Property Specification Patterns at Work: Verification and Inconsistency Explanation

Massimo Narizzano[1], Luca Pulina[2], Armando Tacchella[1], and Simone Vuotto[1,2]

[1] DIBRIS, University of Genoa, Viale Causa 13, 16145 Genoa, Italy
massimo.narizzano@unige.it, armando.tacchella@unige.it
[2] Chemistry and Pharmacy Dept., University of Sassari, Via Vienna 2, Sassari, Italy
lpulina@uniss.it, svuotto@uniss.it

Abstract. Property Specification Patterns (PSPs) have been proposed to ease the formalization of requirements, yet enable automated verification thereof. In particular, the internal consistency of specifications written with PSPs can be checked automatically with the use of, e.g., Linear Temporal Logic (LTL) satisfiability solvers. However, for most practical applications, the expressiveness of PSPs is too restricted to enable writing useful requirement specifications, and proving that a set of requirements is inconsistent can be worthless unless a minimal set of conflicting requirements is extracted to help designers to correct a wrong specification. In this paper, we extend PSPs by considering Boolean as well as atomic numerical assertions, we contribute an encoding from extended PSPs to LTL formulas, and we present an algorithm computing inconsistency explanations, i.e., irreducible inconsistent subsets of the original set of requirements. Our extension enables us to reason about the internal consistency of functional requirements which would not be captured by basic PSPs. Experimental results demonstrate that our approach can check and explain (in)consistencies in specifications with nearly two thousand requirements generated using a probabilistic model, and that it enables effective handling of real-world case studies as well.

1 Introduction

In the context of safety- and security-critical Cyber-Physical Systems (CPSs), checking the sanity of functional requirements is an important, yet challenging task. 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) [20] 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) [41], Computational Tree Logic (CTL) [12], or Graphical Interval Logic (GIL) [16]. 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., [27, 30, 48, 11, 23].

Sanity checking of requirements may consist of three parts: redundancy (vacuity) checking, completeness checking and consistency checking [3]. A specification is satisfied vacuously in a model if it is satisfied in some non-interesting way; borrowing the example from [45], 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 [21, 46]. Completeness checking is equivalent to verify if the set of requirements covers all reasonable behaviours of a system. Completeness can be checked in combination with a system model, but in [3] 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 [46] are possible: (*i*) *realizability*, i.e., testing whether there is an *open* system that satisfies all the properties in the set [42], 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 property is either valid, or unsatisfiable this must be due to an error. Even if the satisfiability test is weaker than the realizability test, its importance is widely recognized [46]. In this paper, 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 — see, e.g., [49, 25, 48, 35] for tableau-based methods and [44–46, 29, 28] for methods based on automata-theoretic approaches.

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 [37][3] 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 numer-

[3] Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments — `http://www.cerbero-h2020.eu/`

ical 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 [14]: in our example above, the encoding would be simply $\square\,(\theta_1 < 170)$. Unfortunately, this approach does not always lend itself to a practical solution, because Constraint Linear Temporal Logic is undecidable in general [13]. Restrictions on $\mathcal{D}$ may restore decidability [14], but they introduce limitations in the expressiveness of the corresponding PSPs. In this paper, 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 requirements, obtaining a *minimal* set of such requirements would be desirable to help designers avoid manual checks to pinpoint problems in a specification. 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., [6, 32] — or domain independent — see, e.g., [24]. They can be further divided into algorithms that find only one inconsistent subset or all inconsistent subsets. To the best of our knowledge, there is no special-purpouse algorithm implemented yet. Indeed, the ones presented in [4, 3, 8], use a variant of the general-purpose algorithm for computing all unsatisfiable cores, also known as the "deletion algorithm". 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 PSP($\mathcal{D}_C$) into a corresponding LTL formula.
- We present a tool[4] based on state-of-the-art decision procedures and model checkers to automatically analyze requirements expressed as PSPs($\mathcal{D}_C$).
- We propose algorithms devoted to extract minimal subsets of inconsistent requirements, and we implement them in the tool mentioned above.

[4] `https://gitlab.sagelab.it/sage/SpecPro`

– We implement a generator of artificial requirements expressed as PSPs($\mathcal{D}_C$); the generator takes a set of parameters in input and emits a collection of PSPs according to a parametrized 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 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 PSP($\mathcal{D}_C$) are carried out using tools and techniques available in the literature [45, 46, 27]. 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. This paper is based on and extends the one presented at the NASA Formal Method Conference [39]. The additional material relates to (*i*) algorithms for inconsistency explanation, including their experimental evaluation, (*ii*) proofs of results that were only stated in [39], and (*iii*) the experimental analysis of the tableau-based tool LEVIATHAN [9].

The rest of the paper is organized as follows. Section 2 contains some basic concepts on LTL, PSPs and some related work. In Section 3 we present the extension of basic PSPs over $\mathcal{D}_C$ and the related encoding to LTL, while in Section 4 we present inconsistency explanation algorithms. In Sections 5 and 6 we report the results of the experimental analysis concerning the scalability and the case study on the embedded controller, respectively. We conclude the paper in Section 7 with some final remarks.

2 Background and Related Work

LTL syntax and semantics. Linear temporal logic (LTL) [40] formulae 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 formulae, $\mathcal{X}$ 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. An LTL formula is interpreted over a *computation*, i.e., a function $\pi : \mathbb{N} \to 2^{Prop}$ which assigns truth values to the elements of $Prop$ at each time instant (natural number). For a computation π and a time instant $i \in \mathbb{N}$:

– $\pi, i \models p$ for $p \in Prop$ iff $p \in \pi(i)$
– $\pi, i \models \neg\alpha$ iff $\pi, i \not\models \alpha$
– $\pi, i \models (\alpha \vee \beta)$ iff $\pi, i \models \alpha$ or $\pi, i \models \beta$

- $\pi, i \models \mathcal{X}\,\alpha$ iff $\pi, i+1 \models \alpha$
- $\pi, i \models \alpha\,\mathcal{U}\,\beta$ 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 *true* and we write $\models \phi$.

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 $\bot$. We introduce $\Diamond\,\phi$ ("eventually") to denote $\top\,\mathcal{U}\,\phi$ and $\Box\,\phi$ ("always") to denote $\neg\Diamond\,\neg\phi$. Finally, some of the PSPs use the "weak until" operator defined as $\alpha\,\mathcal{W}\,\beta = \Box\,\alpha \vee (\alpha\,\mathcal{U}\,\beta)$.

LTL satisfiability. Among various approaches to decide LTL satisfiability, reduction to model checking was proposed in [44] to check the consistency of requirements expressed as LTL formulae. Given a formula ϕ over a set *Prop* of atomic propositions, a *universal* model M can be constructed. Intuitively, a universal model encodes all the possible computations over *Prop* as (infinite) traces, and therefore ϕ is satisfiable precisely when M does not satisfy $\neg\phi$. In [46] a first improvement over this basic strategy is presented together with the tool PANDA[5] whereas in [29] 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 [28] and [27]. In the latter, a portfolio LTL satisfiability solver called POLSAT is proposed to run different techniques in parallel and return the result of the first one to terminate successfully.

Property Specification Patterns (PSPs). The original proposal of PSPs is to be found in [20]. They are meant to describe the structure of systems' behaviours and provide expressions of such behaviors in a range of common formalisms. An example of a PSP is given in Figure 1 — with some parts omitted for sake of readability.[6] A pattern is comprised of a *Name* (Response in Figure 1), an (informal) statement describing the behaviour captured by the pattern, and a (structured English) statement [26] that should be used to express requirements. 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. 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

[5] `https://ti.arc.nasa.gov/m/profile/kyrozier/PANDA/PANDA.html`

[6] We omitted aspects which are not relevant for our work, e.g., translations to other logics like CTL [20]. The full list of PSPs considered in this paper and their mapping to LTL and other logics is available at `http://patterns.projects.cis.ksu.edu/`.

Response

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

Structured English Grammar

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

LTL Mappings

Globally	$\square\,(P \rightarrow \Diamond\, S)$
Before R	$\Diamond\, R \rightarrow (P \rightarrow (\overline{R}\ \mathcal{U}\ (S \wedge \overline{R})))\ \mathcal{U}\ R$
After Q	$\square\,(Q \rightarrow \square\,(P \rightarrow \Diamond\, S))$
Between Q and R	$\square\,((Q \wedge \overline{R} \wedge \Diamond\, R) \rightarrow (P \rightarrow (\overline{R}\ \mathcal{U}\ (S \wedge \overline{R})))\ \mathcal{U}\ R)$
After Q until R	$\square\,(Q \wedge \overline{R} \rightarrow ((P \rightarrow (\overline{R}\ \mathcal{U}\ (S \wedge \overline{R})))\ \mathcal{W}\ R)$

Example

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

Fig. 1. Response Pattern ($\overline{\alpha}$ stands for $\neg\alpha$).

end. The *body* of a pattern, describes the behavior that we want to specify. In [20], 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 [43], 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.

Inconsistency Explanation. 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*) [7], *minimal unsatisfiable subset* [6] (*MUS*), *minimal unsatisfiable core* [32] (*MUC*), and also High-Level MUC (HLMUC) [38] 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 [5], 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 [19]. 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 [19, 10, 2, 15]. 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. The majority of the algorithms presented in literature are domain specific [31, 36, 32, 6] and, to the best of our knowledge, no specific approach that works for LTL has been proposed so far. 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 [31], 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.

Related Work. In [33] the framework *Property Specification Pattern Wizard (PSP-Wizard)* is presented. Its purpose is the machine-assisted definition of temporal formulae 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 [26], an extension is presented to deal with real-time specifications, together with mappings to Metric temporal logic (MTL), Timed computational tree logic (TCTL) and Real-time graphical interval logic (RTGIL). 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 [26] also provided inspiration to a recent set of works [18, 17] 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 formulae 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.

3 Constraint Property Specification Patterns

Let us start by defining a *constraint system* $\mathcal{D}$ as a tuple $\mathcal{D} = (D, R_1, \ldots, R_n, \mathcal{I})$, 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 $R_i(t_1, \ldots, t_{a_i})$ for some $1 \leq i \leq n$ and $t_j \in T$ for all $1 \leq j \leq a_i$ — we also use the term *constraint* when $\mathcal{D}$ is understood from the context. We define *linear temporal logic modulo constraints* — LTL($\mathcal{D}$) for short — as an extension of LTL with additional atomic constraints. Given a set of Boolean propositions $Prop$, a constraint system $\mathcal{D} = (D, R_1, \ldots, R_n, \mathcal{I})$, and a set of terms $T = C \cup X$, an LTL($\mathcal{D}$) formula is defined as:

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

where $p \in Prop$, ϕ, ϕ_1, ϕ_2 are 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 LTL($\mathcal{D}$) formulas is a (strict) subset of those in constraint linear temporal logic — CLTL($\mathcal{D}$) for short — as defined, e.g., in [14]. LTL($\mathcal{D}$) formulas are interpreted over computations of the form $\pi : \mathbb{N} \to 2^{Prop}$ plus additional *evaluations* of the form $\nu : T \times \mathbb{N} \to D$ 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 LTL($\mathcal{D}$) by handling constraints:

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

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. While CLTL($\mathcal{D}$) is undecidable in general [14, 13], 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 [20] 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($\mathcal{D}_C$)[7] can be rewritten as the following LTL($\mathcal{D}_C$) formula:

$$\begin{array}{l} \Box(v < 5.0 \vee v = 5.0) \quad \wedge \\ \Box(a \rightarrow \Diamond(v < 8.5) \vee (v = 8.5)) \wedge \\ \Box(a \rightarrow \Box(\neg(v < 3.2) \rightarrow \Diamond z)) \end{array} \tag{1}$$

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 $X(\phi)$ 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 $x \in X(\phi)$ is compared to; more precisely, for every variable $x \in X(\phi)$ we construct a set $S_x(\phi) = \{c_1, .., c_n\}$ 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 (1), we have $X = \{v\}$ 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 $x \in X(\phi)$, let $S_x(\phi) = \{c_1, \ldots, c_n\}$ be the set of thresholds for which we define the corresponding sets of *inequality propositions* $Q_x(\phi) = \{q_1, \ldots, q_n\}$ and *equality propositions* $E_x(\phi) = \{e_1, \ldots, e_n\}$. Informally, inequality propositions should be true exactly when a variable $x \in X(\phi)$ 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;

[7] Strictly 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 $(v < 5.0) \vee (v = 5.0)$.

– $e_j \in \pi(i)$ 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.

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

$$x < c_k \rightsquigarrow \bigvee_{j=1}^{k} q_j \vee \bigvee_{j=1}^{k-1} e_j \qquad \text{and} \qquad x = c_k \rightsquigarrow e_k. \tag{2}$$

Replacing atomic numerical constraints is not enough to ensure equisatisfiability of ϕ' with respect to ϕ. In particular, for every $x \in X(\phi)$, 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:

$$\phi_M = \bigwedge_{x \in X(\phi)} \left(\bigwedge_{a,b \in M_x(\phi), a \neq b} \Box\neg(a \wedge b) \right) \tag{3}$$

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

For instance, given example (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:

$$\begin{aligned} \phi_M = &\Box\neg(q_1 \wedge q_2) \wedge \Box\neg(q_1 \wedge q_3) \wedge \Box\neg(q_1 \wedge e_1) \wedge \Box\neg(q_1 \wedge e_2) \wedge \\ &\Box\neg(q_1 \wedge e_3) \wedge \Box\neg(q_2 \wedge q_3) \wedge \Box\neg(q_2 \wedge e_1) \wedge \Box\neg(q_2 \wedge e_2) \wedge \\ &\Box\neg(q_2 \wedge e_3) \wedge \Box\neg(q_3 \wedge e_1) \wedge \Box\neg(q_3 \wedge e_2) \wedge \Box\neg(q_3 \wedge e_3) \wedge \\ &\Box\neg(e_1 \wedge e_2) \wedge \Box\neg(e_1 \wedge e_3) \wedge \Box\neg(e_2 \wedge e_3). \end{aligned} \tag{4}$$

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

$$\begin{aligned} \phi_M \wedge (\ &\Box(q_1 \vee q_2 \vee e_1 \vee e_2) \wedge \\ &\Box(a \rightarrow \Diamond(\textstyle\bigvee_{i=1}^{3} q_i \vee e_i)) \wedge \\ &\Box(a \rightarrow \Box(\neg q_1 \rightarrow \Diamond z))). \end{aligned} \tag{5}$$

We can now state the following:

Theorem 1. *Let ϕ be an LTL($\mathcal{D}_C$) formula on the set of proposition Prop and terms $T = X(\phi) \cup C(\phi)$; for every $x \in X(\phi)$, 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 $Prop \cup$*

$\bigcup_{x \in X(\phi)} Q_x(\phi) \cup E_x(\phi)$ *obtained from* ϕ *by applying substitutions (2) for every* $x \in X(\phi)$ *and* $c_k \in S_x(\phi)$*, and let* ϕ_M *be the LTL formula obtained as in (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 $x \in X(\phi)$, for which the corresponding set of thresholds is $S_x(\phi) = \{c_1, \ldots, c_n\}$. Considering that thresholds are ordered in ascending order, we construct the following sets of time instants:

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

which, given the standard semantics of "$<$" and "$=$", are a partition of $\mathbb{N}$. Let $\mathcal{N}_x$ denote such partition for a specific variable $x \in X(\phi)$. We construct a computation π' such that, for all time instants $i \in \mathbb{N}$ and propositions $p \in \mathit{Prop}$, we have $p \in \pi'(i)$ exactly when $p \in \pi(i)$ and, for each variable $x \in X(\phi)$, given $Q_x(\phi) = \{q_1, \ldots q_n\}$ and $E_x(\phi) = \{e_1, \ldots e_n\}$, 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 $i \in N_{c_1<x<c_2}$;
- $\ldots$
- $q_n \in \pi'(i)$ 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 $x \in X(\phi)$, 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 and the corresponding translation ϕ'. Let ψ and ψ' be two subformulas of ϕ and ϕ', respectively. For every $i \in \mathbb{N}$:

- if $\psi \equiv p$ for $p \in \mathit{Prop}$ 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 $x \in X(\phi)$ and some constant $c_k \in S_x(\phi)$ then, according to (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

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

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 $x \in X(\phi)$ and some constant $c_k \in S_x(\phi)$ then, according to (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 $\pi, i, \nu \not\models_{\mathcal{D}_C} \neg\alpha$ 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 $i \in mathbbN$, $\pi, i, \nu \models_{\mathcal{D}_C} \mathcal{X}\,\alpha$ 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 LTL($\mathcal{D}_C$). First we observe that, for a generic variable $x \in X(\phi)$, 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 $x \in X(\phi)$ 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 $p \in Prop$, 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 $x \in X(\phi)$, 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.

□

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 proposition Prop and terms $T = X(\phi) \cup C(\phi)$; for every $x \in X(\phi)$, 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 $Prop \cup \bigcup_{x \in X(\phi)} Q_x(\phi) \cup E_x(\phi)$ obtained from ϕ by applying substitutions (2) for every $x \in X(\phi)$ and $c_k \in S_x(\phi)$, and ϕ_M be the LTL formula obtained as in (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 (3), for each variable $x \in X(\phi)$, 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

$$\binom{2n}{2} = \frac{2n!}{2!(2n-2)!} = \frac{2n(2n-1)}{2} = n(2n-1) \tag{6}$$

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 1. Set R of inconsistent PSPs.

If we consider $m = \max_{x \in X(\phi)} |S_x(\phi)|$, it follows that $|\phi_M| = O(|X(\phi)| \cdot m^2)$. Now it remains to show the effect of substitution (2) in ϕ'. For every variable $x \in X(\phi)$, if we let again $n = |S_x(\phi)|$, then for each constant $c_k \in S_x(\phi)$ we have:

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

Therefore, we can say that each condition corresponding to x is translated to a formula of size $O(n)$. Now, let $m = \max_{x \in X(\phi)} |S_x(\phi)|$, and let p be the maximum number of occurrences in ϕ of a condition $x = c$ or $x < c$ for specific values of $x \in X(\phi)$ and $c \in C(\phi)$; then we have that $|\phi| = O(|X(\phi)| \cdot p \cdot m + r)$, where r is the number of symbols that do not appear in conditions. Since each condition in ϕ is translated to a formula of size $O(m)$ in ϕ', we have that $|\phi'| = O(|X(\phi)| \cdot p \cdot m^2 + r)$. Considering also the bound for $|\phi_M|$ we obtain

$$O(|\phi' + \phi_M|) = O(|X(\phi)| \cdot m^2 \cdot (p + 1) + r) = O(|X(\phi)| \cdot p \cdot m^2 + r).$$

Since $|\phi| = O(|X(\phi)| \cdot p \cdot m)$, and the values of the parameters $|X(\phi)|$, p and r do not depend on the translation, we can conclude that $O(|\phi' + \phi_M|) = O(|\phi|^2)$. □

4 Inconsistency Explanation

Given a set $R = \{r_1, \ldots, r_n\}$ 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 from I makes the set consistent again. Table 1 shows an inconsistent specification as a set $R = \{r_1, \ldots r_7\}$ 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.

4.1 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.

Algorithm 1. Linear Deletion-Based MUC Extraction Algorithm

```
1: function findInconsistency(R)
2:     R' ← R
3:     for r_i ∈ R do
4:         R' ← R' \ {r_i}
5:         if isConsistent(R') then
6:             R' ← R' ∪ {r_i}
7:         end if
8:     end for
9:     return R'
10: end function
```

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 number of calls to the model checker.

Example 1. Considering the set R in Table 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 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.

Algorithm 2. Dichotomic MUC Extraction Algorithm

```
 1: function findInconsistency(R)
 2:     return findInconsistency(R, ∅)
 3: end function

 4: function findInconsistency(R, I)
 5:     if |R| ≤ 1 then
 6:         if isConsistent(I) then
 7:             return I ∪ R
 8:         else
 9:             return I
10:         end if
11:     end if
12:     (R1, R2) ← split(R)
13:     if |R1| > 1 and |R2| > 1 then
14:         if ¬ isConsistent(R1 ∪ I) then
15:             return findInconsistency(R1, I)
16:         end if
17:         if ¬ isConsistent(R2 ∪ I) then
18:             return findInconsistency(R2, I)
19:         end if
20:     end if
21:     I ← findInconsistency(R2, I ∪ R1)
22:     I ← findInconsistency(R1, I \ R1)
23:     return I
24: end function
```

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. Taking

into account Table 1, 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 FINDINCONSISNTENCY 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 5.2, when $|I| \ll |R|$ it is noticeable faster than the linear version.

Example 2. Considering again the set R reported in Table 1, in the following we report step-by-step how Algorithm 2 works. For lack of space in the table we replace ISCONSISTENT with *isCons*.

Step	R	R_1	R_2	I	$isCons(R_1 \cup I)$	$isCons(R_2 \cup I)$
1:	$\{r_1, \ldots, r_7\}$	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6, r_7\}$	$\{\}$	$False$	–
2:	$\{r_1, r_2, r_3\}$	$\{r_1\}$	$\{r_2, r_3\}$	$\{\}$	$True$	$True$
3:	$\{r_2, r_3\}$	$\{r_2\}$	$\{r_3\}$	$\{r_1\}$	–	–
4:	$\{r_2\}$	–	–	$\{r_1\}$	–	–
5:	$\{r_1\}$	–	–	$\{r_2\}$	–	–

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.

5 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 3 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, 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.

5.1 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 3. 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.

Evaluation of LTL satisfiability solvers. The tools considered in our analysis are the ones included in the portfolio solver POLSAT [27], namely AALTA [30], NuSMV [11], PLTL [48], and TRP++ [23]. We also consider LEVIATHAN [9], 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 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 by

dom	2				4				8				16			
$\#vars$	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
NUSMV-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
NUSMV-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 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).

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 NUSMV 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 2.

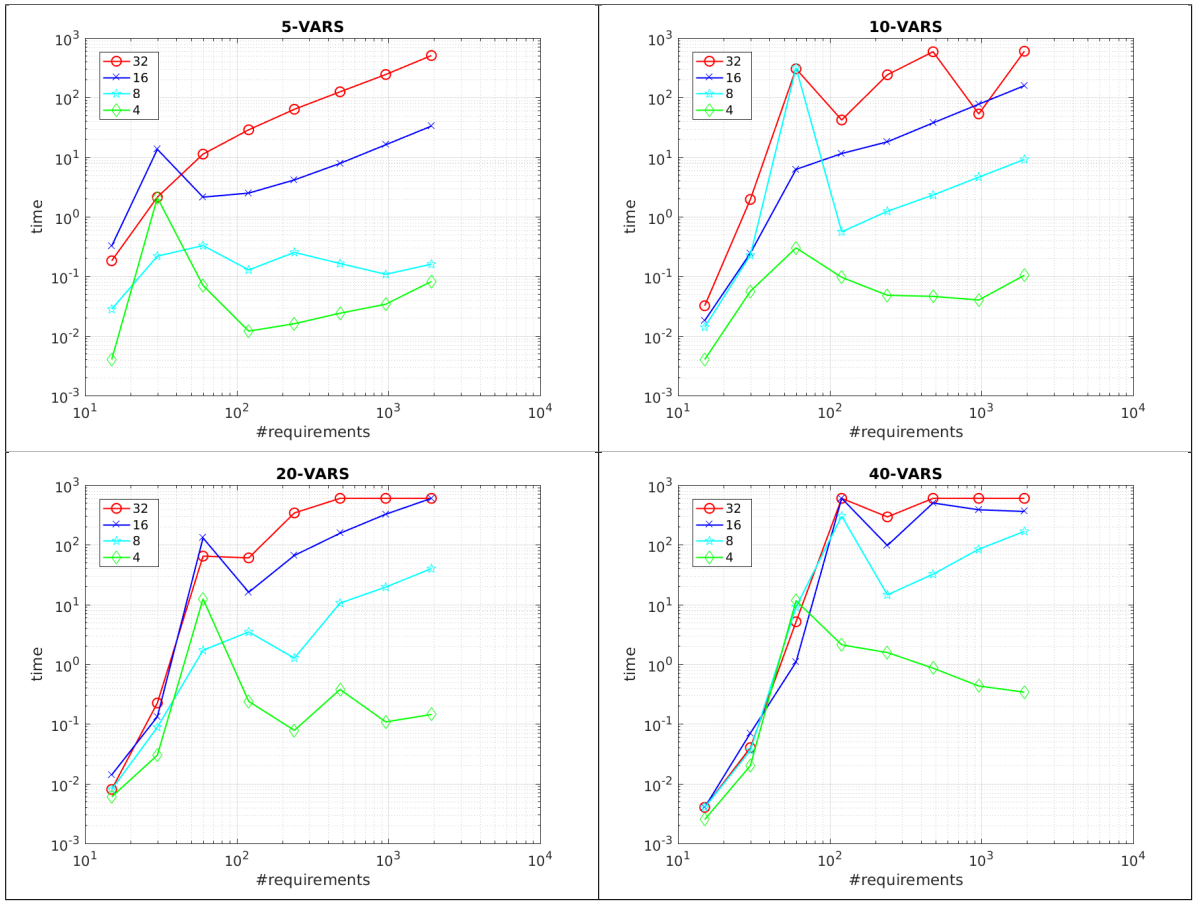

Fig. 2. Scalability Analysis (Part 1). On the x-axes (y-axes resp.) we report $\#req$ (CPU time in seconds resp.). Axis are both in logarithmic scale. In each plot we consider different values of $\#dom$. In particular, the diamond green line is for $\#dom = 4$, the light blue line with stars is for $\#dom = 8$, the blue crossed lines and red circled ones denote $\#dom = 16$ and $\#dom = 32$, respectively.

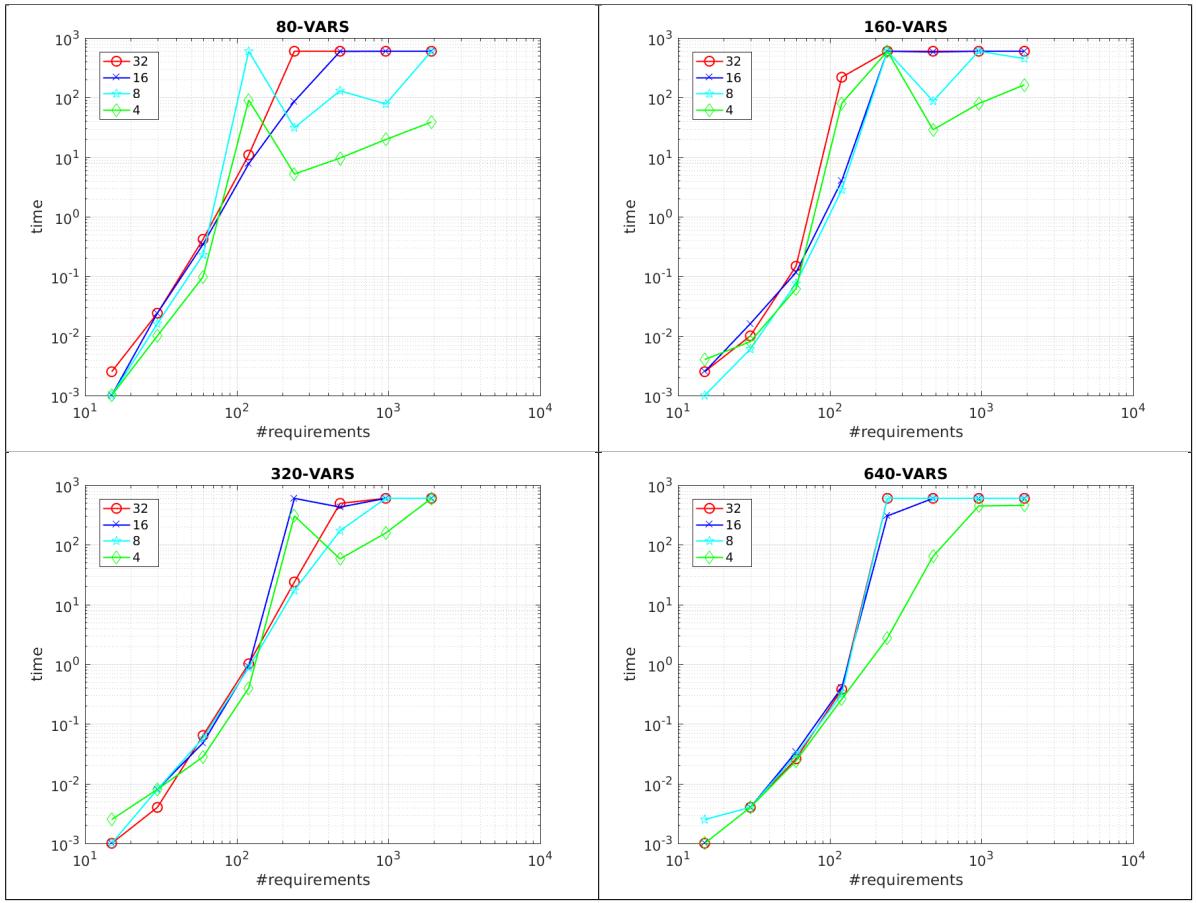

Fig. 3. Scalability Analysis (Part 2). Plots are organized as in Figure 2.

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 2 and Figure 3 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 2 and 3, we can see that the difficulty of the problem increases when all the values of the considered parameters increase, 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 then 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.

5.2 Evaluation of MUC Extraction

In order to evaluate the algorithms proposed in Section 4, we consider the pool of inconsistent benchmarks resulting from the experiment presented in Section 5.1, for a total amount of 559, having different requirements set dimension as reported in Table 5.2. 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. We limit the presentation of the results to the algorithms presented in Section 4 because state-of-the-art tools able to cope with this task, namely PLTL-MUP [22] and TRP++UC [47], report the same correctness and scalability issues of their counterpart presented in Section 5.1; We also considered PROCMINE [1], but we do not report its results for the same motivation.

In Figure 4 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.

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), looking at the plots in the middle of Figure 4 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. More, 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.

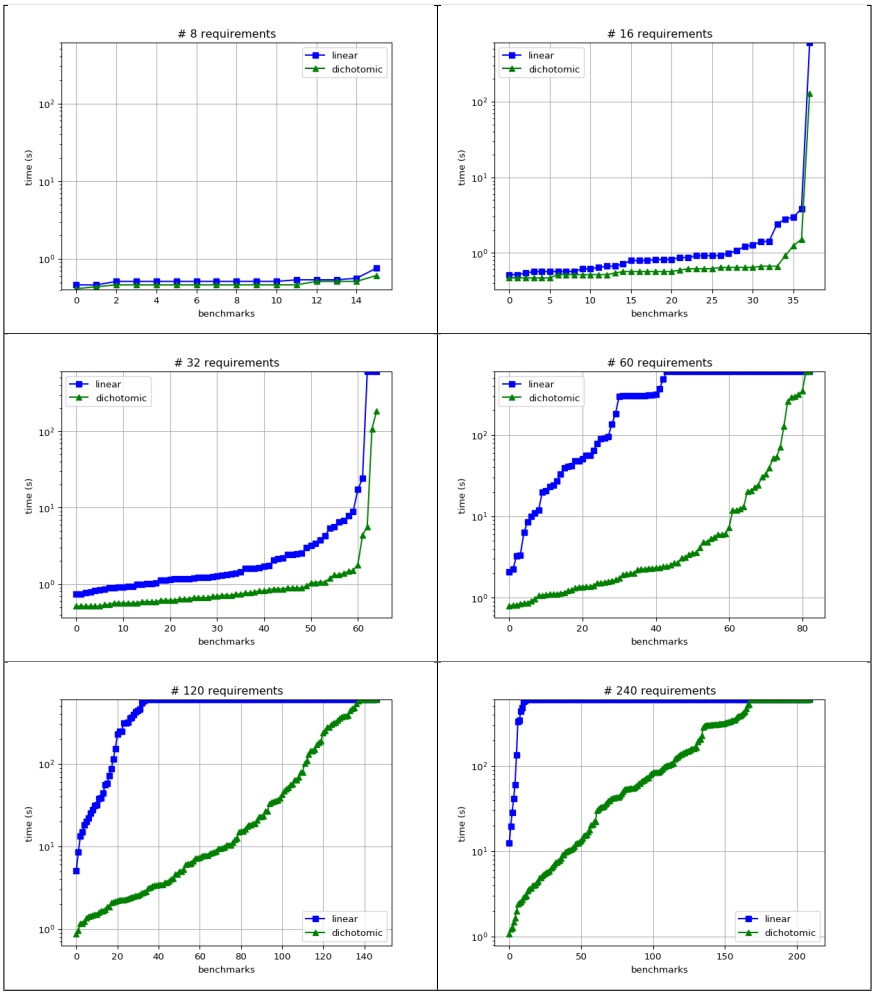

Fig. 4. Performance of the algorithms for MUC extraction. On the x-axes we report the number of benchmarks, and on the y-axes we report the time in logarithmic scale. In each plot we consider different values of #req. The green and blue lines shows median times of the dichotomic and linear algorithms, respectively.

Considering the plots in the bottom of Figure 4, 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.

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

Table 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").

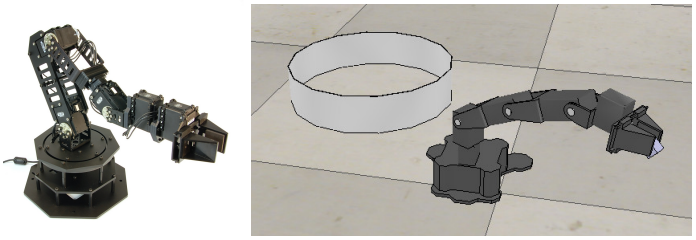

Fig. 5. WidowX robotic arm (left) and the simulated arm moving a grabbed object in the bucket on the left (right).

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.

6 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 arm[8] equipped with a gripper: Figure 5 shows a snapshot of

[8] `http://www.trossenrobotics.com/widowxrobotarm`.

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. 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.

the robot in the intended usage scenario taken from V-REP[9] simulator. The design of the embedded controller is currently part of the activities related to the "Self-Healing System for Planetary Exploration" use case [37] 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 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 we present a synopsis of the requirements, to give an idea of the kind of patterns used in the specification.[10] 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 [20] and [43], it is often the case that few PSPs cover the majority of specifications whereas others are sparsely used.

Our first experiment[11] is to run NuSMV-INVAR on the intended specification translated to LTL($\mathcal{D}_C$). The motivation for presenting the results with 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

[9] `http://www.coppeliarobotics.com/`

[10] The full list of requirements and the fault injection examples are available at `https://github.com/SAGE-Lab/robot-arm-usecase`.

[11] Experiments 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.

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. 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.

7 Conclusions

In this paper, we have extended basic PSPs over the constraint system $\mathcal{D}_C$, and we have provided an encoding from any 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. Inconsistency explanations could be provided for all, but the largest specifications in our benchmark base. These results witness that our approach is viable and worth of adoption in the process of requirement engineering. Our next steps toward this goal will include easing the translation from natural language requirements to patterns, and extending the pattern language to deal with other relevant aspects of cyber-physical systems, such as real-time constraints and related logics (e.g., Signal Temporal Logic [34]). Further elements will also include domain-specific strategies to search for MUCs in requirements aiming at improving the performance of the algorithm presented in Section 4, i.e., discovering or approximating the minimum set of requirements causing the inconsistency while looking for the consistency of the set, instead to do it at the end of the consistency checking, as we did in Section 4.

Acknowledgments The research of Luca Pulina and Simone Vuotto has been funded by the EU Commissions H2020 Programme under grant agreement N.732105 (CERBERO project). The research of Luca Pulina has been also partially funded by the Sardinian Regional Project PROSSIMO (POR FESR 2014/20-ASSE I).

References

1. Awad, A., Goré, R., Thomson, J., Weidlich, M.: An iterative approach for business process template synthesis from compliance rules. In: International Conference on Advanced Information Systems Engineering. pp. 406–421. Springer (2011)
2. Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving over-determined constraint satisfaction problems. In: IJCAI. vol. 93, pp. 276–281 (1993)
3. Barnat, J., Bauch, P., Beneš, N., Brim, L., Beran, J., Kratochvíla, T.: Analysing sanity of requirements for avionics systems. Formal Aspects of Computing 28(1), 45–63 (2016)

4. Barnat, J., Bauch, P., Brim, L.: Checking sanity of software requirements. In: International Conference on Software Engineering and Formal Methods. pp. 48–62. Springer (2012)
5. Belov, A., Marques-Silva, J.: Accelerating mus extraction with recursive model rotation. In: Formal Methods in Computer-Aided Design (FMCAD), 2011. pp. 37–40. IEEE (2011)
6. Belov, A., Marques-Silva, J.: Muser2: An efficient mus extractor. Journal on Satisfiability, Boolean Modeling and Computation 8, 123–128 (2012)
7. Bendík, J.: Consistency checking in requirements analysis. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 408–411. ACM (2017)
8. Bendík, J., Beneš, N., Barnat, J., Černá, I.: Finding boundary elements in ordered sets with application to safety and requirements analysis. In: International Conference on Software Engineering and Formal Methods. pp. 121–136. Springer (2016)
9. Bertello, M., Gigante, N., Montanari, A., Reynolds, M.: 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. pp. 950–956. IJCAI'16, AAAI Press (2016), http://dl.acm.org/citation.cfm?id=3060621.3060753
10. Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. ORSA Journal on Computing 3(2), 157–168 (1991)
11. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: 14th International Conference on Computer Aided Verification (CAV 2002). pp. 359–364 (2002)
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2), 244–263 (1986)
13. Comon, H., Cortier, V.: Flatness is not a weakness. In: International Workshop on Computer Science Logic. pp. 262–276 (2000)
14. Demri, S., DSouza, D.: An automata-theoretic approach to constraint LTL. Information and Computation 205(3), 380–415 (2007)
15. Desrosiers, C., Galinier, P., Hertz, A., Paroz, S.: Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. Journal of combinatorial optimization 18(2), 124–150 (2009)
16. Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M., Ramakrishna, Y.S.: A graphical interval logic for specifying concurrent systems. ACM Transactions on Software Engineering and Methodology (TOSEM) 3(2), 131–165 (1994)
17. Dokhanchi, A., Hoxha, B., Fainekos, G.: Metric interval temporal logic specification elicitation and debugging. In: 13th ACM-IEEE International Conference on Formal Methods and Models for Codesign. pp. 21–23 (2015)
18. Dokhanchi, A., Hoxha, B., Fainekos, G.: Formal requirement debugging for testing and verification of cyber-physical systems. arXiv preprint arXiv:1607.02549 (2016)
19. Dravnieks, E.W.: Identifying minimal sets of inconsistent constraints in linear programs: Deletion, squeeze and sensitivity filtering. (1991)
20. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International conference on Software engineering. pp. 411–420 (1999)
21. Fisman, D., Kupferman, O., Sheinvald-Faragy, S., Vardi, M.Y.: A framework for inherent vacuity. In: Haifa Verification Conference. pp. 7–22. Springer (2008)

22. Goré, R., Huang, J., Sergeant, T., Thomson, J.: Finding minimal unsatisfiable subsets in linear temporal logic using bdds (2013)
23. Hustadt, U., Konev, B.: TRP++ 2.0: A temporal resolution prover. In: 19th International Conference on Automated Deduction. pp. 274–278 (2003)
24. Junker, U.: Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI01 Workshop on Modelling and Solving problems with constraints (2001)
25. Kesten, Y., Manna, Z., McGuire, H., Pnueli, A.: A decision algorithm for full propositional temporal logic. In: International Conference on Computer Aided Verification. pp. 97–109. Springer (1993)
26. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering. pp. 372–381 (2005)
27. Li, J., Pu, G., Zhang, L., Yao, Y., Vardi, M.Y., et al.: Polsat: A portfolio LTL satisfiability solver. arXiv preprint arXiv:1311.1602 (2013)
28. Li, J., Yao, Y., Pu, G., Zhang, L., He, J.: Aalta: an LTL satisfiability checker over infinite/finite traces. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 731–734 (2014)
29. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL satisfiability checking revisited. In: 20th International Symposium on Temporal Representation and Reasoning. pp. 91–98 (2013)
30. Li, J., Zhu, S., Pu, G., Vardi, M.Y.: Sat-based explicit ltl reasoning. In: 11th Haifa Verification Conference. pp. 209–224 (2015)
31. Liffiton, M.H., Malik, A.: Enumerating infeasibility: Finding multiple muses quickly. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 160–175. Springer (2013)
32. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (2008)
33. Lumpe, M., Meedeniya, I., Grunske, L.: 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. pp. 468–471 (2011)
34. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer (2004)
35. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer Science & Business Media (2012)
36. Marques-Silva, J., Lynce, I.: On improving mus extraction algorithms. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 159–173. Springer (2011)
37. Masin, M., Palumbo, F., Myrhaug, H., de Oliveira Filho, J., Pastena, M., Pelcat, M., Raffo, L., Regazzoni, F., Sanchez, A., Toffetti, A., et al.: Cross-layer design of reconfigurable cyber-physical systems. In: 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 740–745. IEEE (2017)
38. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. pp. 221–229. FMCAD Inc (2010)
39. Narizzano, M., Pulina, L., Tacchella, A., Vuotto, S.: Consistency of property specification patterns with boolean and constrained numerical signals. In: NASA Formal Methods: 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings. vol. 10811, pp. 383–398. Springer (2018)

40. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57. IEEE (1977)
41. Pnueli, A., Manna, Z.: The temporal logic of reactive and concurrent systems. Springer 16, 12 (1992)
42. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 179–190. ACM (1989)
43. Post, A., Hoenicke, J.: Formalization and analysis of real-time requirements: A feasibility study at BOSCH. Verified Software: Theories, Tools, Experiments pp. 225–240 (2012)
44. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Spin. vol. 4595, pp. 149–167. Springer (2007)
45. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. International Journal on Software Tools for Technology Transfer (STTT) 12(2), 123–137 (2010)
46. Rozier, K.Y., Vardi, M.Y.: A multi-encoding approach for LTL symbolic satisfiability checking. In: International Symposium on Formal Methods. pp. 417–431. Springer (2011)
47. Schuppan, V.: Extracting unsatisfiable cores for ltl via temporal resolution. Acta Informatica 53(3), 247–299 (2016)
48. Schwendimann, S.: A new one-pass tableau calculus for pltl. In: International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. pp. 277–291. Springer (1998)
49. Wolper, P.: The tableau method for temporal logic: An overview. Logique et Analyse pp. 119–136 (1985)