & \Box (a \rightarrow \Box (\neg q_1 \rightarrow \Diamond z))). \end{aligned} \quad (4.5)$$

We can now state the following:

Theorem 1. Let ϕ be an LTL(\mathcal{D}_C) formula on the set of proposition AP and terms $T = Var(\phi) \cup C(\phi)$; for every $x \in Var(\phi)$, let $S_x(\phi)$, $Q_x(\phi)$ and $E_x(\phi)$ be the corresponding set of thresholds, inequality propositions and equality propositions, respectively; let ϕ' be the LTL formula on the set of proposition $AP \cup \bigcup_{x \in Var(\phi)} Q_x(\phi) \cup E_x(\phi)$ obtained from ϕ by applying substitutions (4.2) for every $x \in Var(\phi)$ and $c_k \in S_x(\phi)$, and let ϕ_M be the LTL formula obtained as in (4.3); then, the LTL(\mathcal{D}_C) formula ϕ is satisfiable if and only if the LTL formula $\phi_M \wedge \phi'$ is satisfiable.

Proof. First, we prove that if ϕ is satisfiable the same holds for $\phi_M \wedge \phi'$. Since ϕ is satisfiable, then there exists a computation π and an evaluation ν such that $\pi, \nu \models_{\mathcal{D}_C} \phi$. Let us consider a generic variable $x \in Var(\phi)$, for which the corresponding set of thresholds is $S_x(\phi) = \{c_1, \dots, c_n\}$. Considering that thresholds are ordered in ascending order, we construct the following sets of time instants:

$$\begin{aligned}
N_{x < c_1} &= \{i \in \mathbb{N} \mid \pi, \nu, i \models_{\mathcal{D}_C} x < c_1\} \\
N_{x = c_1} &= \{i \in \mathbb{N} \mid \pi, \nu, i \models_{\mathcal{D}_C} x = c_1\} \\
N_{c_1 < x < c_2} &= \{i \in \mathbb{N} \mid \pi, \nu, i \models_{\mathcal{D}_C} x > c_1 \wedge x < c_2\} \\
&\dots \\
N_{c_{n-1} < x < c_n} &= \{i \in \mathbb{N} \mid \pi, \nu, i \models_{\mathcal{D}_C} x > c_{n-1} \wedge x < c_n\} \\
N_{x = c_n} &= \{i \in \mathbb{N} \mid \pi, \nu, i \models_{\mathcal{D}_C} x = c_n\} \\
N_{x > c_n} &= \{i \in \mathbb{N} \mid \pi, \nu, i \models_{\mathcal{D}_C} x > c_n\}
\end{aligned}$$

which, given the standard semantics of “<” and “=”, are a partition of \mathbb{N} . Let \mathcal{N}_x denote such partition for a specific variable $x \in Var(\phi)$. We construct a computation π' such that, for all time instants $i \in \mathbb{N}$ and propositions $p \in AP$, we have $p \in \pi'(i)$ exactly when $p \in \pi(i)$ and, for each variable $x \in Var(\phi)$, given $Q_x(\phi) = \{q_1, \dots, q_n\}$ and $E_x(\phi) = \{e_1, \dots, e_n\}$, we have also

- $q_1 \in \pi'(i)$ exactly when $i \in N_{x < c_1}$;
- $e_1 \in \pi'(i)$ exactly when $i \in N_{x = c_1}$;
- $q_2 \in \pi'(i)$ exactly when $i \in N_{c_1 < x < c_2}$;
- ...
- $q_n \in \pi'(i)$ exactly when $i \in N_{c_{n-1} < x < c_n}$;

- $e_n \in \pi'(i)$ exactly when $i \in N_{x=c_n}$.

Notice that for all $i \in N_{x>c_n}$, we have that $\pi'(i) \cap M_x(\phi) = \emptyset$, where $M_x(\phi) = Q_x(\phi) \cup E_x(\phi)$. Since \mathcal{N}_x is a partition of \mathbb{N} for each variable $x \in Var(\phi)$, it follows that $\pi' \models \phi_M$ because for all $a, b \in M_x(\phi)$, there is no time instant $i \in \mathbb{N}$ such that $\pi', i \models a \wedge b$. Now we show that for every $i \in \mathbb{N}$, $\pi', i \models \phi'$ if and only if $\pi, \nu, i \models_{\mathcal{D}_C} \phi$ by induction on the set of subformulas of ϕ . Let ψ and ψ' be two subformulas of ϕ and ϕ' , respectively. For every $i \in \mathbb{N}$:

- if $\psi \equiv p$ for $p \in AP$ then $\psi' \equiv p$; therefore, for any given $i \in \mathbb{N}$, we have $\pi, \nu, i \models_{\mathcal{D}_C} p$ if and only if $\pi', i \models p$ by construction of π' .
- if $\psi \equiv (x < c_k)$ for some variable $x \in Var(\phi)$ and some constant $c_k \in S_x(\phi)$ then, according to (4.2),

$$\psi' \equiv \bigvee_{j=1}^k q_j \vee \bigvee_{j=1}^{k-1} e_j.$$

Let $N_{x,k}$ be the set defined as

$$N_{x,k} = N_{x<c_1} \cup N_{x=c_1} \cup \dots \cup N_{c_{k-1}<x<c_k}$$

There are two cases: either $i \in N_{x,k}$ or $i \notin N_{x,k}$. In the former case, we have that $\pi, \nu, i \models_{\mathcal{D}_C} (x < c_k)$ and, by construction of π' , this happens exactly when $\pi', i \models q_j$ for some $1 \leq j \leq k$ or $\pi', i \models e_j$ for some $1 \leq j < k$ which, by the semantics of disjunction and construction of π' , is also exactly when $\pi', i \models \psi'$. In the second case, $\pi, \nu, i \not\models_{\mathcal{D}_C} (x < c_k)$ and, by construction of π' , this happens exactly when $\pi', i \not\models q_j$ for all $1 \leq j \leq k$ and $\pi', i \not\models e_j$ for all $1 \leq j < k$ which, by the semantics of disjunction, is also exactly when $\pi', i \not\models \psi'$.

- if $\psi \equiv x = c_k$ for some variable $x \in Var(\phi)$ and some constant $c_k \in S_x(\phi)$ then, according to (4.2), $\psi' \equiv e_k$. The time instants $i \in \mathbb{N}$ in which $\pi, \nu, i \models_{\mathcal{D}_C} x = c_k$ are contained in the set $N_{x=c_k}$, so there are two cases: either $i \in N_{x=c_k}$ or $i \notin N_{x=c_k}$. In the former case, we have that $\pi, \nu, i \models_{\mathcal{D}_C} (x = c_k)$ and, by construction of π' , this happens exactly when $\pi', i \models e_k$. In the second case, $\pi, \nu, i \not\models_{\mathcal{D}_C} (x = c_k)$ and, by construction of π' , this happens exactly when $\pi', i \not\models e_k$.

- if $\psi = \neg\alpha$ then $\psi' = \neg\alpha'$; by induction, we can assume that for every i , we have $\pi, \nu, i \models_{\mathcal{D}_C} \alpha$ if and only if $\pi', i \models \alpha'$, and thus $\pi, i \not\models_{\mathcal{D}_C} \alpha$ if and only if $\pi', i \not\models \alpha'$. By the semantics of negation, we have that for any given $i \in \mathbb{N}$, $\pi, i, \nu \models_{\mathcal{D}_C} \neg\alpha$ if and only if $\pi, i, \nu \not\models_{\mathcal{D}_C} \alpha$ and this happens exactly when $\pi', i \not\models \alpha'$, i.e., $\pi', i \models \neg\alpha'$;
- if $\psi \equiv (\alpha \vee \beta)$ then $\psi' \equiv \alpha' \vee \beta'$; by induction, we can assume that for all $i \in \mathbb{N}$ we have that $\pi, \nu, i \models_{\mathcal{D}_C} \alpha$ and $\pi, \nu, i \models_{\mathcal{D}_C} \beta$ if and only if $\pi', i \models \alpha'$ and $\pi', i \models \beta'$, respectively. By the semantics of disjunction, we have that, for any given $i \in \mathbb{N}$, $\pi, i, \nu \models_{\mathcal{D}_C} \alpha \vee \beta$ exactly when $\pi, \nu, i \models \alpha$ or $\pi, \nu, i \models \beta$ and this happens exactly when $\pi', i \models \alpha'$ or $\pi', i \models \beta'$, i.e., by the semantics of disjunction, $\pi', i \models \alpha' \vee \beta'$.
- if $\psi \equiv \mathcal{X}\alpha$ then $\psi' \equiv \mathcal{X}\alpha'$; by induction, we can assume that for all $j \in \mathbb{N}$ we have $\pi, \nu, j \models_{\mathcal{D}_C} \alpha$ if and only if $\pi', j \models \alpha'$. By the semantics of the “next” operator we have that, for any given $i \in \mathbb{N}$, $\pi, i, \nu \models_{\mathcal{D}_C} \mathcal{X}\alpha$ if and only if $\pi, \nu, i+1 \models_{\mathcal{D}_C} \alpha$ which happens exactly when $\pi', i+1 \models \alpha'$, i.e., $\pi', i \models \mathcal{X}\alpha$.
- if $\psi \equiv \alpha \mathcal{U} \beta$ then $\psi' = \alpha' \mathcal{U} \beta'$; by induction, we can assume that, for all $j \in \mathbb{N}$, we have $\pi, \nu, j \models_{\mathcal{D}_C} \beta$ if and only if $\pi', j \models \beta'$ and that $\pi, \nu, j \models_{\mathcal{D}_C} \alpha$ if and only if $\pi', j \models \alpha'$. By the semantics of the “until” operator we have that, for any given i , $\pi, i, \nu \models_{\mathcal{D}_C} \alpha \mathcal{U} \beta$ if and only if for some $j \geq i$ we have $\pi, \nu, j \models_{\mathcal{D}_C} \beta$ and for all k such that $i \leq k < j$ we have $\pi, \nu, k \models_{\mathcal{D}_C} \alpha$. However, the former happens exactly when for the same $j \in \mathbb{N}$ we have $\pi', j \models \beta'$ and for all k such that $i \leq k < j$ we have $\pi', k \models_{\mathcal{D}_C} \alpha'$, i.e., $\pi', i \models \alpha' \mathcal{U} \beta'$.

We now prove that the satisfiability of $\phi_M \wedge \phi'$ in LTL implies the satisfiability of ϕ in $\text{LTL}(\mathcal{D}_C)$. First we observe that, for a generic variable $x \in \text{Var}(\phi)$, and for all time instants $i \in \mathbb{N}$, every computation π' such that $\pi' \models \phi_M$ has at most one proposition $p \in M_x(\phi)$ for which $p \in \pi(i)$. Therefore, for all variables $x \in \text{Var}(\phi)$ and for every time instant $i \in \mathbb{N}$, we have the following cases only (where $n = |S_x(\phi)| = |E_x(\phi)| = |Q_x(\phi)|$):

1. $\pi', i \models e_k$ for some $e_k \in E_x(\phi)$; consequently, as long as $k < n$, also $\pi', i \models \bigvee_{j=1}^{k+1} q_j \vee \bigvee_{j=1}^k e_j$ holds.

2. $\pi', i \models q_k$ for some $q_k \in Q_x(\phi)$, and thus $\pi', i \models \bigvee_{j=1}^k q_j \vee \bigvee_{j=1}^{k-1} e_j$ holds.
3. $\pi', i \not\models p$ for every $p \in M_x(\phi)$; consequently, for all k it is also the case that $\pi', i \not\models \bigvee_{j=1}^k q_j \vee \bigvee_{j=1}^{k-1} e_j$ and $\pi', i \not\models e_k$.

A computation π and an evaluation ν such that $\pi, \nu \models_{\mathcal{D}_C} \phi$ can be constructed as follows. For every $p \in AP$, and time instant $i \in \mathbb{N}$, let $p \in \pi(i)$ exactly when $p \in \pi'(i)$. As for the evaluation ν , for a generic variable $x \in Var(\phi)$, and for every time instant $i \in \mathbb{N}$, we can construct ν considering that π' is bound to satisfy the three cases above :

1. $\nu(x, i) = c_k$ for the same k s.t. $\pi', i \models e_k$; consequently, as long as $k < n$, both $\pi, \nu, i \models x < c_{k+1}$ and $\pi, \nu, i \models x = c_k$ hold.
2. $\nu(x, i) = v$ and, for the same k s.t. $\pi', i \models q_k$, if $k > 1$, then $c_{k-1} < v < c_k$, else if $k = 1$, then $v < c_1$; consequently $\pi, \nu, i \models x < c_k$ holds and, in case $k > 1$, $\pi, \nu, i \not\models x < c_j$ for all $j < k$.
3. $\nu(x, i) = v$ with $v > c_n$; consequently $\pi, \nu, i \not\models x < c_k$ and $\pi, \nu, i \not\models x = c_k$ for all k

An induction proof analogous to the one provided for the “if” part can be provided to show that if $\pi' \models \phi'$, then also $\pi, \nu \models \phi$, with π and ν constructed as shown above. \square

The proposed translation from $LTL(\mathcal{D}_C)$ to a LTL formula is also quite compact, i.e., the number of symbols in the LTL encoding grows at most quadratically with the number of symbols in the original formula. Let us define the size of a formula ϕ , denoted as $|\phi|$, in the usual way, i.e., by counting the number of symbols in it. We can state the following:

Theorem 2. Let ϕ be an $LTL(\mathcal{D}_C)$ formula on the set of propositions AP and terms $T = Var(\phi) \cup C(\phi)$; for every $x \in Var(\phi)$, let $S_x(\phi)$, $Q_x(\phi)$ and $E_x(\phi)$ be the corresponding set of thresholds, inequality propositions and equality propositions, respectively; let ϕ' be the LTL formula on the set of proposition $AP \cup \bigcup_{x \in Var(\phi)} Q_x(\phi) \cup E_x(\phi)$ obtained from ϕ by applying substitutions (4.2) for every $x \in Var(\phi)$ and $c_k \in S_x(\phi)$, and ϕ_M be the LTL formula obtained as in (4.3); the size of $\phi' \wedge \phi_M$ is at most quadratic in the size of ϕ , i.e., $O(|\phi' \wedge \phi_M|) = O(|\phi|^2)$.

Proof. From Equation (4.3), for each variable $x \in Var(\phi)$, all combinations of two elements from the set $M_x(\phi) = Q_x(\phi) \cup E_x(\phi)$ are required to build ϕ_M . Therefore, if $n = |S_x(\phi)|$, the number of conjuncts of the form $\Box\neg(a \wedge b)$ in ϕ_M is

$$\binom{2n}{2} = \frac{2n!}{2!(2n-2)!} = \frac{2n(2n-1)}{2} = n(2n-1) \quad (4.6)$$

If we consider $m = \max_{x \in Var(\phi)} |S_x(\phi)|$ and the number of conjuncts derived in equation (4.6), it follows that

$$|\phi_M| = O(|Var(\phi)| \cdot m(2m-1)) = O(|Var(\phi)| \cdot m^2). \quad (4.7)$$

Now it remains to show the effect of substitution (4.2) in ϕ . For every variable $x \in Var(\phi)$ and for each constant $c_k \in S_x(\phi)$ in ϕ we have:

- one proposition in ϕ' for each occurrence of the term $x = c_k$ in ϕ ;
- a formula of size $2k-1$ in ϕ' for each occurrence of the term $x < c_k$ in ϕ .

Let $m = \max_{x \in Var(\phi)} |S_x(\phi)|$, and p be the maximum number of occurrences in ϕ of any condition $x = c$ or $x < c$ for specific values of $x \in Var(\phi)$ and $c \in C(\phi)$. Then we can write

$$|\phi| = O(|Var(\phi)| \cdot p \cdot m + r) \quad (4.8)$$

where r is the number of symbols that are not terms. Since each term in ϕ is translated to a formula of size $O(m)$ in ϕ' , we have that

$$|\phi'| = O(|Var(\phi)| \cdot p \cdot m^2 + r) \quad (4.9)$$

Considering (4.7) together with (4.9) we obtain

$$\begin{aligned} O(|\phi' + \phi_M|) &= O(|Var(\phi)| \cdot m^2) + O(|Var(\phi)| \cdot p \cdot m^2 + r) \\ &= O(|Var(\phi)| \cdot m^2 \cdot (1 + p) + r) \end{aligned} \quad (4.10)$$

Given (4.8) and the fact that the values of the parameters $|Var(\phi)|$, p and r do not depend on the translation, from (4.10) we conclude that $O(|\phi' + \phi_M|) = O(|\phi|^2)$. \square

4.2 Inconsistency Explanation

Given a set $R = \{r_1, \dots, r_n\}$ of inconsistent requirements written as $PSP(\mathcal{D}_C)$, the aim of the algorithms proposed in this Section is to compute a *Minimal Unsatisfiable Core* (*MUC*), i.e., a subset $I \subseteq R$ such that removing any element r_i

r_i	PSP
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.

Table 4.1: Set R of inconsistent PSPs.

from I makes the set consistent again. Table 4.1 shows an inconsistent specification as a set $R = \{r_1, \dots, r_7\}$ of seven requirements. Looking at the table, we can see that there are 4 different MUCs in R , namely $\{r_1, r_2\}$, $\{r_2, r_6\}$, $\{r_3, r_4, r_5\}$, $\{r_4, r_5, r_6\}$. In the remainder of the section we present two algorithms devoted to the extraction of MUC for PSPs.

4.2.1 Linear Deletion-Based MUC Extraction

The first algorithm we present is based on a deletion-based strategy, and its pseudo-code is depicted in Algorithm 1. The procedure works as follows. If the set $R' \leftarrow R \setminus \{r\}$ with $r \in R$ is inconsistent, then r is not in the MUC. On the other hand, if R' is consistent, then r is part of a MUC and cannot be removed. Such operation is repeated iteratively and the algorithm terminates when all requirements have been checked for inclusion in the MUC.

It is easy to see that, with $|R| = n$, the loop iterates n times, and that at each iteration the `ISCONSISTENT` function is called once. The input of the function is R' and its size is given by $|R'|$. The number of elements in R' is reduced by one at each iteration, but r_i could be added back again in R' , depending on the result of `ISCONSISTENT`. The worst case is obtained when all requirements are part of the MUC, i.e., each requirement r_i is first removed and then reinserted again. In this case the model checker is called each time with $n - 1$ requirements. The overall complexity is therefore $O(n \cdot C(n))$, where n is the number of elements initially in R and $C(n)$ is the complexity for the consistency check of n requirements. The algorithm is therefore linear in the

Algorithm 1: Linear Deletion-Based MUC Extraction 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

```

number of calls to the model checker.

Example 1. Considering the set R in Table 4.1, Algorithm 1 works along the following steps.

Step	r_i	R'	ISCONSISTENT(R')
1:	r_1	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$	FALSE
2:	r_2	$\{r_3, r_4, r_5, r_6, r_7\}$	FALSE
3:	r_3	$\{r_4, r_5, r_6, r_7\}$	FALSE
4:	r_4	$\{r_5, r_6, r_7\}$	TRUE
5:	r_5	$\{r_4, r_6, r_7\}$	TRUE
6:	r_6	$\{r_4, r_5, r_7\}$	TRUE
7:	r_7	$\{r_4, r_5, r_6\}$	FALSE

The final result is $R' = \{r_4, r_5, r_6\}$. It is worth to notice that this result depends on the extraction order of the requirements. It is easy to see that processing the requirements in reverse order would yield $R' = \{r_1, r_2\}$ as a result instead.

4.2.2 Dichotomic MUC Extraction

Algorithm 2 is based on the same general-purpose structure of algorithm 1, but it also exploits the fact that the dimension of the MUC is often much smaller than $|R|$. Therefore, it is possible to exploit a “divide and conquer” strategy to reduce the search space. Considering Algorithm 2, R is split in two halves R_1

and R_2 , such that $R_1 \cup R_2 = R$ and $R_1 \cap R_2 = \emptyset$. If one of the two halves (plus I) is inconsistent, then there is no need to explore the other one and we can proceed recursively. Otherwise it means that the MUC has been split in the two halves and further search is needed. This is done by means of two recursive calls (lines 21–22); The former performs the search on R_2 considering the whole set R_1 as inconsistent, while the latter continues the search on R_1 , removing from I the requirements that still need to be checked. The algorithm terminates when R has 1 or 0 elements.

As for the complexity of the algorithm the best case occurs when the MUC is always in the first half of R . In such a case, half of the requirements are discarded at each iteration, and it is easy to see that complexity is $\Omega(\log |R|)$. The worst case occurs when the set of inconsistent requirements I coincides with R . For example, let R be comprised of $\{r_1, r_2, r_3, r_4\}$ and let MUC be R itself. At the first step, the algorithm checks $R'_1 = \{r_1, r_2\}$ and $R'_2 = \{r_3, r_4\}$ but both sets are consistent. Therefore FINDINCONSISTENCY is called recursively with $R = \{r_3, r_4\}$ and $I = \{r_1, r_2\}$. At this point we have $R'_1 = \{r_3\}$ and $R'_2 = \{r_4\}$. The algorithm checks the consistency of $\{r_1, r_2, r_3\}$ and $\{r_1, r_2, r_4\}$ and returns to the previous recursive call. This time FINDINCONSISTENCY is called again, but with $R = \{r_1, r_2\}$ and $I = \{r_3, r_4\}$ and the same process is applied. In general, if $|R| = n$ and $C(n)$ is the complexity for the consistency check of n requirements, then the worst case complexity of this algorithm is $O(n \cdot C(n))$ – the same as the previous one. However, as we will show in Section 4.3.2, when $|I| \ll |R|$ it is noticeable faster than the linear version.

Example 2. Considering again the set R reported in Table 4.1, in the following we report step-by-step how Algorithm 2 works. For lack of space in the table we replace ISCONSISTENT(R) with $C(R)$.

Step	R	R_1	R_2	I	$C(R_1 \cup I)$	$C(R_2 \cup I)$
1:	$\{r_1, \dots, r_7\}$	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6, r_7\}$	$\{\}$	<i>False</i>	–
2:	$\{r_1, r_2, r_3\}$	$\{r_1\}$	$\{r_2, r_3\}$	$\{\}$	<i>True</i>	<i>True</i>
3:	$\{r_2, r_3\}$	$\{r_2\}$	$\{r_3\}$	$\{r_1\}$	–	–
4:	$\{r_2\}$	–	–	$\{r_1\}$	–	–
5:	$\{r_1\}$	–	–	$\{r_2\}$	–	–

Algorithm 2: Dichotomic MUC Extraction Algorithm

```

1: function FINDINCONSISTENCY( $R$ )
2:   return FINDINCONSISTENCY( $R, \emptyset$ )
3: end function

4: function FINDINCONSISTENCY( $R, I$ )
5:   if  $|R| \leq 1$  then
6:     if ISCONSISTENT( $I$ ) then
7:       return  $I \cup R$ 
8:     else
9:       return  $I$ 
10:    end if
11:  end if
12:   $(R_1, R_2) \leftarrow \text{SPLIT}(R)$ 
13:  if  $|R_1| > 1$  and  $|R_2| > 1$  then
14:    if  $\neg$  ISCONSISTENT( $R_1 \cup I$ ) then
15:      return FINDINCONSISTENCY( $R_1, I$ )
16:    end if
17:    if  $\neg$  ISCONSISTENT( $R_2 \cup I$ ) then
18:      return FINDINCONSISTENCY( $R_2, I$ )
19:    end if
20:  end if
21:   $I \leftarrow \text{FINDINCONSISTENCY}(R_2, I \cup R_1)$ 
22:   $I \leftarrow \text{FINDINCONSISTENCY}(R_1, I \setminus R_1)$ 
23:  return  $I$ 
24: end function

```

In the first step, the algorithm splits the initial set R in two subset R_1 and R_2 , and checks the consistency of the first one. Since R_1 is inconsistent, the algorithm automatically discards R_2 and continue with step 2. Also in this case the new set $R = \{r_1, r_2, r_3\}$ is split in two, but this time both are consistent and so the two recursive calls in line 21–22 are executed: the first one is resolved in step 3 and 4, while the second one in step 5. In the last two steps, the basic

case is reached (lines 5–11), and since the call to `ISCONSISTENT(I)` returns true in both cases, r_1 and r_2 are added to I . Therefore, $I = \{r_1, r_2\}$ is returned as final answer. In this case `ISCONSISTENT` is called 6 times instead of 7 as in the previous example, and with smaller instances.

4.3 Analysis with Probabilistic Requirement Generation

The aim of this Section is twofold; On the one hand, we evaluate the scalability of our approach for consistency checking, experimenting the encoding proposed in Section 4.1 with a pool of state-of-the-art LTL model checkers. On the other hand, we assess the performance of the MUC extraction algorithms described in Section 4.2, in order to evaluate the possibility of their usage in contexts of practical interest.

Since we want to have control over different dimensions of the specifications – namely, the kind of requirements, the number of constraints, and the size of the corresponding domains – we generate artificial specifications using a probabilistic model that we devised and implemented specifically to carry out the experiments herein presented.

In particular, the following parameters can be tuned in our generator of specifications:

- The number of requirements generated ($\#req$).
- The probability of each different body to occur in a pattern.
- The probability of each different scope to occur in a pattern.
- The size ($\#vars$) of the set from which variables are picked uniformly at random to build patterns.
- The size (dom) of the domain from which the thresholds of the atomic constraints are chosen uniformly at random.

4.3.1 Evaluation of LTL(\mathcal{D}_c) Satisfiability

The goal of this experiment is to evaluate the performance – in terms of correctness, efficiency, and scalability – of LTL model checkers for the consistency checking task described in Section 4.1. To this end, we evaluate the performances of state-of-the-art tools for LTL satisfiability, and then we consider the best among such tools to assess whether our approach can scale to sets of requirements of realistic size. All the experiments here reported ran on a workstation equipped with 2 Intel Xeon E5-2640 v4 CPUs and 256GB RAM running Debian with kernel 3.16.0-4.

4.3.1.1 Evaluation of LTL satisfiability solvers.

The tools considered in our analysis are the ones included in the portfolio solver POLSAT [LPZ⁺13], namely AALTA [LZPV15], NUSMV [CCG⁺02], PLTL [Sch98], and TRP++ [HK03]. We also consider LEVIATHAN [BGMR16], a tableaux-based system for consistency checking that has been recently published. Notice that in the case of NUSMV, we consider two different encodings. With reference to Property 1, the first encoding defines ϕ_M as an invariant — denoted as NUSMV-INVAR — and ϕ' is the property to check; the second encoding considers $\phi_M \wedge \phi$ as the property to check — denoted as NUSMV-NOINVAR. Finally, concerning AALTA, we slightly modified its default version in order to be able to evaluate large formulas. In particular, we modified the source code increasing of two orders of magnitude the input size buffer.

In our experimental analysis we set the range of the parameters as follows: $\#vars \in \{16, 32\}$, $dom \in \{2, 4, 8, 16\}$, and $\#req \in \{8, 16, 32, 64\}$. For each combination of the parameters with $v \in \#vars$, $r \in \#req$ and $d \in dom$, we generate 10 different benchmarks. Each benchmark is a specification containing r requirements where each scope has (uniform) probability 0.2 and each body has (uniform) probability 0.1. Then, for each atomic numerical constraint in the benchmark, we choose a variable out of v possible ones, and a threshold value out of d possible ones. In Table 4.2 we show the results of the analysis. Notice that we do not show the results of TRP++ because of the high number of failures obtained. Looking at the table, we can see that AALTA is the tool with the best performances, as it is capable of solving two times the problems solved

<i>dom</i>	2				4				8				16			
<i>#vars</i>	16		32		16		32		16		32		16		32	
Tool	S	T	S	T	S	T	S	T	S	T	S	T	S	T	S	T
AALTA	16	0.0	27	0.1	22	0.1	29	0.4	26	0.6	29	1.4	25	2.8	31	4.9
LEVIATHAN	4	0.1	6	0.3	7	0.8	5	0.2	0	–	7	2.3	4	47.7	7	12.8
NUMMV-INVAR	11	30.4	10	185.1	10	804.2	9	881.3	11	68.1	8	402.9	10	1172.6	8	1001.9
NUMMV-NOINVAR	11	65.0	10	489.7	7	303.6	7	505.5	11	92.4	10	1277.6	8	660.0	9	1394.5
PLTL	8	25.0	11	108.1	9	1.2	10	0.6	10	19.6	11	0.1	11	14.5	14	3.5

Table 4.2: Evaluation of LTL satisfiability solvers on randomly generated requirements. The first line reports the size of the domain (*dom*), while the second line reports the total amount of variables (*vars*) for each domain size. Then, for each tool (on the first column), the table shows the total amount of solved problems and the CPU time (in seconds) spent to solve them (columns “S” and “T”, respectively).

by other solvers in most cases. Moreover, AALTA is up to 3 orders of magnitude faster than its competitors. Considering unsolved instances, it is worth noticing that in our experiments AALTA never reaches the granted time limit (10 CPU minutes), but it always fails beforehand. This is probably due to the fact that AALTA is still in a relatively early stage of development and it is not as mature as NUMMV and PLTL. Most importantly, we did not find any discrepancies in the satisfiability results of the evaluated tools, with the noticeable exception of TRP++, for which we did not report performance in Table 4.2.

4.3.1.2 Evaluation of scalability.

The analysis involves 2560 different benchmarks generated as in the previous experiment. The initial value of *#req* has been set to 15, and it has been doubled until 1920, thus obtaining benchmarks with a total amount of requirements equals to 15, 30, 60, 120, 240, 480, 960, and 1920. Similarly has been done for *#vars* and *#dom*; the former ranges from 5 to 640, while the latter ranges from 4 to 32. At the end of the generation, we obtained 10 different sets composed of 256 benchmarks. In Figure 4.1 and Figure 4.2 we present the results, obtained running AALTA. The Figure is composed by 8 plots, one for each value of *#vars*. Looking at the plots in Figures 4.1 and 4.2, we can see that the difficulty of the problem increases when all the values of the considered parameters increase,

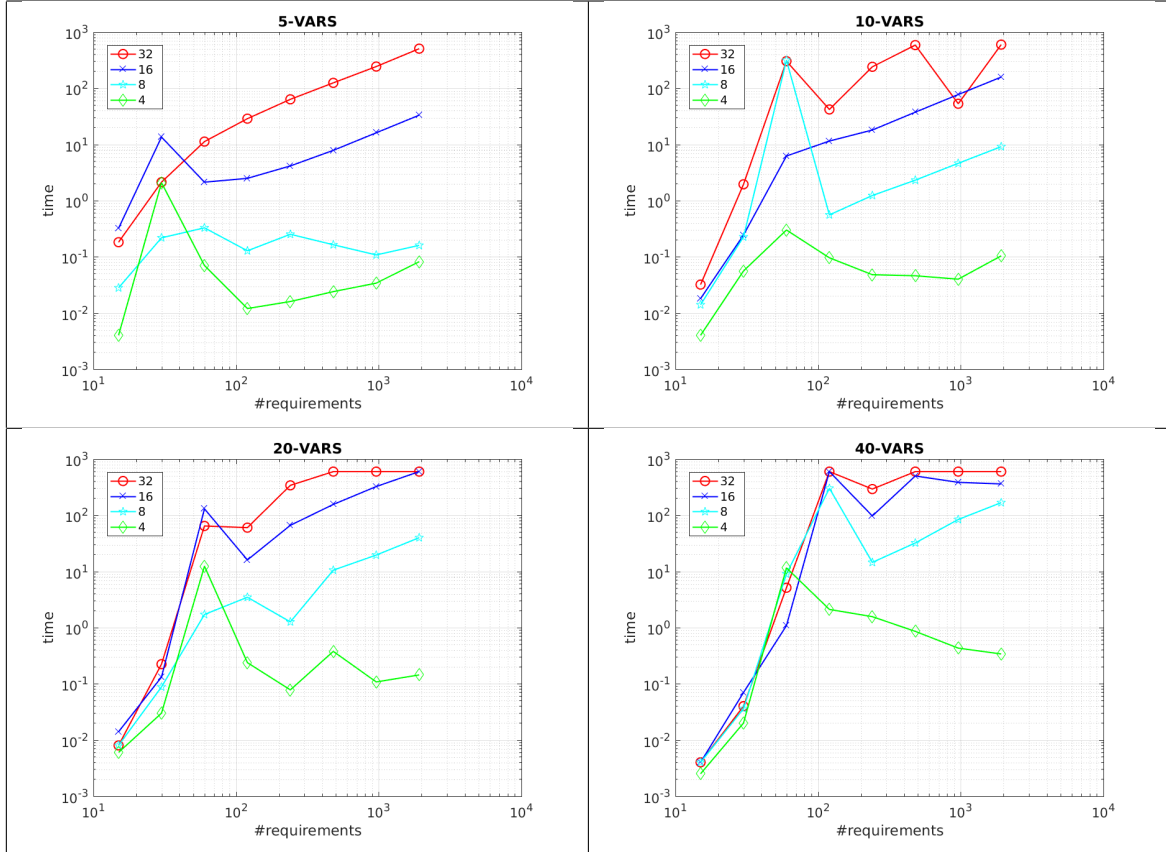


Figure 4.1: Scalability Analysis (Part 1). On the x -axes (y -axes resp.) we report $\#req$ (CPU time in seconds resp.). Axis are both in logarithmic scale. In each plot we consider different values of $\#dom$. In particular, the diamond green line is for $\#dom = 4$, the light blue line with stars is for $\#dom = 8$, the blue crossed lines and red circled ones denote $\#dom = 16$ and $\#dom = 32$, respectively.

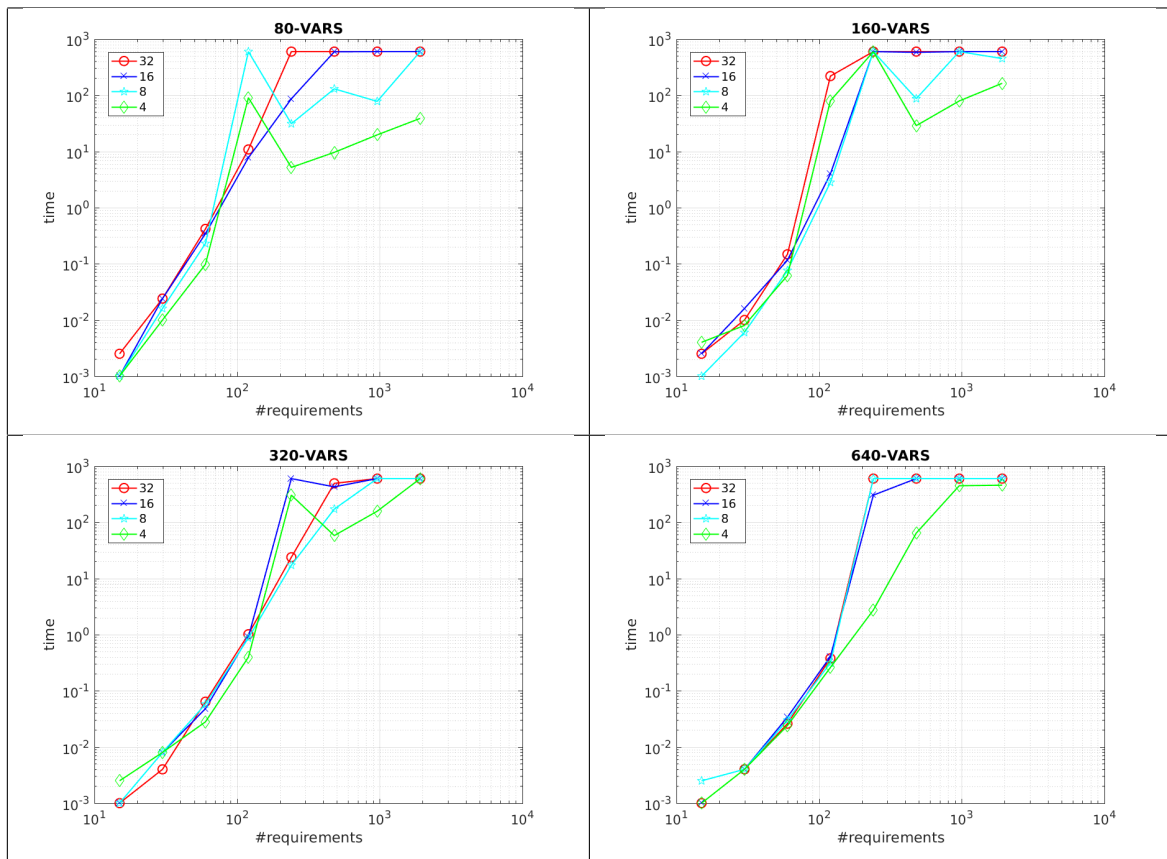


Figure 4.2: Scalability Analysis (Part 2). Plots are organized as in Figure 4.1.

#req	N
8	16
16	38
32	65
60	83
120	147
240	210

Table 4.3: Synopsis of the pool of benchmarks involved in the analysis of MUC extraction algorithms. The table is organized in two columns, namely the total amount of requirements for each benchmark (column “#req”) and the total amount of benchmarks falling in the related category (column “N”).

and this is particularly true considering the total amount of requirements. The parameter $\#dom$ has a higher impact of difficulty when the number of variables is small. Indeed, when the number of variables is less than 40 there is a clear difference between solving time with $\#dom = 4$ and $\#dom = 32$. On the other hand when the number of variables increases, all the plots for various values of $\#dom$ are very close to each other. As a final remark, we can see that even considering the largest problem ($\#vars = 640$, $\#dom = 32$), more than the 60% of the problems are solved by AALTA within the time limit of 10 minutes.

4.3.2 Evaluation of MUC Extraction

In order to evaluate the algorithms proposed in Section 4.2, we consider the pool of inconsistent benchmarks resulting from the experiment presented in Section 4.3.1, for a total amount of 559, having different requirements set dimension as reported in Table 4.3. All the experiments reported in this section ran on a workstation equipped with an Intel Xeon E31245 @ 3.30GHz CPU and 16GB RAM running Ubuntu 14.04 LTS.

In Figure 4.3 we report the results obtained from the experiment described above. For each plot, we report the median CPU time (in seconds) over 10 runs of the same benchmark, granting for each run 600 CPU seconds. AALTA has been used for the satisfiability check.

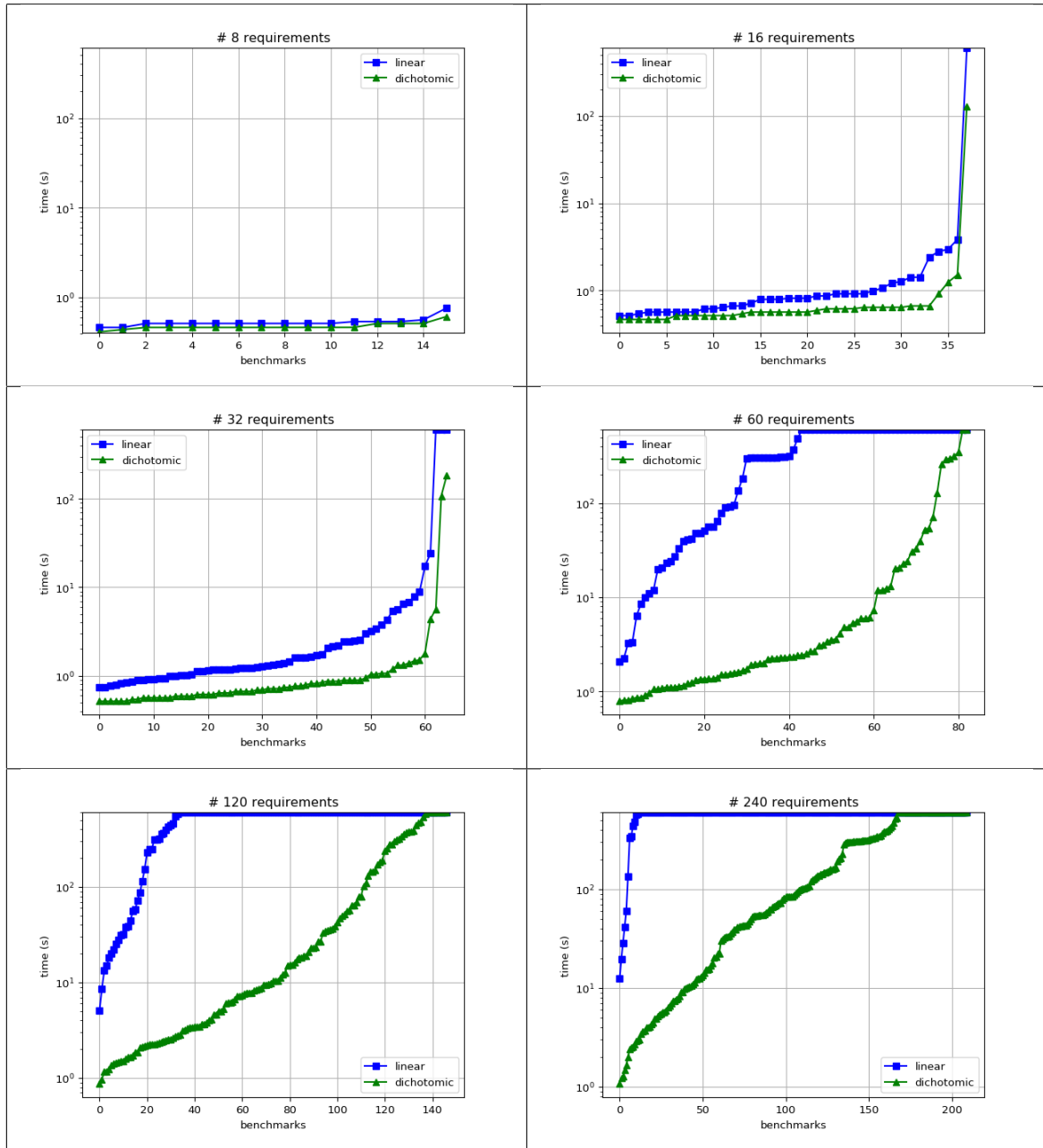


Figure 4.3: Performance of the algorithms for MUC extraction. On the x-axes we report the number of benchmarks, and on the y-axes we report the time in logarithmic scale. In each plot we consider different values of $\#req$. The green and blue lines shows median times of the dichotomic and linear algorithms, respectively.

Looking at the plots, we can see that the dichotomic algorithm is, as expected, overall faster than the linear one. Despite the fact that they show similar performance for benchmarks having 8 and 16 requirements (top-most plots in Figure 4.3), looking at the plots in the middle of Figure 4.3 we can see that the dichotomic algorithm is at least one order of magnitude faster than the linear one for benchmarks having 32 and 60 requirements. Moreover, we report that the latter was able to return MUCs only for 62 out of 65 and 43 out of 83, while the former returned a solution for all instances with 32 requirements and 81 out of 83 for instances with 60 requirements.

Considering the plots in the bottom of Figure 4.3, we can see that the gap between the two algorithms increases even further: the linear one was able to return MUCs only for 34 and 12 benchmarks of 120 and 240 requirements respectively, while the dichotomic one returned a MUC for 138 out of 147 and 168 out of 210 benchmarks. In addition, it is worth noticing that the MUCs found are usually small in size; indeed, in all 6 configurations, the median size of the MUCs found by the two algorithms is 2.

Finally, we report that we involved in our analysis also benchmarks composed of 480 requirements, but our algorithms were not able to return a solution within the considered CPU time limit.

As a final remark, notice that we limit the presentation of the results to the algorithms presented in Section 4.2 because state-of-the-art tools able to cope with this task, namely PLTL-MUP [GHST13] and TRP++UC [Sch16b], report the same correctness and scalability issues of their counterparts presented in Section 4.3.1. For instance, considering the benchmark with 60 requirements – the first one for which we can see a noticeable difference between the performance of the linear and the dichotomic algorithm – we report that PLTL-MUP was not able to solve any instance, while TRP++UC tops its performance at 37% of our worst algorithm (the linear one solved 43 instances out of 83). We also involved in our preliminary analysis also PROC-MINE [AGTW11], but we do not report its results for similar motivations.

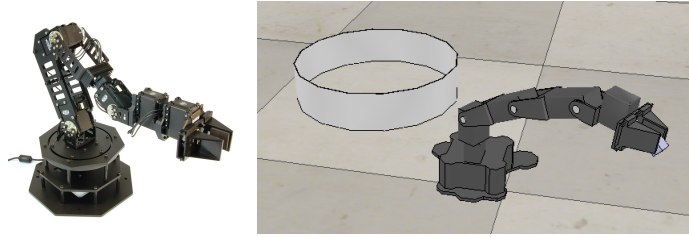


Figure 4.4: WidowX robotic arm (left) and the simulated arm moving a grabbed object in the bucket on the left (right).

4.4 Analysis with a Controller for a Robotic Manipulator

In this Section, as a basis for our experimental analysis, we consider a set of requirements from the design of an embedded controller for a robotic manipulator. The controller should direct a properly initialized robotic arm — and related vision system — to look for an object placed in a given position and move to such position in order to grab the object; once grabbed, the object is to be moved into a bucket placed in a given position and released without touching the bucket. The robot must stop also in the case of an unintended collision with other objects or with the robot itself — collisions can be detected using torque estimation from current sensors placed in the joints. Finally, if a general alarm is detected, e.g., by the interaction with a human supervisor, the robot must stop as soon as possible. The manipulator is a 4 degrees-of-freedom Trossen Robotics WidowX arm³ equipped with a gripper: Figure 4.4 shows a snapshot of the robot in the intended usage scenario taken from CoppeliaSim⁴ simulator. The design of the embedded controller was part of the activities related to the “Self-Healing System for Planetary Exploration” use case [MPM⁺17] in the context of the EU project CERBERO.

In this case study, constrained numerical signals are used to represent requirements related to various parameters, namely angle, speed, acceleration, and torque of the 4 joints, size of the object picked, and force exerted by the

³<http://www.trossenrobotics.com/widowxrobotarm>.

⁴<http://www.coppeliarobotics.com/>

Pattern	Specification			Fault injections		
	AFTER	AFTER_UNTIL	GLOBALLY	AFTER	AFTER_UNTIL	GLOBALLY
Absence	–	12	14	[F4]	–	[F3]
Existence	9	–	–	–	[F5]	[F4, F6]
Invariant	–	–	29	–	–	[F2, F6]
Precedence	–	–	1	–	–	–
ResponseChain	–	–	2	–	–	–
Response	1	–	4	–	–	[F1]
Universality	2	–	1	–	–	–

Table 4.4: Robotic use case requirements synopsis. The table is organized as follows: the first column reports the name of the patterns and it is followed by two groups of three columns denoted with the scope type: the first group refers to the intended specification, the second to the one with fault injections. Each cell in the first group reports the number of requirements grouped by pattern and by scope type. Cells in the second group categorize the 6 injected faults, labeled with F1, . . . , F6.

end-effector. We consider 75 requirements, including those involving scenario-independent constraints like joints limits, and mutual exclusion among states, as well as specific requirements related to the conditions to be met at each state. The set of requirements involved in our analysis includes 14 Boolean signals and 20 numerical ones. In Table 4.4 we present a synopsis of the requirements, to give an idea of the kind of patterns used in the specification.⁵ While most requirements are expressed with the Invariant pattern, e.g., mutual exclusiveness of states and safety conditions, the expressivity of LTL is required to describe the evolution of the system. Indeed, as shown in [DAC99] and [PH12], it is often the case that few PSPs cover the majority of specifications whereas others are sparsely used.

Our first experiment⁶ is to run NUSMV-INVAR on the intended specification translated to $LTL(\mathcal{D}_C)$. The motivation for presenting the results with

⁵The full list of requirements and the fault injection examples are available at <https://github.com/SAGE-Lab/robot-arm-usecase>.

⁶Experiments herein presented ran on a PC equipped with a CPU Intel Core i7-2760QM @ 2.40GHz (8 cores) and 8GB of RAM, running Ubuntu 14.04 LTS.

NUSMV-INVAR rather than AALTA is twofold: While its performances are worse than AALTA, NUSMV-INVAR is more robust in the sense that it either reaches the time limit or it solves the problem, without ever failing for unspecified reasons like AALTA does at times; second, it turns out that NUSMV-INVAR can deal flawlessly and in reasonable CPU times with all the specifications we consider in this Section, both the intended one and the ones obtained by injecting faults. In particular, on the intended specification, NUSMV-INVAR is able to find a valid model for the specification in 37.1 CPU seconds, meaning that there exists at least a model able to satisfy all the requirements simultaneously. Notice that the translation time from patterns to formulas in $LTL(\mathcal{D}_C)$ is negligible with respect to the solving time. Our second experiment is to run NUSMV-INVAR on the specification with some faults injected. In particular, we consider six different faults, and we extend the specification in six different ways considering one fault at a time. The patterns related to the faults are summarized in Table 4.4. In case of faulty specifications, NUSMV-INVAR concludes that there is no model able to satisfy all the requirements simultaneously. In particular, in the case of F2 and F3, NUSMV-INVAR returned the result in 2.1 and 1.7 CPU seconds, respectively. Concerning the other faults, the tools was one order of magnitude slower in returning the satisfiability result. In particular, it spent 16.8, 50.4, 12.2, and 25.6 CPU seconds in the evaluation of the requirements when faults 1, 4, 5 and 6 are injected, respectively.

The noticeable difference in performances when checking for different faults in the specification is mainly due to the fact that F2 and F3 introduce an initial inconsistency, i.e., it would not be possible to initialize the system if they were present in the specification, whereas the remaining faults introduce inconsistencies related to interplay among constraints in time, and thus additional search is needed to spot problems. In order to explain this difference, let us first consider fault 2:

*Globally, it is always the case that if `state_init` holds,
then not `arm_idle` holds as well.*

It turns out that in the intended specification there is one requirement specifying exactly the opposite, i.e., that when the robot is in `state_init`, then `arm_idle` must hold as well. Thus, the only models that satisfy both requirements are the ones preventing the robot arm to be in `state_init`. However, this is not

possible because other requirements related to the state evolution of the system impose that `state_init` will eventually occur and, in particular, that it should be the first one. On the other hand, if we consider fault 6:

*Globally, it is always the case that if `arm_moving` holds,
then `joint1.speed > 15.5` holds as well.
Globally, `arm_moving` and `proximity_sensor = 10.0`
eventually holds.*

we can see that the first requirement sets a lower speed bound at 15.5 *deg/s* for `joint1` when the arm is moving, while there exists a requirement in the intended specification setting an upper speed bound at 10 *deg/s* when the proximity sensor detects an object closer than 20 *cm*. In this case, the model checker is still able to find a valid model in which `proximity_sensor < 20.0` never happens when `arm_moving` holds, but the second requirements in fault 6 prohibits this opportunity. It is exactly this kind of interplay among different temporal properties which makes NUSMV-INVAR slower in assessing the (in)consistency of some specifications.

Chapter 5

Requirements-Based

Black-Box Testing

In Chapter 4 we presented a way to formalize and check the consistency of a set of requirements, using $LTL(\mathcal{D}_C)$ as the underline formal logic. In this Chapter we deal with the problem of checking whether a reactive system, *i.e.* a system that maintains an ongoing interaction with its environment [MP12], conforms to such requirements through testing. The work presented in this chapter refers to standard LTL as the underling specification logic (see Definition 2.2.1), keeping in mind that $LTL(\mathcal{D}_C)$ can be reduced to LTL with the encoding presented in the previous chapter. Furthermore, we assume the SUT to be accessible for testing, *i.e.*, we can execute inputs and observe outputs, but that no internal representation of the system is available. This problem arises in a variety of contexts, e.g., when a system is developed by integrating commercial off-the-shelf (COTS) components [LCB⁺09]. In these scenarios, techniques such as model checking [BK08] or (white-box) model-based testing [UL07] are ruled out. Also, classical black-box techniques like random testing, equivalence partitioning or boundary analysis [Bur06] either do not take into account the specification or require manual effort to assemble meaningful test suites.

Our approach is inspired by [AGR13], but aims to deal with a more general class of properties. Our methodology is based on a visit of the Büchi automaton corresponding to the requirements. The visit starts from the initial state of

the automaton and generates a sequence of input values with which the black-box system is fed to obtain a corresponding sequence of output values. We check such input/output sequence against the automaton, i.e., we check whether there exists at least one state in the automaton that can be reached along the sequence. If there is no such state, then the system is not conformant to the requirements and the sequence provides a counterexample. Otherwise, we can continue the generation of the sequence by iterating the above steps until either (i) an acceptance state of the automaton is reached with a sequence of length at least k_{min} or (ii) an acceptance state cannot be reached with a sequence of length at most k_{max} , where k_{min} and k_{max} are two parameters such that $k_{min} < k_{max}$. Multiple tests can be obtained by iterating this procedure until all the reachable transitions have been visited at least once.

We evaluate our approach in three different experimental settings. In the first one we consider benchmarks taken from the LTL Track of the 2018 edition of the Reactive Synthesis Competition (SYNTCOMP 2018)¹ and we compare our approach with the one described in [AGR13]. In the second setting we use the Adaptive Cruise Control (ACC) prototype implemented in [AHDR18] and we compare the tests generated by our approach with those generated with a model-based generation strategy. In the third setting we test the model of a robotic arm controller in order to evaluate our approach on a large set of requirements coming from an industry-grade prototype. In the two former settings we use a mix of fault-injection [HTI97] and mutation analysis [ABLN06] in order to compare different approaches. In the third setting we inject faults manually. The results we obtained with our experiments show that our approach can outperform the one in [AGR13] by finding more induced faults. Furthermore, generating tests based on the specification can be as effective as approaches based on the system model, discovering almost the same number of faults. Finally, our approach can be effective in finding faults in small-to-medium sized industry-grade systems.

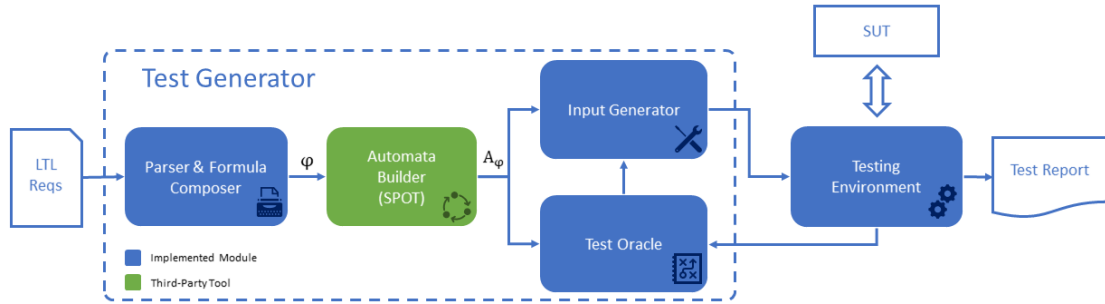


Figure 5.1: The main workflow of our approach.

5.1 Automatic Test Case Generation from LTL specification

In order to test black-box systems, our approach adopts the workflow presented in Figure 5.1. We assume that the specification is composed of a list of LTL formulas, the declaration of the set I of input propositions, and the set O of output propositions such that $I \cup O = AP$ and $I \cap O = \emptyset$. The “Test Generator” pipeline in Figure 5.1 has the goal to produce a set of valid tests to execute on the system under test (SUT). The pipeline comprises four components:

- **Parser** reads the input specification, creates the intermediate data structures and builds the conjunction of requirements.
- **Automata Builder** builds a Büchi or equivalent automaton representation of the input specification.
- **Input Generator** chooses which inputs to execute on the SUT.
- **Test Oracle** evaluates the output produced by the SUT and checks if it satisfies the specifications.

Testing Environment is responsible for orchestrating the interaction between the components. It queries Input Generator for new inputs to test and it executes them on the SUT. Testing Environment collects the output and passes it to Test Oracle for evaluation. If the test is complete, Testing Environment stores the final verdict and resets the environment to start a new test. Moreover, the Test Oracle provides to the Input Generator the set of possible states in which the

¹<http://www.syntcomp.org/>

automaton can currently be, given the executed trace. Notice that the SUT is supposed to run synchronously with the **Testing Environment** and it should return to its initial state when a reset command is received. Where necessary, the user can provide a software layer to abstract the real SUT and implement these features (see Section 6.1.3.1 for more details).

In the following, we present each step of our implementation in more detail.

5.1.1 Requirements and Automata Processing

The input of the test generator algorithm is a set $R = \{\phi_1, \dots, \phi_n\}$ of LTL formulas along with the list of input and output variables. The parser reads the input formulas as a conjunction $\Phi = \phi_1 \wedge \dots \wedge \phi_n$ to build the corresponding automaton. We rely on SPOT [DLLF⁺16] to perform the construction of the Büchi automaton represented as a directed graph. Before test generation starts, we preprocess the automaton by expanding the edges where SPOT groups different equivalent assignments to move from one state another, to obtain exactly one assignment for each edge. During preprocessing, variables are omitted if they are not relevant for a particular transition, e.g., if the transition is enabled independently from their value. In such cases, we set the input variables to false by default, while we leave the outputs unchanged. This is because we want to have a fully defined and deterministic input, but we do not want to impose additional constraints that are not specified by the requirements on the outputs. Other choices are possible; for example, one could set the undefined inputs randomly or could copy the value of such variables from previous assignments, if any.

5.1.2 Test Oracle

The aim of the test oracle is to decide if a trace τ , composed of input and output variables, is correct with respect to the given LTL specification Φ . A more permissive check, often considered for runtime monitoring, consists in verifying that τ is a valid prefix of the language $Words(\Phi)$. This can be done by checking that there exists a run induced by τ on the automaton A_Φ , or, equivalently, using monitors. This kind of check is useful to identify violations of safety properties, but it is ineffective for liveness ones, even for the the co-safety subclass. For example, we cannot detect violations of the formula $\phi = \diamond a$ with a monitor,

because every prefix is valid as long as the proposition a becomes true eventually. In order to solve this issue, a number of different LTL semantics for finite traces have been proposed, such as *FLTL*[MP12], *LTL[≠]*[EFH⁺03], *LTL₃*[BLS06] and *LTL-RV*[BLS10]. In [BBNR18] the authors propose a *counting semantics* making predictions based on the number of steps necessary to witness the satisfaction or violation of a formula. Evaluations under such semantics can range from a 2-valued verdict – namely *True* (\top) or *False* (\perp) – to a 5-value one; *True* (\top), *Presumably True* (\top_P), *Inconclusive* (?), *Presumably False* (\perp_P) and *False* (\perp). The choice of the semantics defines the specific kind of conformance to the specification adopted and implemented by the test oracle. In the following, we rely on the FLTL semantics, formalized below in Definition 5.1.1 — for a discussion of different semantics, we refer the reader to [BLS10].

Definition 5.1.1. Given a finite word (or trace) τ of length n and an FLTL formula ϕ , $\tau(= \tau, 0)$ satisfies ϕ , denoted as $\tau \models \phi$, under the following conditions (s.t. $0 \leq i < n$):

$$\begin{aligned}
 \tau, i \models p \in AP & \quad \text{iff } a \in \tau[i] \\
 \tau, i \models \neg\phi & \quad \text{iff } \tau, i \not\models \phi \\
 \tau, i \models \phi_1 \wedge \phi_2 & \quad \text{iff } \tau, i \models \phi_1 \text{ and } \tau, i \models \phi_2 \\
 \tau, i \models \mathcal{X}\phi & \quad \text{iff } (i + 1 < n) \text{ and } \tau, i + 1 \models \phi \\
 \tau, i \models \mathcal{N}\phi & \quad \text{iff } (i + 1 \geq n) \text{ or } \tau, i + 1 \models \phi \\
 \tau, i \models \phi_1 \mathcal{U} \phi_2 & \quad \text{iff } \exists i \leq j < n. (\tau, j \models \phi_2 \wedge \forall i \leq m < j. (\tau, m \models \phi_1)) \\
 \tau, i \models \diamond\phi & \quad \text{iff } \exists i \leq j < n. (\tau, j \models \phi) \\
 \tau, i \models \square\phi & \quad \text{iff } \forall i \leq j < n. (\tau, j \models \phi)
 \end{aligned}$$

Regarding the boolean operators, FLTL semantics coincides with the standard LTL semantics on infinite words. However, with temporal operators, such as \mathcal{X} and \mathcal{U} , there is a difference concerning the maximum length of the word. In particular, the semantics distinguishes between a strong next operator \mathcal{X} , which require a next time step to exists, and a weak version \mathcal{N} , which it is always satisfied at the last step of a trace. In our requirements, however, we only make use of the strong variant. In our approach, the FLTL oracle is implemented on an automaton and traces are checked directly on the generated Büchi Automa. We posit that every trace τ ending in an acceptance state q^* of the Automata A_Φ , also satisfies the formula Φ from which the automaton is

built.

5.1.3 Input Generator

The main idea behind the generation of input sequences for testing the SUT consists in exploring different paths of the automaton A_Φ that represents the specification. Given a choice of (i) an exploration strategy to prioritize paths and (ii) a termination condition to end the search, we obtain our algorithm Guided Depth First Search (GDFS) presented in Algorithm 3. As the name suggests, it is a variant of the classical depth-first search algorithm on directed graphs.

The algorithm takes as input the automaton A_Φ , the interval k_{min} and k_{max} , i.e., the minimum and the maximum length of each trace, the *oracle* object and the environment *env* object. The algorithm starts with the initialization of the *visitCounter* map, that counts how many times an edge has been explored (lines 2-5). Notice that only the outgoing edges from the initial state are initialized, while the other ones are incrementally added during the exploration (lines 11 - 13). The algorithm terminates when all the edges in *visitCounter* have been visited at least once. At the beginning of each test, the trace τ is initialized to an empty word and the current state s_c is initialized to the initial state of the automaton (lines 7-8). Then the environment is reset to start at the initial state (line 9). The test is computed by iteratively choosing an edge (line 14), extracting the input on its label (line 15), executing it on the SUT by means of the *env* object (line 19) and using the output to choose the successor state, if any, and to build the trace τ (lines 20 - 21). The function `selectNextEdge` chooses the next state to execute by selecting the edge with less visits so far. In case of multiple edges with the same score, it sorts them with an heuristics that takes into account the distance from the nearest acceptance state and the degree of the target state. Moreover, the *visitCounter* is updated after each choice (lines 16 - 18) by increasing the counter of all edges leaving s_c that present the input i . This is a small optimization to reduce the number of steps necessary to terminate, because many edges could produce the same input but expect different accepted outputs. From an input point of view, these edges are equivalent, but only one of them will be traversed, depending on the produced

Algorithm 3: Guided Depth First Search

```

1: function GDFS( $A_\Phi, k_{min}, k_{max}, oracle, env$ )
2:    $visitCounter \leftarrow \text{EMPTYMAP}()$ 
3:   for  $e \in A_\Phi.\text{OUTGOINGEDGES}(A_\Phi.\text{initState})$  do
4:      $visitCounter[e] \leftarrow 0$ 
5:   end for
6:   while  $\exists e \in visitCounter.(visitCounter[e] == 0)$  do
7:      $\tau \leftarrow \{\}$ 
8:      $s_c \leftarrow A_\Phi.\text{initState}$ 
9:      $env.\text{RESET}()$ 
10:    while  $oracle.\text{VALIDPREFIX}(\tau) \wedge |\tau| < k_{max}$  do
11:      for  $e \in A_\Phi.\text{OUTGOINGEDGES}(s_c) \wedge e \notin visitCounter$  do
12:         $visitCounter[e] \leftarrow 0$ 
13:      end for
14:       $e \leftarrow \text{SELECTNEXTEDGE}(A_\Phi, s_c, visitCounter)$ 
15:       $i \leftarrow \text{GETINPUT}(e)$ 
16:      for  $e \in A_\Phi.\text{OUTGOINGEDGES}(s_c) \wedge \text{GETINPUT}(e) == i$  do
17:         $visitCounter[e] \leftarrow visitCounter[e] + 1$ 
18:      end for
19:       $o \leftarrow env.\text{PERFORMACTION}(i)$ 
20:       $s_c \leftarrow \text{GETSUCCESSOR}(A_\Phi, s_c, i \cup o)$ 
21:       $\tau.\text{APPEND}(i \cup o)$ 
22:      if  $|\tau| \geq k_{min} \wedge s_c \in A_\Phi.\text{acceptanceStates}$  then
23:        break
24:      end if
25:    end while
26:     $res \leftarrow oracle.\text{EVALUATE}(\tau)$ 
27:     $env.\text{ADDTTEST}(\tau, res)$ 
28:  end while
29: end function

```

output. Termination of a test occurs exactly when one of the following three cases is true: (i) τ is no more a valid prefix of $\mathcal{L}(A_\Phi)$ and therefore the test

failed; (ii) the length τ reached the maximum length k_{max} ; (iii) the length of τ is greater than k_{min} and the exploration reached an acceptance state. At the end of each test, the oracle gives its final verdict and the result is stored in the *env* object (lines 26 - 27).

5.2 Experimental Analysis

We present the results of three experiments² involving the framework previously introduced. In the first one, we aim to assess the quality of the generated test suite involving a set of benchmarks borrowed by the LTL Track of the Reactive Synthesis Competition 2018³ (SYNTCOMP 2018). The second experiment aims to compare the effectiveness of our approach with respect to model-based strategies; in order to do that, we consider the use case of an Adaptive Cruise Control System made available in [AHDR18] and we compare our algorithm with state-of-the-art model-based approaches when it comes to spotting erroneous mutants. Finally, our last experiment aims to evaluate the scalability of our approach in a real world use case. So, we consider a set of requirements from the design of an embedded controller for a robotic manipulator used in the context of the EU project CERBERO⁴ [MPM⁺17, PFS⁺19]. The experiments described in the following ran on a workstation equipped with an Intel Xeon E31245 @ 3.30GHz CPU and 32GB RAM running Ubuntu 18.10 64bits. For all the experiments, we granted a time limit of 600 CPU seconds (10 minutes) and a memory limit of 30GBs.

5.2.1 Syntcomp Benchmarks

The set of benchmarks we consider is the one provided for the LTL Track of the Reactive Synthesis Competition 2018. We first translate the TLSF [JKS16] specifications into equivalent LTL ones accepted by our tool. Note that we do not use SyFCo, a tool for manipulating and transforming TLSF specifications in other existing specification formats for synthesis, because we handle ASSUME formulae in a different way. In particular, SyFCo would translate ASSUME

²All benchmarks are available at <https://gitlab.sagelab.it/sage/benchmarks-tests>

³<http://www.syntcomp.org/>

⁴<http://cerbero-h2020.eu>

formulae as preconditions (left-hand side of an implication) and the ASSERT and GUARANTEE formulae as postconditions (right-hand side of an implication). Therefore, if an ASSUME formula is violated, the system is not required to satisfy the given requirements. This behavior would lead to many useless tests, because whenever an assumption is falsified during the test execution, the specification would be trivially satisfied and no constraint would be enforced on the output. In order to solve this problem, we require the ASSUME part to be satisfied together with the ASSERT and GUARANTEE part, i.e., we replace implication with conjunction. We refer the reader to [JKS16] for more details on the standard translation from TLSF to LTL. We exclude benchmarks whose output assignments appear in the ASSUME part of the specification. This is because, as explained before, we require the assumptions to hold during the execution of the test, but assumptions containing outputs can always be falsified, thus failing the test. We synthesize Mealy machines for the specifications with `Strix` [ML18], the winner of the SYNTCOMP 2018 competition, and we exclude benchmarks for which `Strix` times out in 600 CPU seconds. For each synthesized Mealy machine, we compute 100 mutants randomly applying one of the following rules:

- change the target state of a random transition to a different one;
- flip the output value of a variable on a random transition, namely setting it to *false* if it was *true* and vice-versa.

We apply only one mutation per mutant because the synthesized models are usually small in size and one variation is often enough to expose a violation of the specification. However, some of the resulting mutants may still be correct with respect to the corresponding specification. At the end of this process we have 128 different benchmarks, each of those with 100 mutants. In the experiment, we compare the results obtained with 5 different algorithms. GDFS-1, GDFS-3 and GDFS-5 are the algorithm described in Section 5.1 with k_{min} set to 1, 3 and 5, respectively. For comparison purpose, we also re-implemented, – and generalized to fit our framework – the algorithm presented in [AGR13]. Briefly, the algorithm traverses the monitor automaton of the specification during the test execution, and stops when a coverage criteria is fulfilled. A test is concluded

either when an objective is reached or when the maximum length k_{max} of the trace is reached. In [AGR13] two strategies are proposed, namely Random Walk (RW) and Guided Walk (GW) and we implemented and tested both of them. As for the coverage criteria, we implemented what they call *Atomic Proposition Coverage* (APC), i.e., each atomic proposition on each transition of the monitor must be covered. For each algorithm we set k_{max} equal to 100 and we stop the execution as soon as a test fails and the mutant is killed. Notice that 600 CPU seconds are allotted to each benchmark, including automata processing and evaluation of all mutants.

Figure 5.2 (left) shows the number of mutants killed per benchmark by each algorithm, ranging from 0 to 100. Figure 5.2 (right) shows the average number of steps executed, namely the sum of the length of each test, averaged over the mutants. In both charts, the abscissa represents the number of benchmarks, while the ordinate shows the number of mutants killed (left) and the number of steps executed (right). Notice that, since the results of RW and GW can vary due to non-deterministic behaviors, we execute the test 3 times and we report the median value as reference for these two algorithms. The results reveal that GDFS-5 clearly outperform all the other algorithms in terms of total amount of mutants killed, and that the number of executed steps is only slightly higher than GDFS-1 and GDFS-3. However, only for two benchmarks all the 100 mutants have been killed. Moreover, in 25 cases it did not kill any mutant, 15 of which due to timeouts. Regarding RW and GW, they both revealed totally ineffective for 73 of the 129 benchmarks, although only 2 timeouts occurred. However, looking at Figure 5.2 (right) we notice that in 59 of these benchmarks, the two algorithms did not perform any testing at all. This phenomenon is due to the nature of the benchmarks involved, where the specification only contains liveness properties and the monitor is a single state automaton accepting all prefixes.

5.2.2 Adaptive Cruise Control

In our second experiment we consider the Adaptive Cruise Control (ACC) prototype implemented in [AHDR18]. The ACC system adjusts the current velocity of the vehicle towards a target cruise velocity defined by driver. If the vehicle

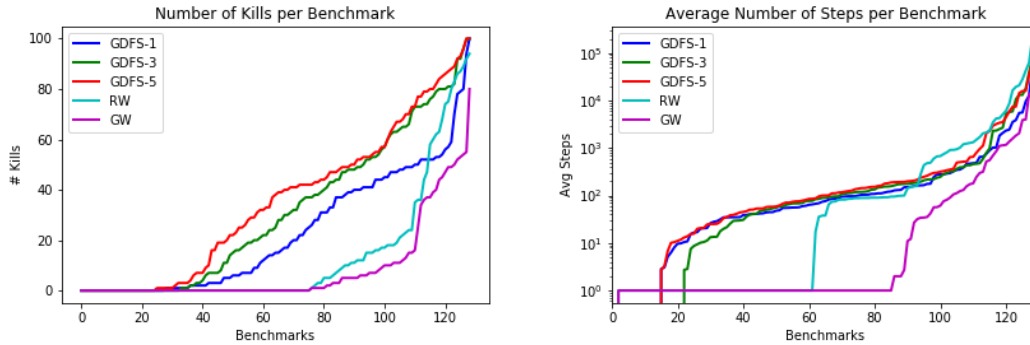


Figure 5.2: Total amount of mutants killed (left) and average number of steps (right) computed by the considered algorithms in the set of SYNTCOMP 2018 benchmarks.

gets too close to the forward vehicle, the ACC system must adjust the current distance between the two and maintain a certain safety distance. Additionally, the driver can intervene by: (1) activating the system via an ACC button; (2) deactivating the system via the ACC button; and (3) deactivating the system by braking or accelerating the car. The authors of [AHDR18] also generated test cases from LTL requirements using three different requirements coverage criteria: requirements coverage (RC), antecedent coverage (AC), and unique first cause coverage (UFC). Tests are generated with a model-based generation strategy: trap-properties are built from requirements, and a counterexample is produced with a model checker. The algorithms are evaluated with 524 mutants of the correct implementation.

The goal of the experiment here described is to compare the performance of our algorithm with respect to model-based techniques that make explicit use of a model to generate test cases. We modified slightly the set of requirements, reducing numerical comparisons and enums (available in the NuSMV [CCG⁺02] models used in [AHDR18]) to boolean variables. This is a mere syntactic variation to represents LTL formulae in the default syntax as described in Section 2.2. The resulting specification is composed of 12 requirements, 6 input and 10 output variables. The results are depicted in Table 5.1. In order to ease the comparison with the model-based approach, we also report the results from [AHDR18]. The results show that the GDFS algorithm performances are com-

	RC	AC	UFC	GDFS-1	GDFS-3	GDFS-5
Number of Test Cases	6	7	18	26	4912	2597
Branch Coverage (%)	78.3	78.3	86.7	45.0	70.0	71.7
Number of Killed Mutants	488	488	488	414	480	480
Killed Mutants (%)	93.1	93.1	93.1	79.0	91.6	91.6

Table 5.1: Experimental results on the ACC use case.

parable to the model-based algorithms, with a difference of only 8 mutants (1.5% of the total) for k_{min} equal 3 or 5, at the expense of many more tests. Notice however that the test generation and execution is still quite small; it takes about 1 second to run GDFS-1, 11 seconds for GDFS-3 and 5 seconds for GDFS-5. Moreover, the whole test suite is executed only if all tests succeed, but if a failure is detected it can terminate much earlier. In the case of GDFS-5, for example, the average number of tests executed per mutant is 329, much lower than the test suite size (2597). However, despite the large test suite, GDFS reaches a lower branch coverage than the model-based counterparts, stopping at 71.7%. Also notice that, in this context, with all requirements being safety properties, the RW algorithm described in the previous experiment performs well, achieving similar results to GDFS-5 (although with some variation due to randomness). These results show that the black-box testing with the framework presented in Section 5.1 can be almost as effective as model-based techniques, where more manual work is required to model the system. A final remark on the k_{min} and k_{max} parameters of the GDFS algorithm is in order. As shown in Table 5.1, k_{min} plays an important role in the test suite size and performance. In our experience, the longer the test, the more the automaton is covered and the less transitions close to the initial state are repeated. Similarly, also k_{max} can influence a test suite size and performance: an excessively small value could lead to some false positive tests, while an excessively large value could produce unnecessarily long tests before declaring them failed. However, the generated test suite depends not only on the algorithm and the specification, but also on the SUT behavior. The optimal values of such parameters is context dependent, and may require some fine tuning.

5.2.3 Robotic Manipulator

Our last experiment considers a set of requirements from the design of an embedded controller for a robotic manipulator. The controller should direct a properly initialized robotic arm — and related vision system — to look for an object placed in a given position and move to such position in order to grab the object; once grabbed, the object has to be moved and released into the bucket without touching it. The robot must stop also in the case of an unintended collision with other objects or with the robot itself — collisions can be detected using torque estimation from current sensors placed in the joints. Finally, if a general alarm is detected, e.g., by the interaction with a human supervisor, the robot must stop as soon as possible. The manipulator is a 4 degrees-of-freedom Trossen Robotics WidowX arm⁵ equipped with a gripper. The design of the embedded controller is part of the activities related to the “Self-Healing System for Planetary Exploration” use case in the context of the EU project CERBERO. In this case the specification is composed of 31 requirements, 3 inputs and 11 outputs. The SUT is implemented as an smv model. With GDFS-5 ($k_{min} = 5$ and $k_{max} = 30$), we obtain 1441 tests and a total of 12867 steps executed in 1171 seconds. At each step, NuSMV [CCG⁺02] is called in order to determine the evolution of the system. Then, we manually inject faults by removing some constraints in the guards (forcing the system to evolve from one state to another) or by modifying value assignments of some variables. At the end, we obtain 10 different NuSMV faulty models. We show the results of this analysis in Table 5.2. First, we report that a failed test has been detected in all considered cases. Looking at the Table, we can observe that, for each bugged system, a small number of tests is necessary to discover the failure. Therefore, in most cases, it is not necessary to perform a complete exploration of the automaton and an early stopping strategy can save substantial time when debugging an application.

⁵<http://www.trossenrobotics.com/widowxrobotarm>.

# Injection	# Tests	# Steps	Time(s)
1	1	2	7.64
2	2	14	8.61
3	2	14	8.74
4	1	2	7.75
5	1	7	8.15
6	4	25	8.61
7	56	502	25.23
8	1	3	8.15
9	1	6	7.84
10	2	10	8.17

Table 5.2: Fault-Injection results on the robotic manipulator use case.

Chapter 6

Tools

The use of formal methods in requirements engineering is an enabler to achieve automation and formal guarantees along the system design life-cycle. In Chapter 4 and Chapter 5 we have seen how to automatically check the consistency of a set of requirements, find inconsistent ones, and use them to test a system that is supposed to implement them. However, formal methods require a high degree of specialization and training, making it difficult to apply them in practice. In order to help non-expert users to adopt these techniques, we need to provide tools to ease their job. To this end, we designed and developed a suite of tools that implements the algorithms presented before and provides easy to use interfaces to guide the user along the process.

The rest of the chapter is structured as follow. In Section 6.1 we introduce the SPEC_{PRO} library, containing the implementation of the main algorithms described in the previous chapters, and we illustrate some of its available APIs. In Section 6.2 we present REQ_V, a tool for the management and analysis of requirements expressed as PSPs. Finally, in Section 6.3 we present REQ_T, a tool for the automatic test generation and evaluation on a provided system under test.

6.1 SpecPro

SPECPRO is an open-source¹ Java library that implements all the algorithms presented in this work and provides classes and data structures for an easy access to the core functionalities. It is meant to ease the application of formal methods for the analysis of requirements, from a developer perspective. Requirements can be expressed either as Property Specification Patterns (PSPs) (defined in Section 2.4) with numerical constraints (see Section 4.1 for more details) or as LTL formulae (defined in Section 2.2).

6.1.1 Parse And Translate Requirements

With SPECPRO it is possible to easily perform a variety of tasks involving LTL requirements. First of all, in order to parse input requirements, and build a `LTLSpec` object, we have to instantiate an object implementing the abstract class `AbstractLTLFrontEnd`. SPECPRO provides two default implementations: `PSPFrontEnd` and `LTLFrontEnd` to read Property Specification Patterns (PSPs) and Linear Temporal Logic (LTL) formulae, respectively.

Listing 6.1: Parse file with requirements in PSP format

```
AbstractLTLFrontEnd fe = new PSPFrontEnd();
LTLSpec spec = fe.parseFile("input.req");
```

The `AbstractLTLFrontEnd` class also provides other auxiliary methods such as `parseString` and `parseStream` to parse a specification stored in a `String` or a generic `InputStream`. The `LTLSpec` class is a very important component into the SPECPRO library because it contains many data structures that are helpful to perform all the other tasks. Moreover, `PSPFrontEnd` produces a object of type `LTLDCSpec`, which directly inherit from `LTLSpec`, that provides additional data structures and methods to handle the numerical constraints and facilitate the conversion between numeric and boolean variables.

An `LTLSpec` instance can be translated into a variety of formats for off-the-shelf model checkers and LTL satisfiability solvers. For example, to translate a `LTLSpec` into a NuSMV specification we instantiate a `NuSMVTranslator` object:

¹<https://gitlab.sagelab.it/sage/SpecPro>

Listing 6.2: Translate a LTLSpec object into a NuSMV specification file

```
LTLSpec spec = ...
OutputStream outputStream = new OutputStream("output.smv");
NuSMVTranslator translator = new NuSMVTranslator();
translator.translate(outputStream, spec);
```

Other available translators are AALTATranslator, SpotTranslator, PandaTranslator, PtlMupTranslator and TRPUCTranslator. The user can add new translators simply extending the LTLToolTranslator abstract class and implementing the translate method.

6.1.2 Consistency Checking

It is possible to check the consistency of requirements directly in Java, implementing the ModelChcker abstract class and instantiating the ConsistencyChecker class provided by the library. SPECPRO provides also two default implementations of ModelChecker: NuSMV and Aalta. In order to use them, it is necessary to set SPECPRO_AALTA and SPECPRO_NUSMV environment variables, respectively, indicating the paths of the model checkers location in the file system.

Listing 6.3: Check the consistency of an LTLSpec object

```
LTLSpec = ...
ModelChecker mc = new Aalta();
ConsistencyChecker consistencyChecker = new
    ConsistencyChecker(mc, spec, "out.temp");
ConsistencyChecker.Result result =
    consistencyChecker.run();
if(result == Consistency.Result.CONSISTENT) {
    System.out.println("Requirements are consistent");
} else {
    System.out.println("Requirements are inconsistent");
}
```

Similarly, if a specification is inconsistent, SPECPRO provides the abstract class InconsistencyFinder that aims at finding a minimal subsets of requirements that explain the inconsistency, also called Minimal Unsatisfiable

Core (MUC). SPECPRO provides two implementations of this class (a detailed explanation of the two algorithms and their performances is provided in Section 4.2): `LinearInconsistencyFinder` and `BinaryInconsistencyFinder`.

Listing 6.4: Find a MUC of an inconsistent LTLSpec object

```
LTLSpec = ...
ModelChecker mc = new Aalta();
ConsistencyChecker consistencyChecker = new
    ConsistencyChecker(mc, spec, "out.temp");
InconsistencyFinder muc = new
    BinaryInconsistencyFinder(consistencyChecker);
List<InputRequirement> reqs = muc.run();
if(reqs == null) {
    System.out.println("Fail occurred during model
        checking call.");
    System.out.println(mc.getMessage());
} else {
    System.out.println("# MUC of " + reqs.size() + "
        elements found: ");
    for (InputRequirement r : reqs) {
        System.out.println(r.getText());
    }
}
```

6.1.3 Testing

In order to test black-box systems, SPECPRO implements the framework presented in Chapter 5. The System Under Test (SUT) is the system we want to test, and SPECPRO interacts with it during execution, probing some inputs and evaluating the produced output. The tests are generated starting from a `LTLSpec` that is assumed to be consistent. In addition to the list of requirements, the specification has to indicate the list of input and output atomic variables. To generate test cases for a model or a system, we have to implement the following steps:

1. Build the `LTLSpec` and an automaton representation of it, using the `LTL2BA` class. This operation requires `SPOT` to be installed on the machine in which the code is running.

Listing 6.5: Build a Büchi Automaton from a `LTLSpec`

```
LTLFrontEnd fe = new LTLFrontEnd();
LTLSpec spec = fe.parseFile("file.ltl");
LTL2BA lt12ba = new LTL2BA();
lt12ba.setType(LTL2BA.AutomatonType.NBA);
BuchiAutomaton automaton = lt12ba.translate(spec);
automaton.expandEdges();
```

2. Instantiate the class responsible for the tests generation. It requires the Büchi Automaton built in the previous step and a list of input variables (the algorithm is described in Section 5.1)

Listing 6.6: Instantiate a `GDFSTestGenerator` class and set k_{min}

```
GDFSTestGenerator testGenerator = new
    GDFSTestGenerator(automaton,
        spec.getInputVariables());
testGenerator.setMinLength(2);
```

3. Create an instance of `SUT`, *i.e.*, an interface for the System Under Test. `SPECPRO` provides the `MealyMachineSUT` default implementation to test models in the `KISS` format. However, it is usually preferable to implement your own `SUT` (see Section 6.1.3.1).

Listing 6.7: Instantiate a `SUT` implemented as a Mealy machine

```
MealyMachine mealy =
    MealyMachineBuilder.parseKISSFile(modelFile);
SUT sut = new MealyMachineSUT(mealy);
```

4. Finally, instantiate a `TestingEnvironment` and start the testing generation on the `SUT` previously instantiated.

Listing 6.8: Instantiate the `TestingEnvironment` and generate tests

```
TestingEnvironment environment = new
    TestingEnvironment(testGenerator, sut);
environment.setMaxTraceLength(10);

Map<Trace, TestOracle.Value> result =
    environment.runTests();
```

At the end of the process, `TestingEnvironment` will produce a map containing the executed traces and their evaluation. `TestingEnvironment` also provides the `setStopOnError` method that immediately stops the testing if a test fails during execution.

6.1.3.1 SUT

SPECPRO provides a flexible framework that allows to test any reactive system. In order to connect a custom SUT to the testing framework, the developer only have to provide an implementation of the `SUT` abstract class, defining its two methods.

Listing 6.9: Implementation of a custom SUT

```
import it.sagelab.specpro.models.ltl.Atom;
import it.sagelab.specpro.models.ltl.assign.Assignment
import it.sagelab.specpro.testing.SUT;

public class CustomSUT extends SUT {
    @Override
    public void reset() {
    }
    @Override
    public Assignment exec(Assignment input) {
        boolean inputVar =
            input.getAssignments().get("inputVar");
        ...
        Assignment output = new Assignment();
```

```
        output.add(new Atom("outputVar"), false);
        return output;
    }
}
```

The two methods to be implemented are:

- **reset:** it is called at the beginning of each test and it is meant to reset the SUT to its initial state;
- **exec:** it is called at every step of each test and takes in input an **Assignment** object containing the value for every input variable. The method's objective is to execute such input on the SUT and to return a new assignment containing the output values of the system.

6.2 ReqV

REQV is an open source² tool for the formal consistency checking of requirements. The main goal of the tool is to provide an easy-to-use environment to enable users with no background knowledge of formal methods and logic languages to write and verify requirements, expressed as a list of properties specification patterns (PSPs). It provides an intuitive interface, accessible within a web browser, and can automatically translate requirements in a formal representation and checks their inner consistency. In case of inconsistency, REQV can also extract a minimal set of conflicting requirements to help designers in correcting the specification.

6.2.1 Architecture

In order to provide an easy setup and usage, REQV has been designed as a web application. REQV's implementation relies on different open-source tools and frameworks and its architecture is outlined in Figure 6.1. The two main components are:

- **front-end:** it is a web application implemented in Typescript, using the

²Source code available at <https://gitlab.sagelab.it/sage/ReqV>

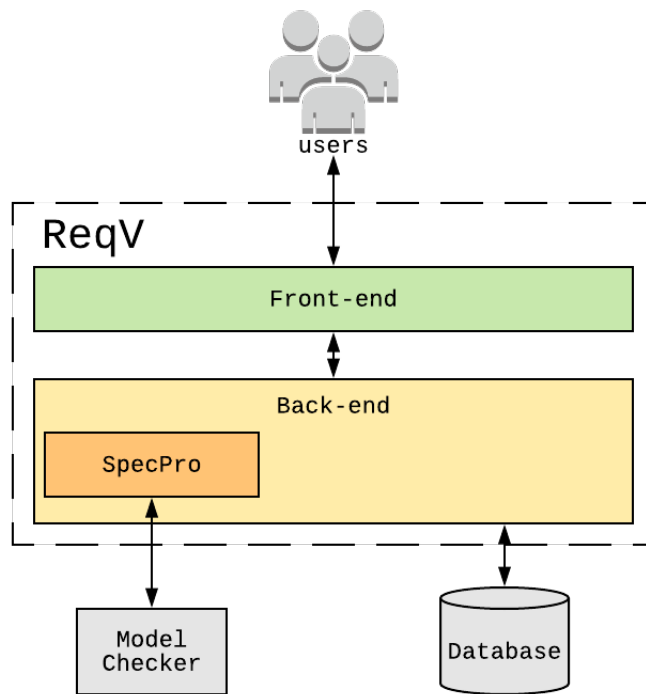


Figure 6.1: ReqV architecture diagram.

ANGULAR³ framework. It provides a graphical user interface for the user and performs asynchronous calls to the back-end.

- **back-end:** it is a Java server application based on the SPRING BOOT⁴ framework and POSTGRESQL⁵ 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[JBS15] open standard for authentication over HTTPS. The back-end also employs SPECPRO and the NuSMV model checker to provide the main functionalities, upon which it builds an additional layer of functionalities.

REQV's back-end can be accessed by multiple users at the same time and provides 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.

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

³<https://angular.io/>

⁴<https://spring.io/projects/spring-boot>

⁵<https://www.postgresql.org/>

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. For a full list of APIs, the reader is redirected to <https://reqv.sagelab.it/api/swagger-ui.html>.

6.2.2 Workflow

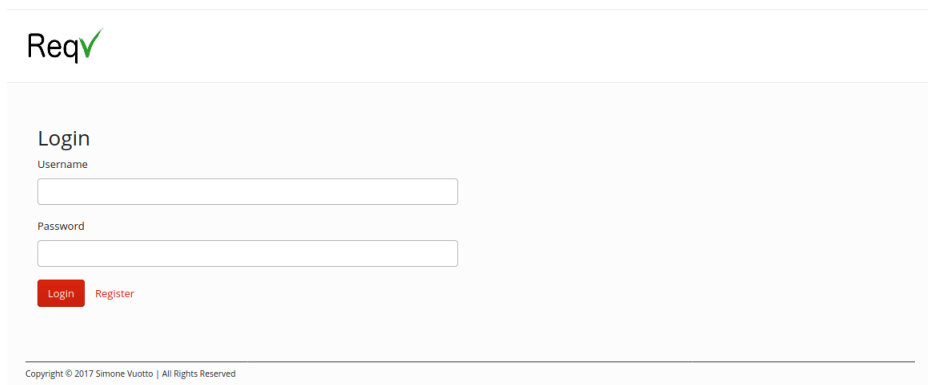


Figure 6.2: REQV login page.

In the workflow of REQV, the first step is authentication; when connecting for the first time to the REQV front-end, the user is redirected to the login page (shown in Figure 6.2). In order to continue, the user has to insert a valid username and password pair. If the login succeed, REQV shows the list of projects that are associated with the authenticated user. It is possible to create a new project simply by clicking the “New Project” button and filling the information required by the form, as illustrated in Figure 6.3. The form requires:

- **Name**: the name of the project;

Figure 6.3: REQV page to create a new project.

- **Description:** an optional description of the project;
- **Type:** the type of requirements used in the project. At the moment only the option **PSP** is available, but new kind of requirements may be added in future releases of the software.

Clicking on the name of a project, the user opens a new page with the details of the chosen project. By default, the **Requirements** tab is selected, showing the list of requirements added in the project, as illustrated in Figure 6.4 (initially the list is empty). The user can select one or more requirements and use the top menu to delete, disable or enable previously disabled requirements. A disabled requirement won't be considered during the analysis of consistency. Clicking the "Add Requirement" button, it is possible to create and add a new requirement to the list. REQV provides a wizard to help the user write a PSP with the right syntax, as illustrated in Figure 6.5. It allows to choose the type of scope and pattern of the requirement, and provides an adequate number of fields to fill the gaps and complete the PSP.

Once the user is satisfied with the inserted requirements, he can switch to the **Tasks** tab and press the "Validate" button to start a check of consistency of the active requirements. The task may take some time, depending on the amount of requirements involved, running asynchronously on the back-end side. When it terminates, REQV will report the result (see Figure 6.6. If the specification results inconsistent, the user can launch another task to search a minimal

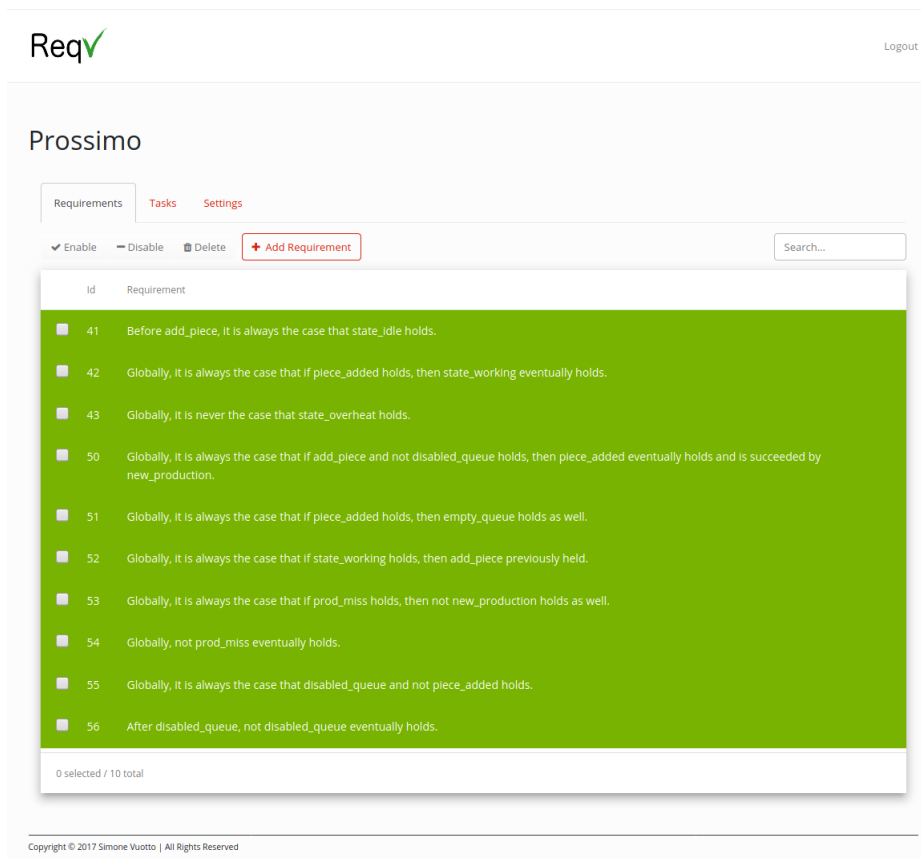


Figure 6.4: List of requirements inserted by the user in a project. Requirements colored in green are syntactically correct, the ones in red are not. A grey requirement indicates it has been disabled and therefore not considering during the analysis.

insatiable core. As before, the task can take some time to run, and a report with a MUC will be provided at completion (see Figure 6.7).

Finally, the **Settings** tab allows the user to change some project's data (*e.g.*, the title and description) and to import/export requirements with different file formats.

Figure 6.5: REQV page for requirements creation.

Figure 6.6: Report of the consistency checking task in REQV.

6.3 ReqT

ReqT is an open-source tool⁶ for the automatic testing of reactive black-box systems. It uses a formal specification to choose which action to perform on a system and to evaluate its response. Like REQV, it aims at providing an easy to use interface for non-expert users. ReqT is designed as a java desktop application, with a simple graphical user interface that allows the user to set up the environment, run the testing and explore the executed tests. ReqT exploits the SPECPRO's API described in Section 6.1.3. For this reason, it requires SPOT to be installed on the computer in which it is running.

⁶Source code available at <https://gitlab.sagelab.it/sage/ReqT>

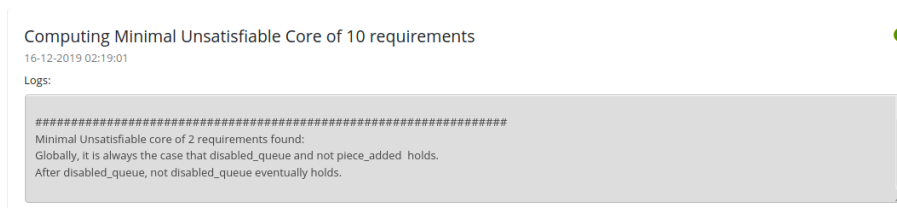


Figure 6.7: Report of the inconsistency explanation task. In the example REQV returns a MUC of 2 requirements.

6.3.1 Workflow

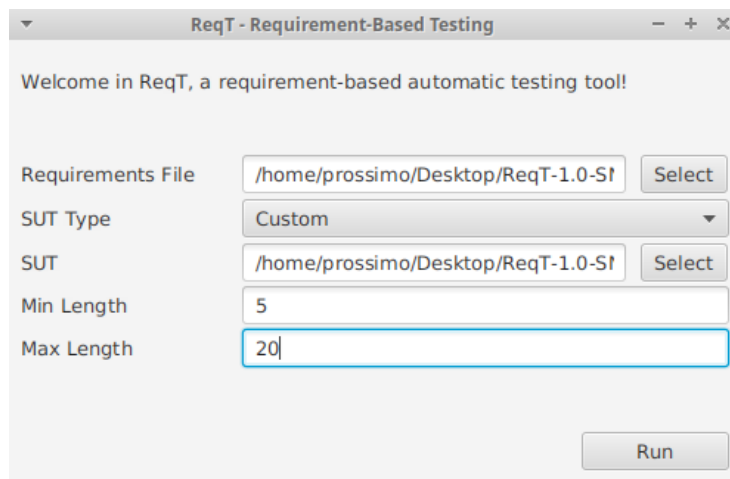
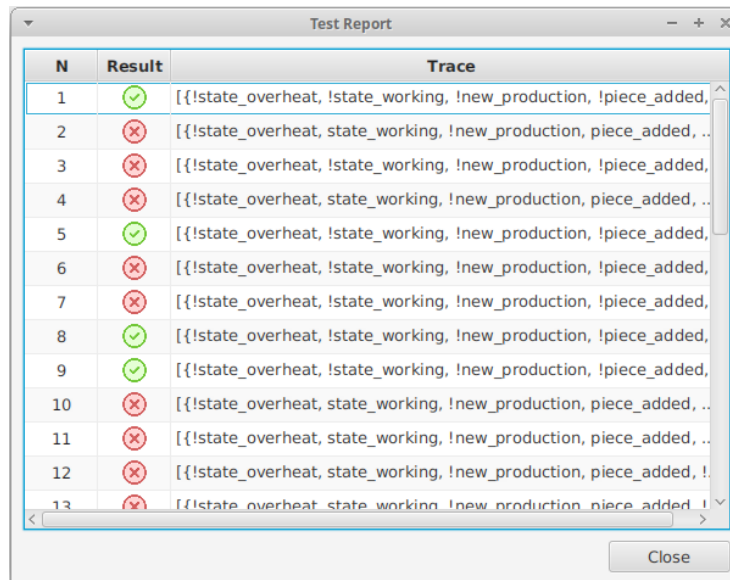


Figure 6.8: REQ-T's configuration window.

When executing REQ-T, it starts by showing to the user a configuration window, as depicted in Figure 6.8. In order to start the testing generation and execution on the SUT, the user has to fill the following information:

- **Requirements File:** The text file containing the requirements that the SUT must implement. The file should also contain the list of input and output variables, so that REQ-T knows how to build a test. REQ-T is able to process both requirements in PSP and LTL format, using the file extension to infer the type.
- **SUT Type:** the type of SUT to test. Currently three types are supported:
 - **Custom (Default):** the SUT is provided as a Java class implementing the SUT abstract class, as explained in Section 6.1.3.1. The class



N	Result	Trace
1	✓	[!state_overheat, !state_working, !new_production, !piece_added, ..
2	✗	[!state_overheat, state_working, !new_production, piece_added, ..
3	✗	[!state_overheat, !state_working, !new_production, !piece_added, ..
4	✗	[!state_overheat, state_working, !new_production, piece_added, ..
5	✓	[!state_overheat, !state_working, !new_production, !piece_added, ..
6	✗	[!state_overheat, !state_working, !new_production, !piece_added, ..
7	✗	[!state_overheat, !state_working, !new_production, !piece_added, ..
8	✓	[!state_overheat, !state_working, !new_production, !piece_added, ..
9	✓	[!state_overheat, !state_working, !new_production, !piece_added, ..
10	✗	[!state_overheat, state_working, !new_production, piece_added, ..
11	✗	[!state_overheat, state_working, !new_production, piece_added, ..
12	✗	[!state_overheat, state_working, !new_production, piece_added, !
13	✗	[!state_overheat, state_working, !new_production, piece_added, !

Figure 6.9: REQT’s tests report window.

can use other classes and libraries, as long as they are in the same directory of the SUT class.

- **KISS:** the SUT is provided as a Mealy Machine model saved in the KISS2 format [Yan91].
- **NuSMV:** the SUT is provided as a NuSMV [CCG⁺02] model. In this case, also the `SPECPRO_NUSMV` environment variable must be defined with the path to the NuSMV model checker.
- **SUT:** The file containing the SUT expressed in the format indicated in the *SUT Type* field.
- **Min Length:** indicates the minimum length for a test. It correspond to the k_{min} parameter described in Section 5.1.
- **Max Length:** indicates the maximum length for a test. It correspond to the k_{max} parameter described in Section 5.1.

When all the fields have been filled, the user can press the “Run” to start the test generation on the indicated SUT. The task may take some time, and a progress bar shows the number of tests executed so far. When the algorithm terminates, a new window appear, showing the tests report, ad depicted in Fig-

Test Number: 2
Status: **Fail**

Step #	prod_miss	add_piece	state_overheat	state_working	new_production
1	T	T	F	T	F
2	T	F	F	T	F
3	F	T	F	T	T
4	F	F	F	T	F
5	F	F	F	T	T
6	F	F	T	F	F

Close

Figure 6.10: REQ T's test details window.

Figure 6.9. The window presents the list of generated tests, with an icon indicating the result – a green icon for successful test and a red one for failed ones – and a summary of the executed trace. Double clicking on one row, the user can see the details of the test, as shown in Figure 6.10. The new window shows the test number, the status of the test and a table displaying the detailed test execution. Each row of the table represents a step of the test, and each column represents the value of a variable, the name of which is indicated in the header of the table. REQ T distinguishes the input and output variables, coloring the former in green and the latter in yellow. Finally, each cell contains a value; T for true and F for false.

Chapter 7

Conclusion

7.1 Summary of Contributions

We conclude this thesis with a summary of our contributions and a discussion of potential lines of research that might follow this work.

The first contribution (Chapter 4) consists in a study regarding the formalization and consistency checking of a set of requirements. To achieve this goal, we have extended basic PSPs over the constraint system \mathcal{D}_C , and we have provided an encoding from any $\text{PSP}(\mathcal{D}_C)$ into a corresponding LTL formula. This enables us to deal with the satisfiability of specifications of practical interest, and to verify them using state-of-the-art reasoning tools currently available for LTL. Noticeably, even considering the largest problem in our experiments ($\#vars = 640$, $\#dom = 32$), more than the 60% of the problems are solved (by AALTA) within the time limit of 10 minutes. Overall, using the specifications generated with our probabilistic model we have shown that our approach implemented on the tool AALTA scales to problems containing more than a thousand requirements over hundreds of variables. Considering a real-world case study in the context of the EU project CERBERO, we have shown that it is feasible to check specifications and uncover injected faults, even with tools other than AALTA. Moreover, we present and compare two algorithms for the extraction of high-level Minimal Unsatisfiable Cores (MUCs), namely any irreducible subset of requirements that is still unsatisfiable, from an inconsistent specification. In

particular, we show that our proposed dichotomic deletion-based algorithm is generally faster than the standard linear deletion-based one for specifications that contains small MUCs. A MUC could be extracted for all, but the largest specifications in our benchmark base.

The second contribution (Chapter 5) is a new approach to conformance testing of black-box reactive systems. We consider system specifications written as linear temporal logic formulas to generate tests as sequences of input/output pairs: inputs are extracted from the Büchi automaton representing to the specification, and outputs are obtained by feeding the inputs to the system. Conformance is checked by comparing input/output sequences with automata traces to detect violations of the specifications. In particular, the GDFS algorithm implements a variant of the deep-first search method, counting the visited transitions to evenly explore the automaton. A test is considered successful only if it can reach an acceptance state of the automaton with length in the interval between k_{min} and k_{max} , two parameters of the algorithm. We evaluated our approach across three different experimental settings. In the first setting we synthesized a set of benchmarks taken from the SYNTCOMP 2018 competition and we showed that our approach is better at finding mutants than (a generalization of) two different algorithms presented in [AGR13]. In the second setting, we showed that our approach compares favorably with state-of-the-art model-based techniques. Finally, in the third setting we tested a controller for a robotic manipulator modeled in SMV and we showed that our approach is able to find some manually injected faults.

Finally, the third contribution (Chapter 6) of this work consist in the implementation of three tools: SPECPRO, REQV and REQ T. SPECPRO ia a Java library that contains the implementation of all the algorithms discussed in this thesis and provides simple APIs for the developers. REQV and REQ T build uponl the capabilities provided by SPECPRO, implementing an additional layer of functionalities to provide a friendly interface to the end user. In particular, REQV is a web application that helps the user to write, manage and verify the consistency of requirements with minimal effort; while REQ T is a desktop application which is designed to help the user execute automated testing on a given SUT, using a formal specification to drive the test generation.

7.2 Open challenges and future work

During the study and realization of the work described in this thesis, we collected useful insights and ideas for possible extensions that have still to be explored. Therefore, we detail in the following some observations and ideas that could be useful for other researchers considering venturing in this field.

Logic Expressiveness In our work, we adopted linear temporal logic as our underline formalism for requirements. This choice was taken as a compromise between the logic expressiveness and the availability of off-the-shelf tools that are able to deal with this logic. In order to improve its expressiveness with atomic numeric constraints, we also proposed the $LTL(\mathcal{D}_C)$ encoding. However, during our study we found difficult to formalize in such logic some relevant properties for cyber-physical systems. For example, defining mutually exclusive states is possible but cumbersome, reducing the readability of the specification. Similarly, constraining some properties to be fulfilled within a given amount of time is not supported by standard LTL, but it can be achieved with some of its extensions (e.g., see MLTL [LVR19]). Furthermore, the atomic numerical constraints defined in $LTL(\mathcal{D}_C)$ resulted sufficient to formalize many relevant requirements, but more general linear arithmetic constrains are sometimes necessary. However, this easily leads to undecidability, as discussed in Section 4.1, so more research is needed in order to find the best trade-off. In particular, we think it may be interesting to investigate the extension of our work to some decidable fragments of first-order logic.

Structured Language The use of Property Specification Patterns (PSPs) is common in the literature because they can both provide a good level of readability and maintain a formal and unambiguous semantics. However, we found them a bit cumbersome and onerous for non-expert users to write such properties without the aid of a tool like REQV. We think that the language can still be improved without compromising its formal semantics. On the other hand, we believe that a full unrestricted natural language is not suited for the kind of tasks presented in this thesis, due to its intrinsic ambiguity that makes impossible to give formal guarantees.

Knowledge Base Sometimes requirement engineers rely on some common knowledge that is not always made explicit in the requirements document. Including domain specific knowledge bases at need would improve the effectiveness of our work, reducing at the same time redundant information and helping the engineer focusing on the specific problem at hand. Moreover, in some contexts, it could be useful to incorporate information from sources other than requirements. Therefore, we believe that the integration of an external knowledge base in the formalization and consistency checking of requirements can both improve the user experience and provide useful insights for debugging.

Testing Framework The testing framework presented in Chapter 5 showed to be effective and competitive with other state-of-the-art methods. Nonetheless, many aspects of the framework components can be customized and more research may lead to better results. In particular, we think that it could be interesting to implement and compare more test oracles, involving different finite LTL semantics, and new exploration strategies of the automaton.

Testing with Numeric Constraints It is possible to incorporate the encoding presented in Chapter 4 in the testing framework introduced in Chapter 5, to test SUTs with numeric input and output variables. We already implemented a prototype in SPECPRO that is able to automatically convert from numeric to Boolean constraints, and viceversa. However, we discovered that there is a scalability issue during the Büchi Automaton generation, limiting its application to small specifications. The process could be improved by splitting the requirements in subsets and dealing with multiple automata, but this direction has not been extensively explored yet.

Bibliography

- [AABdB⁺19] Zaid Al-Ars, Twan Basten, Ad de Beer, Marc Geilen, Dip Goswami, Pekka Jääskeläinen, Jiří Kadlec, Marcos Martinez de Alejandro, Francesca Palumbo, Geran Peeren, et al. The fitoptivis ecsel project: highly efficient distributed embedded image/video processing in cyber-physical systems. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 333–338, 2019.
- [ABLN06] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.
- [AGL⁺15] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.
- [AGR13] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Online testing of ltl properties for java code. In *Haifa Verification Conference*, pages 95–111. Springer, 2013.
- [AGTW11] Ahmed Awad, Rajeev Goré, James Thomson, and Matthias Weidlich. An iterative approach for business process template synthesis from compliance rules. In *International Conference*

- on Advanced Information Systems Engineering*, pages 406–421. Springer, 2011.
- [AHDR18] Adina Aniculaesei, Falk Howar, Peer Denecke, and Andreas Rausch. Automated generation of requirements-based test cases for an adaptive cruise control system. In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 11–15. IEEE, 2018.
- [BBB⁺16] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing*, 28(1):45–63, 2016.
- [BBCB16] Jaroslav Bendík, Nikola Benes, Ivana Cerná, and Jiri Barnat. Tunable online mus/mss enumeration. *arXiv preprint arXiv:1606.03289*, 2016.
- [BBD19] Benoît Barbot, Nicolas Basset, and Thao Dang. Generation of signals under temporal constraints for CPS testing. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2019.
- [BBNR18] Ezio Bartocci, Roderick Bloem, Dejan Nickovic, and Franz Roeck. A counting semantics for monitoring ltl specifications over finite traces. In *International Conference on Computer Aided Verification*, pages 547–564. Springer, 2018.
- [BDTW93] René R Bakker, F Dikker, Frank Tempelman, and Petronella Maria Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI*, volume 93, pages 276–281, 1993.
- [Ben17] Jaroslav Bendík. Consistency checking in requirements analysis. In *Proceedings of the 26th ACM SIGSOFT International Sym-*

- posium on Software Testing and Analysis*, pages 408–411. ACM, 2017.
- [BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
- [BGMR16] Matteo Bertello, Nicola Gigante, Angelo Montanari, and Mark Reynolds. Leviathan: A new ltl satisfiability checking tool based on a one-pass tree-shaped tableau. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI’16*, pages 950–956. AAAI Press, 2016. URL: <http://dl.acm.org/citation.cfm?id=3060621.3060753>.
- [BGPS12] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 968–976. IEEE, 2012.
- [BGST12] Daniel Berry, Ricardo Gacitua, Pete Sawyer, and Sri Fatimah Tjong. The case for dumb requirements engineering tools. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 211–217. Springer, 2012.
- [BJK⁺05] Manfred Broy, Bengt Jonsson, J-P Katoen, Martin Leucker, and Alexander Pretschner. Model-based testing of reactive systems. In *Volume 3472 of Springer LNCS*. Springer, 2005.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.
- [BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer, 2006.

- [BLS10] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing ltl semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
- [BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.
- [BMS11] Anton Belov and Joao Marques-Silva. Accelerating mus extraction with recursive model rotation. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 37–40. IEEE, 2011.
- [BMS12] Anton Belov and Joao Marques-Silva. Muser2: An efficient mus extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:123–128, 2012.
- [Bur06] Ilene Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.
- [CC00] Hubert Comon and Véronique Cortier. Flatness is not a weakness. In *International Workshop on Computer Science Logic*, pages 262–276, 2000.
- [CCG⁺02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *14th International Conference on Computer Aided Verification (CAV 2002)*, pages 359–364, 2002.
- [CD91] John W Chinneck and Erik W Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.
- [CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

- [CGM⁺20] Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta. Smt-based satisfiability of first-order ltl with event freezing functions and metric operators. *Information and Computation*, 272:104502, 2020.
- [CRST07] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Stefano Tonetta. Boolean abstraction for temporal logic satisfiability. In *International Conference on Computer Aided Verification*, pages 532–546. Springer, 2007.
- [CRST08] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. Diagnostic information for realizability. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 52–67. Springer, 2008.
- [CRST11] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Formalizing requirements with object models and temporal constraints. *Software & Systems Modeling*, 10(2):147–160, 2011.
- [CRT09] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Requirements validation for hybrid systems. In *International Conference on Computer Aided Verification*, pages 188–203. Springer, 2009.
- [CRT15] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Hreltl: A temporal logic for hybrid systems. *Information and Computation*, 245:54–71, 2015.
- [DAC99] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International conference on Software engineering*, pages 411–420, 1999.
- [DD07] Stéphane Demri and Deepak D’Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

- [DGHP09] Christian Desrosiers, Philippe Galinier, Alain Hertz, and Sandrine Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *Journal of combinatorial optimization*, 18(2):124–150, 2009.
- [DHF15] A Dokhanchi, B Hoxha, and GE Fainekos. Metric interval temporal logic specification elicitation and debugging. In *13th ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 21–23, 2015.
- [DHF16] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Formal requirement debugging for testing and verification of cyber-physical systems. *arXiv preprint arXiv:1607.02549*, 2016.
- [DKM⁺94] Laura K Dillon, George Kutty, Louise E Moser, P Michael Melliar-Smith, and Y Srinivas Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994.
- [DLLF⁺16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω -automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA '16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, 2016. https://doi.org/10.1007/978-3-319-46520-3_8.
- [Dra89] Erik Willy Dravnieks. Identifying minimal sets of inconsistent constraints in linear programs: Deletion, squeeze and sensitivity filtering. *M.Sc. Thesis, Department of Systems and Computer Engineering, Carleton University*, 1989. <https://doi.org/10.22215/etd/1989-01696>.
- [EFH⁺03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with

- temporal logic on truncated paths. In *International conference on computer aided verification*, pages 27–39. Springer, 2003.
- [EKN⁺12] Marie-Aude Esteve, Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein. Formal correctness, safety, dependability, and performance analysis of a satellite. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1022–1031. IEEE, 2012.
- [FB16] Yishai A Feldman and Henry Broodney. A cognitive journey for requirements engineering. In *INCOSE International Symposium*, volume 26, pages 430–444. Wiley Online Library, 2016.
- [FKK08] Norbert E Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto controlled english for knowledge representation. In *Reasoning Web*, pages 104–124. Springer, 2008.
- [FKSFV08] Dana Fisman, Orna Kupferman, Sarai Sheinvald-Faragy, and Moshe Y Vardi. A framework for inherent vacuity. In *Haifa Verification Conference*, pages 7–22. Springer, 2008.
- [FLM⁺04] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Pistore, Marco Roveri, and Paolo Traverso. Specifying and analyzing early requirements in tropos. *Requirements Engineering*, 9(2):132–150, 2004.
- [FWA09] Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.
- [GEL⁺16] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. Arsenal: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pages 41–46. Springer, 2016.
- [GHST13] Rajeev Goré, Jinbo Huang, Timothy Sergeant, and Jimmy Thomson. Finding minimal unsatisfiable subsets in linear temporal logic using bdds, 2013.

- [HK03] Ullrich Hustadt and Boris Konev. TRP++ 2.0: A temporal resolution prover. In *19th International Conference on Automated Deduction*, pages 274–278, 2003.
- [HMU01] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.
- [HTI97] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.
- [JBS15] Michael Jones, John Bradley, and Nat Sakimura. Json web token (jwt). Technical report, 2015. <https://tools.ietf.org/html/rfc7519>.
- [JJ05] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.
- [JKS16] Swen Jacobs, Felix Klein, and Sebastian Schirmer. A high-level ltl synthesis format: Tlsf v1. 1. *arXiv preprint arXiv:1604.02284*, 2016.
- [Jun01] Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI’01 Workshop on Modelling and Solving problems with constraints*, 2001.
- [KC05] Sascha Konrad and Betty HC Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381, 2005.
- [KGHS98] Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt. Autolink—a tool for automatic test generation from sdl specifications. In *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 114–125. IEEE, 1998.
- [KMMP93] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A decision algorithm for full propositional temporal logic. In *Inter-*

- national Conference on Computer Aided Verification*, pages 97–109. Springer, 1993.
- [KT04] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *International SPIN Workshop on Model Checking of Software*, pages 109–126. Springer, 2004.
- [Lap17] Phillip A Laplante. *Requirements engineering for software and systems*. CRC Press, 2017.
- [LCB⁺09] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N Slyngstad, and Maurizio Morisio. Development with off-the-shelf components: 10 facts. *IEEE software*, 26(2):80–87, 2009.
- [LM13] Mark H Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple muses quickly. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 160–175. Springer, 2013.
- [LMG11] Markus Lumpe, Indika Meedeniya, and Lars 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*, pages 468–471, 2011.
- [LPZ⁺13] Jianwen Li, Geguang Pu, Lijun Zhang, Yinbo Yao, Moshe Y Vardi, et al. Palsat: A portfolio LTL satisfiability solver. *arXiv preprint arXiv:1311.1602*, 2013.
- [LS08] Mark H Liffiton and Karem A Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.
- [LVR19] Jianwen Li, Moshe Y Vardi, and Kristin Y Rozier. Satisfiability checking for mission-time ltl. In *International Conference on Computer Aided Verification*, pages 3–22. Springer, 2019.

- [LYP⁺14] Jianwen Li, Yinbo Yao, Geguang Pu, Lijun Zhang, and Jifeng He. Aalta: an LTL satisfiability checker over infinite/finite traces. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 731–734, 2014.
- [LZP⁺13] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y Vardi, and Jifeng He. LTL satisfiability checking revisited. In *20th International Symposium on Temporal Representation and Reasoning*, pages 91–98, 2013.
- [LZPV15] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y Vardi. Sat-based explicit ltl reasoning. In *11th Haifa Verification Conference*, pages 209–224, 2015.
- [ML18] Salomon Sickert Michael Luttenberger, Philipp J. Meyer. Strix, 2018. [Online; accessed 27-June-2019]. URL: `\url{https://strix.model.in.tum.de/}`.
- [MP12] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.
- [MPM⁺17] Michael Masin, Francesca Palumbo, Hans Myrhaug, JA de Oliveira Filho, M Pastena, Maxime Pelcat, Luigi Raffo, Francesco Regazzoni, AA Sanchez, Antonella Toffetti, et al. Cross-layer design of reconfigurable cyber-physical systems. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 740–745. European Design and Automation Association, 2017.
- [MSL11] Joao Marques-Silva and Ines Lynce. On improving mus extraction algorithms. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 159–173. Springer, 2011.
- [Nad10] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 221–229. FMCAD Inc, 2010.

- [NPTV18] Massimo Narizzano, Luca Pulina, Armando Tacchella, and Simone 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*, volume 10811, pages 383–398. Springer, Springer Verlag, 2018. https://doi.org/10.1007/978-3-319-77935-5_26.
- [NPTV19] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto. Property specification patterns at work: verification and inconsistency explanation. *Innovations in Systems and Software Engineering*, 15(3-4):307–323, 2019. <https://doi.org/10.1007/s11334-019-00339-1>.
- [NPTV20] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto. Automated requirements-based testing of black-box reactive systems. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings*, volume 12229, pages 153–169. Springer Nature, 2020. https://doi.org/10.1007/978-3-030-55754-6_9.
- [PFS⁺19] Francesca Palumbo, Tiziana Fanni, Carlo Sau, Luca Pulina, Luigi Raffo, Michael Masin, Evgeny Shindin, Pablo Sanchez de Rojas, Karol Desnos, Maxime Pelcat, et al. Cerbero: Cross-layer model-based framework for multi-objective design of reconfigurable systems in uncertain hybrid environments: Invited paper: Cerbero teams from uniss, unica, ibm research, tase, insa-rennes, upm, usi, abinsula, ambiesense, tno, s&t, crf. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 320–325. ACM, 2019.
- [PH12] Amalinda Post and Jochen Hoenicke. Formalization and analysis of real-time requirements: A feasibility study at BOSCH. *Verified Software: Theories, Tools, Experiments*, pages 225–240, 2012.
- [PM92] Amir Pnueli and Zohar Manna. The temporal logic of reactive and concurrent systems. *Springer*, 16:12, 1992.

- [PMHP12] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch. *Requirements Engineering*, 17(1):19–33, 2012.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.
- [PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.
- [PPVon] L. Pandolfo, L. Pulina, and S. Vuotto. Smt-based consistency checking of configuration-based components specifications. Under submission.
- [PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.
- [RV07] Kristin Y Rozier and Moshe Y Vardi. LTL satisfiability checking. In *Spin*, volume 4595, pages 149–167. Springer, 2007.
- [RV10] Kristin Y Rozier and Moshe Y Vardi. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):123–137, 2010.
- [RV11] Kristin Y Rozier and Moshe Y Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *International Symposium on Formal Methods*, pages 417–431. Springer, 2011.
- [Sch98] Stefan Schwendimann. A new one-pass tableau calculus for pltl. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 277–291. Springer, 1998.

- [Sch12] Viktor Schuppan. Towards a notion of unsatisfiable and unrealizable cores for ltl. *Science of Computer Programming*, 77(7-8):908–939, 2012.
- [Sch16a] Viktor Schuppan. Enhancing unsatisfiable cores for ltl with information on temporal relevance. *Theoretical Computer Science*, 655:155–192, 2016.
- [Sch16b] Viktor Schuppan. Extracting unsatisfiable cores for ltl via temporal resolution. *Acta Informatica*, 53(3):247–299, 2016.
- [SEG00] Michael Schmitt, Michael Ebner, and Jens Grabowski. Test generation with autolink and testcomposer. In *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC-SAM*, volume 2000, 2000.
- [TSL04] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In *Conference on Information Reuse and Integration*, pages 483–498, 2004.
- [UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
- [VNPT19a] S. Vuotto, M. Narizzano, L. Pulina, and A. Tacchella. Automata based test generation with specpro. In *2019 IEEE/ACM 6th International Workshop on Requirements Engineering and Testing (RET)*, pages 13–16. IEEE, 2019. <https://doi.org/10.1109/RET.2019.00010>.
- [VNPT19b] S. Vuotto, M. Narizzano, L. Pulina, and A. Tacchella. Poster: Automatic consistency checking of requirements with reqv. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 363–366. IEEE, 2019. <https://doi.org/10.1109/ICST.2019.00043>.
- [Vuo18] S. Vuotto. Requirements-driven design of cyber-physical systems. In *Proceedings of the Cyber-Physical Systems PhD & Postdoc*

- Workshop 2018*, volume 2208 of *CEUR Workshop Proceedings*, pages 38–44. CEUR-WS, 2018. <http://ceur-ws.org/Vol-2208/6.pdf>.
- [Vuo19] S. Vuotto. Automata-based generation of test cases for reactive systems. In *Proceedings of the Cyber-Physical Systems PhD & Postdoc Workshop 2019*, volume 2457 of *CEUR Workshop Proceedings*, pages 96–106. CEUR-WS, 2019. <http://ceur-ws.org/Vol-2457/10.pdf>.
- [WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.
- [Wol85] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, pages 119–136, 1985.
- [Yan91] S Yang. Logic synthesis and optimization bench marks user guide. technical report-microelectronic center of north carolina. 1991.
- [ZT15] Bolong Zeng and Li Tan. Test reactive systems with buchi automata: acceptance condition coverage criteria and performance evaluation. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 380–387. IEEE, 2015.
- [ZT16] Bolong Zeng and Li Tan. Test reactive systems with büchi-automaton-based temporal requirements. In *Theoretical Information Reuse and Integration*, pages 31–57. Springer, 2016.