Enumerating Minimal Unsatisfiable Cores of LTLf Formulae

Antonio Ielo[1], **Giuseppe Mazzotta**[1], **Rafael Peñaloza**[2], **Francesco Ricca**[1]

[1] University of Calabria, Rende, Italy
[2] University of Milano-Bicocca, Milano, Italy
antonio.ielo@unical.it, giuseppe.mazzotta@unical.it, francesco.ricca@unical.it
rafael.penaloza@unimib.it

Abstract

Linear Temporal Logic over finite traces (LTL_f) is a widely used formalism with applications in AI, process mining, model checking, and more. The primary reasoning task for LTL_f is satisfiability checking; yet, the recent focus on explainable AI has increased interest in analyzing inconsistent formulae, making the enumeration of minimal explanations for unsatisfiability a relevant task also for LTL_f. We introduce a novel technique for enumerating minimal unsatisfiable cores (MUCs) of an LTL_f specification. The main idea is to encode an LTL_f formula into an Answer Set Programming (ASP) specification, such that the minimal unsatisfiable subsets (MUSes) of the ASP program directly correspond to the MUCs of the original LTL_f specification. Leveraging recent advancements in ASP solving yields an MUC enumerator achieving good performance in experiments conducted on established benchmarks from the literature.

System — https://www.github.com/ainnoot/mus2muc
Experiments — https://osf.io/uthxv

Introduction

Linear temporal logic over finite traces (LTL_f) (De Giacomo and Vardi 2013) is a simple yet powerful language for expressing and reasoning about temporal specifications, which is particularly well-suited for Artificial Intelligence (AI) applications (Bacchus and Kabanza 1998; Calvanese, De Giacomo, and Vardi 2002; De Giacomo et al. 2016; De Giacomo and Vardi 1999). Perhaps its most widely recognised use to-date is as the logic underlying temporal process modelling languages such as Declare (Pesic, Schonenberg, and van der Aalst 2007). Very briefly, a Declare specification is a set of contraints on the potential evolution of a process, which is expressed through a syntactic variant of a subclass of LTL_f formulae. The full specification can thus be seen as a conjunction of LTL_f formulae. As specifications become bigger—specially when they are automatically mined from trace logs (Di Ciccio and Montali 2022)—it is not uncommon to encounter inconsistencies (i.e., business process models which are intrinsically contradictory) or other errors.

To understand and correct these errors, it is thus important to highlight the sets of formulae in the specification that are responsible for them (Niu et al. 2023; Roveri et al. 2024). Specifically, we are interested in computing the *minimal unsatisfiable cores* (MUCs): subset-minimal subsets of formulae (from the original specification) that are collectively inconsistent (Liffiton et al. 2016; Niu et al. 2023; Roveri et al. 2024). These can be seen as the prime causes of the error. Notably, a single specification can yield multiple MUCs of varying sizes, depending on the specific constraints involved. Exploring more than one MUC can be crucial for analyzing and understanding the causes of incoherence (as recognized in explainable AI (Miller 2019; Audemard, Koriche, and Marquis 2020)). Thus, a system capable of efficiently enumerating MUCs would be of significant value.

A similar problem has been studied for answer set programming (ASP) (Brewka, Eiter, and Truszczynski 2011; Gelfond and Lifschitz 1991), where the goal is to find *minimal unsatisfiable subsets* (MUS) of atoms that make an ASP program incoherent (Brewka, Thimm, and Ulbricht 2019; Mencía and Marques-Silva 2020; Alviano et al. 2023). Efficient implementations of MUS enumerators have been developed (Alviano et al. 2023).

Our goal is to take advantage of the declarativity of ASP and the efficiency of ASP systems to also enumerate MUCs of LTL_f formulae. Hence, we present a new transformation which constructs, given a set of LTL_f formulae, an ASP program whose MUSes are in a biunivocal correspondence with the MUCs of the original specification. Although we base our reduction on a well-known encoding of LTL_f bounded satisfiability (Fionda and Greco 2018; Fionda, Ielo, and Ricca 2024), the idea is general enough to be applicable to other decision procedures, as long as they are expressible in ASP. A progression strategy on trace lengths is used to generate efficiently MUCs (Morgado et al. 2013). To our knowledge, this is the first MUC enumerator for LTL_f.

We empirically compare our implementation with existing systems designed to produce only *one* MUC (or one potentially non-minimal unsatisfiable core) (Niu et al. 2023; Roveri et al. 2024) and observe that our (more general) system outperforms against those established systems on benchmarks from the literature. MUC enumeration is also very efficient.

Related Work

The task of computing MUCs has been considered, under different names, for several representation languages including propositional logic (Liffiton and Sakallah 2008), constraint satisfaction problems (Mencía and Marques-Silva 2014), databases (Meliou, Roy, and Suciu 2014), description logics (Schlobach and Cornet 2003), and ASP (Alviano et al. 2023; Brewka, Thimm, and Ulbricht 2019) among many others. For a general overview see (Peñaloza 2020).

Although the task was briefly studied for LTL (over infinite traces) in (Baader and Peñaloza 2010), it was only recently considered for the specific case of LTL_f (Niu et al. 2023; Roveri et al. 2024). Interestingly, for LTL_f the focus has been only on computing one (potentially non-minimal) unsatisfiable core. To our knowledge, we are the first to propose a full-fledged LTL_f MUC enumerator.

The idea of using a highly optimised reasoner from one language to enumerate MUCs from another one was already considered, first exploiting SAT solvers (Sebastiani and Vescovi 2009) and later on using ASP solvers (Peñaloza and Ricca 2022). Our approach falls into the latter class. Our reduction to ASP is inspired on the automata-based satisfiability procedure, previously used for SAT-based satisfiability checking (Fionda and Greco 2018; Li et al. 2020), alongside an incremental approach that verifies the (non-)existence of models up to a certain length. Notably, an ASP-based approach to compute minimal inconsistent sets for LTL_f under the assumption (not present in our paper) that traces have a *bounded known* length k was proposed (Kuhlmann and Corea 2024). Such k cannot be determined efficiently *a-priori* for general LTL_f formulae (Fionda and Greco 2018), so that approach cannot be used as-is in the general case.

Preliminaries

We briefly recap required notions of linear temporal logic over finite traces (LTL_f) (De Giacomo and Vardi 2013) and Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011).

Answer Set Programming

ASP Language. A *term* is either a *variable* or a *constant*, where *variables* are alphanumeric strings starting with uppercase letters, while *constants* are either integers or alphanumeric strings starting with lowercase letters. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate of ariety n and $t_1, \ldots, t_n$ are terms; it is *ground* if all its terms are constants. We say that an atom $p(t_1, \ldots, t_k)$ has *signature* p/k. An atom α *matches* a signature p/k if $\alpha = p(t_1, \ldots, t_k)$. A *literal* is either an atom a or its negation $not\, a$, where not denotes the negation as failure. A literal is said to be *negative* if it is of the form $not\, a$, otherwise it is positive. For a literal l, $\bar{l}$ denotes the complement of l. More precisely, $\bar{l} = a$ if $l = not\, a$, otherwise $\bar{l} = not\, a$. A *normal rule* is an expression of the form $h \leftarrow b_1, \ldots, b_n$ where h is an atom referred to as *head*, denoted by H_r, that can also be omitted, $n \geq 0$, and $b_1, \ldots, b_n$ is a conjunction of literals referred to as *body*, denoted by B_r. In particular a normal rule is said to be a *constraint* if its head is omitted, while it is said to be a *fact* if $n = 0$. A normal rule r is *safe* if each variable r appears at least in one positive literal in the body of r. A *program* is a finite set of safe normal rules. In what follows we will also use choice rules, which abbreviate complex expressions (Calimeri et al. 2020). A *choice element* is of the form $h : l_1, \ldots, l_k$, where h is an atom, and $l_1, \ldots, l_k$ is a conjunction of literals. A *choice rule* is an expression of the form $\{e_1; \ldots; e_m\} \leftarrow b_1, \ldots, b_n$, which is a shorthand for the set of normal rules $h_i \leftarrow l^i_1, \ldots, l^i_{k_i}, b_1, \ldots, b_n, not\, nh_i$; $nh_i \leftarrow l^i_1, \ldots, l^i_{k_i}, b_1, \ldots, b_n, not\, h_i$, for each $i \in 1, \ldots, m$ where e_i are of the form $h_i : l^i_1, \ldots, l^i_{k_i}$ and nh_i is a fresh atom not appearing anywhere else. Given a program P, and $r \in P$, $ground(r)$ is the set of ground instantiations of r obtained by replacing variables in r with constants in P; whereas $ground(P)$ is the union of ground instantiations of rules in P. Concerning the semantics of ASP, we recall that, given a program P, an *interpretation* I (i.e. a subset of the Herbrand's base of P) is an answer set of P iff (i) I is a model, namely for each rule $r \in ground(P)$ either the head of r is true wrt I or the body of r is false wrt I; and (iii) I is a minimal model of its GL-reduct (Gelfond and Lifschitz 1991). Given an answer set S and a signature σ, the *projection of S on σ* is the set $S_{|\sigma} = \{\alpha \in S : \alpha \text{ matches } \sigma\}$.

Minimal Unsatisfiable Subprograms. Consider a program P and a set of (fresh) objective atoms $O \subseteq \mathcal{B}_P$, where $\mathcal{B}_P$ is the Herbrand's base of atoms constructible from constants and predicate names in the logic program. For $S \subseteq O$, $\mathsf{enforce}(P, O, S)$ is the program obtained from P by adding a choice rule over atoms in O (i.e. $\{o_1; \ldots; o_n\} \leftarrow$) and a set of constraints of the form $\leftarrow not\, o$, for every $o \in S$. Intuitively, $\mathsf{enforce}(P, O, S)$ augments the program P such that the objective atoms can be arbitrarily chosen (i.e. either as true or false) but the atoms in S are *enforced* to be true.

An *unsatisfiable subset* for P wrt the set of objective atoms O is a set of atoms $U \subseteq O$ such that $\mathsf{enforce}(P, O, U)$ is incoherent (Alviano et al. 2023). $US(P, O)$ denotes the set of unsatisfiable subsets of P wrt O. An unsastisfiable subset $U \in US(P, O)$ is a *minimal unsatisfiable subset* (MUS) of P wrt O iff for every $U' \subset U$, $U' \notin US(P, O)$.

Linear Temporal Logic over Finite Traces

Linear Temporal Logic (LTL) (Pnueli 1977) is an extension of propositional logic enabling reasoning over infinite sequences of propositional interpretations or *traces*. LTL_f (De Giacomo and Vardi 2013) is a variant of this logic that considers only finite traces. Let $\mathcal{A}$ be a finite set of propositional symbols. The class of LTL_f formulae over $\mathcal{A}$ is defined through the grammar $\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi\, \mathsf{U}\, \varphi \mid \mathsf{X}\, \varphi \mid \top$ where $a \in \mathcal{A}$. A formula is in *conjunctive form* if it is expressed as a conjunction of formulae. In this case, we often represent a formula as the set of its conjuncts; i.e., the formula $\phi_1 \wedge \cdots \wedge \phi_k$ is expressed by the set $\{\phi_1, \ldots, \phi_k\}$.

A *state* is a subset of $\mathcal{A}$; a *trace* is a finite sequence $\pi = \sigma_0 \cdots \sigma_k$ of states; it has length $|\pi| = k+1$. The i-th state of the trace π is denoted by $\pi(i) = \sigma_i$ The *satisfaction relation* is defined recursively over the structure of φ. We say that the trace π satisfies φ at time i, $0 \leq i < |\pi|$, denoted by

$\pi, i \models \varphi$ iff (i) $\pi, i \models a \in \mathcal{A}$ iff $a \in \pi(i)$; (ii) $\pi, i \models \top$; (iii) $\pi, i \models \mathsf{X}\,\varphi'$ iff $i < |\pi| - 1$ and $\pi, i+1 \models \varphi'$; (iv) $\pi, i \models \psi \wedge \phi$ iff $\pi, i \models \psi$ and $\pi, i \models \phi$; and (v) $\pi, i \models \psi\ \mathsf{U}\ \phi$ if there exists $i \leq j < |\pi|$ s.t. $\pi, j \models \phi$ and for all $i \leq k < j$, $\pi, k \models \psi$. The trace π is a *model* of φ ($\pi \models \varphi$) whenever $\pi, 0 \models \varphi$. The *satisfiability problem* consists on deciding whether a formula φ admits a model; i.e., if there exists π such that $\pi \models \varphi$. This problem is well-known to be PSPACE-complete (De Giacomo and Vardi 2013).

Given an unsatisfiable formula $\varphi = \{\phi_1, \ldots, \phi_k\}$ in conjunctive form, a *minimal unsatisfiable core* (MUC) of φ is an unsatisfiable formula $\psi \subseteq \varphi$ which is minimal (wrt set inclusion); i.e., removing any conjunct from ψ yields a satisfiable formula (Niu et al. 2023). It is known that a single formula may have exponentially many MUCs, but computing one MUC requires only polynomial space; just as deciding satisfiability (Peñaloza 2019, 2020).

Leveraging ASP for MUC Enumeration

Our approach relies on leveraging ASP *minimal unsatisfiable sets* (MUSes) enumeration algorithms to generate a sequence of candidate *minimal unsatisfible cores* (MUCs) for an LTL_f formula. In order to put in place such approach, a formal connection between these objects must be established. In this section, we introduce the notion of *probe* and *k-MUC* to investigate this relationship. The notion of *probe* abstracts the properties of ASP encodings that are suitable for MUC enumeration in our algorithm; and, k-MUCs are a *relaxation* wrt model length of the concept of MUC, which is revealed to be more suitable for ASP-based reasoning.

MUS and Probes

In this paper, we consider LTL_f specifications in *conjunctive form*. Conjunctive specifications naturally arise in LTL_f, as logical conjunction is the natural way to combine subformulae that encode distinct requirements. This is the standard setting for MUC computation used in LTL_f specification, used both by (Niu et al. 2023) and (Roveri et al. 2024). Following the notation by (Niu et al. 2023), let $\varphi = \{\phi_1, \ldots, \phi_n\}$ be a formula in conjunctive form, where ϕ_i is a *conjunct* of φ. With a slight abuse of notation, we will identify φ with the *set* of its conjuncts.

Our starting point is that there exists a uniform way to encode LTL_f formulae in conjunctive normal form into logic programs. We are interested in *encodings where original conjuncts of φ can be told apart by means of special atoms*.

Definition 1 (Reification Function)**.** *A reification function for a formula φ is a function mapping φ into a logic program whose Herbrand base contains an atom $phi(i)$ for each $\phi_i \in \varphi$. $\mathcal{O}(\varphi)$ denotes the set of atoms matching signature $phi/1$.*

Each subset $S \subseteq \mathcal{O}(\varphi)$ uniquely identifies the set of conjuncts $\psi = \{\phi_i : phi(i) \in \mathcal{O}(\varphi)\} \subseteq \varphi$. Therefore, we denote ψ by $\mathsf{Formula}(S)$ and S by $\mathsf{Atoms}(\psi)$. Reification functions allow to encode LTL_f formulae into logic programs. Since in this paper we are concerned with notions of *satisfiability*, *unsatisfiability* wrt *subset minimality* of LTL_f formulae, among all possible reification functions, we are interested in ones that preserve as much information about these properties. We thus introduce the notion of *probe*.

Definition 2 (k-Probe)**.** *Let $k \in \mathbb{N}$. A reification function ρ is a* probe of depth k *(or* k-probe *for short) for φ if for each set $S \subseteq \mathcal{O}(\varphi)$ it holds that $\mathsf{Formula}(S)$ admits a model of length at most k if and only if there exists an answer set M of $\rho(\varphi)$ such that $S = M_{|\mathcal{O}(\varphi)}$.*

There exist multiple ASP encodings that satisfy the definition of probe: one can obtain a probe by adapting any ASP encoding for bounded LTL_f satisfiability (Fionda, Ielo, and Ricca 2024). We focus on how probes relate to MUCs of φ.

Lemma 3. *Let ρ be a probe of depth k for φ and S a minimal unsatisfiable subset of $\rho(\varphi)$ wrt the objective atoms $\mathcal{O}(\varphi)$. $\mathsf{Formula}(S)$ is either an MUC of φ or it is satisfiable but its shortest satisfying trace has length greater than k.*

Proof. Assume S is a minimal unsatisfiable subset wrt $\mathcal{O}(\varphi)$. Then, all its (proper) subsets can be extended to answer sets — thus, interpreting them as formulae yields an LTL_f formula that admits a model of length at most k, by Definition 2. Hence, all proper subsets of $\mathsf{Formula}(S)$ are satisfiable, while $\mathsf{Formula}(S)$ itself is either unsatisfiable or its shortest model trace has a length greater than k. In the former case, it matches the definition of MUC. □

Example 4. *Consider the formula $\varphi = \{\mathsf{X}^5\beta, \mathsf{X}^5\neg\beta\}$, where $\mathsf{X}^i\varphi$ is a shorthand for the repeat application of X i times (e.g., $\mathsf{X}\,(\mathsf{X}\,(\mathsf{X}\,(\mathsf{X}\,(\mathsf{X}\,(\varphi)))))$). This formula has a unique MUC, namely $\mathsf{X}^5\beta \wedge \mathsf{X}^5\neg\beta$. If we consider a probe $\rho_3 = \rho(3, \varphi)$, it has two MUSes, namely $\{\mathsf{X}^5\beta\}, \{\mathsf{X}^5\neg\beta\}$, since these formulae do not admit models of length at most 3. Using probes of depth at least 5, we can detect the MUC through the MUS $\{\mathsf{X}^5\beta, \mathsf{X}^5\neg\beta\}$.*

Formulae exhibiting the property stated in Lemma 3 are the key objects which allow us to levarage MUS enumeration to enumerate MUCs, yielding the next definition.

Definition 5 (k-bound MUC)**.** *Let $k \in \mathbb{N}$. A* k-bound MUC *(or* k-MUC*) for the formula φ is a minimal subset of φ that does not admit a model of length at most k. We denote by $\mathsf{MUC}^k(\varphi)$ the set of all k-MUCs for a formula φ.*

Lemma 6. *Let $S \in \mathsf{MUC}^k(\varphi)$. If S is unsatisfiable, then S is an MUC for φ.*

Proof. Given that $S \in \mathsf{MUC}^k(\varphi)$, it means that any proper subset of S admits a model of length at most k, hence it is satisfiable. Thus, since S is unsatisfiable, it is an MUC. □

Lemma 3 can be rewritten by applying Definition 5, to obtain the following:

Lemma 7. *Let ρ be a probe of depth k for φ and S a minimal unsatisfiable subset of $\rho(\varphi)$ wrt the objective atoms $\mathcal{O}(\varphi)$. $\mathsf{Formula}(S)$ is a k-MUC of φ.*

Proof. By Lemma 3, we have that $\mathsf{Formula}(S)$ is either unsatisfiable or all its models have length at least $k+1$. This is exactly the definition of k-MUC. □

Example 4 highlights an interesting property. The probe at depth $k = 3$ yields two (singleton) MUSes that intepreted as formulae do not admit models of length at most k. However, increasing the probe depth to $k' = 5$ yields a *single MUS*, since the MUSes (of the previous probe) are both satisfiable if we consider models of length at most k', but jointly unsatisfiable considering models of length at most k'. Intuitively, this makes the probe at depth k' *more effective*, since it enables *discarding* MUSes not leading to an MUC.

In this regard, with the aim of enumerating MUCs, the most interesting probes would be the ones that allow to detect all MUCs with no false positives. More formally, the *most effective probe* is a probe at a depth k^* such that for each $k' \geq k^*$ it holds if α is an MUS in a k^*-probe, it will also be an MUS in the k'-probe. We provide an argument to show that such a probe depth k^* exists.

If $\varphi = \{\phi_1, \ldots, \phi_n\}$, φ can have at most 2^n MUCs. Let $h(\varphi)$ be the least integer $z \in \mathbb{N}$ such that *any* satisfiable subset of φ admits a model of length at most z. We refer to $h(\varphi)$ as the *completeness threshold* for φ, and probes of depth greater or equal to $h(\varphi)$ as *complete probes*.

This leads us to establishing a bijection between MUSes of complete probes and MUCs of φ.

Theorem 8. *If ρ is a complete probe for φ, then S is an MUS of P wrt $\mathcal{O}(\varphi)$ if and only if* Formula(S) *is an MUC of φ.*

Proof. ($\rightarrow$) Let S be an MUS of ρ with respect to $\mathcal{O}(\varphi)$. By Lemma 7 Formula(S) is a k-MUC, thus it is either unsatisfiable; or satisfiable with a satisfying trace with length greater than k — in this latter case, ρ would not be a complete probe. Hence, Formula(S) must be an MUC for φ. ($\leftarrow$) Let ψ be an MUC of $\varphi = \{\phi_1, \ldots, \phi_n\}$. Without loss of generality, we can assume $\psi = \{\phi_1, \ldots, \phi_m\}$. All its proper subsets ψ^j — which denotes the LTL$_\text{f}$ formula obtained by removing from ψ the j-th conjunct — are satisfiable. Since ρ is a complete probe, for each ψ^j there exists an answer set of $\rho(\varphi)$ that extends Atoms(ψ^j), but there exists no answer set that extends Atoms(ψ). Since Atoms$(\psi^j) =$ Atoms$(\psi) \setminus \{phi(j)\}$, this shows that Atoms$(\psi)$ is an MUS for $\rho(\varphi)$ wrt $\mathcal{O}(\varphi)$. □

Theorem 8 characterizes MUCs of φ as MUSes of complete probes for φ. There is no efficient way of computing the completeness threshold, which can be exponential in the size of the formula (De Giacomo and Vardi 2013; Maggi, Montali, and Peñaloza 2020; Fionda and Greco 2018). Nonetheless, experiments show approaching the problem by iteratively expanding the probe depth works well in practice.

MUC Enumeration by MUS Enumeration

We know by Theorem 8 that we can enumerate MUCs of φ by enumerating MUSes of a complete probe for φ. In general, computing the completeness threshold for φ is not feasible. By Lemma 6, we also know that *some* k-MUCs with $k \leq h(\varphi)$ could also be MUCs. These results suggest two anytime LTL$_\text{f}$ MUC enumeration algorithms: *(i)* an algorithm (Algorithm 1) that computes all MUCs among the k-MUCs for a given k, and *(ii)* its iterative deepening variant (Algorithm 2) which expands the probe depth k whenever a k-MUC is not unsatisfiable. Both algorithms make use of the subroutines `probe`, `enumerate_mus`, `to_formula`, `check_satisfiability` explained next: **`probe`**(φ, k) (the counterpart of $\rho(k, \varphi)$) builds the logic program from which we will extract k-MUCs; **`enumerate_mus`**(P) invokes an ASP solver to extract MUSes of the probe P wrt the objective atoms Φ; **`to_formula`**(x) given a MUS x of P, rebuilds the LTL$_\text{f}$ formula Formula(x); **`check_satisfiability`**(ψ) determines wheter an LTL$_\text{f}$ formula is satisfiable and returns the length of a satisfying trace (or -1 if unsatisfiable).

Algorithm 1: Enumerate unsatisfiable k-MUCs

```
1: def enumerate_k_mucs(φ, k):
2:     mucs = []
3:     P = probe(φ, k)
4:     for x in enumerate_mus(P):
5:         ψ = to_formula(x)
6:         k' = check_satisfiability(ψ):
7:         if k' = −1:
8:             mucs.append(ψ)
9:     return mucs
```

Algorithm 2: Enumerate MUCs - Iterative Deepening

```
1: def enumerate_mucs(φ):
2:     k = 1
3:     complete = False
4:     mucs = []
5:     while not complete:
6:         P = probe(φ, k)
7:         complete = True
8:         for x in enumerate_mus(P):
9:             ψ = to_formula(x)
10:            if ψ ∈ mucs:
11:                skip
12:            k' = check_satisfiability(ψ):
13:            if k' = −1:
14:                mucs.append(ψ)
15:            else:
16:                k = k'
17:                complete = False
18:                break
19:    return mucs
```

Both algorithms are compatible with any ASP solver that implements MUS enumeration (that is, an implementation of the procedure `enumerate_mus`) and (complete) LTL$_\text{f}$ solvers that can *(i)* provide a satisfying trace length for satisfiable formulae *(ii)* prove unsatisfiability (that is, an implementation of the procedure `check_satisfiability`).

Algorithm 1 is straightforward. We enumerate MUSes of a k-probe, which yields a sequence of k-MUCs. Each k-MUC is a *candidate* MUC for φ, to be *certified* or *disproved* by a call to an LTL$_\text{f}$ satisfiability oracle. Following such a call, we discard *false positive* candidates (i.e., k-MUCs that

are actually satisfiable) as we meet them. This approach does not detect all MUCs, unless $k \geq h(\varphi)$. Conjuncts whose shortest model has length greater than k will be discarded. Algorithm 2 extends Algorithm 1. A false positive k-MUC ψ is a witness of the fact the current k is below the completeness threshold for φ. Thus, we increase k according to the length of the model π that satisfies ψ. Since $h(\varphi)$ is finite, k will eventually converge to $h(\varphi)$. At that point, all k-MUCs of the $h(\varphi)$-probe result in MUCs for φ.

Experiments

We implemented our strategies in the system `mus2muc`, using a probe based on the bounded satisfiability encoding presented by Fionda, Ielo, and Ricca (2024). We performed different experiments, to answer the following research questions: (***Q1***) *how does* `mus2muc` *perform in computing a single MUC?*; (***Q2***) *how effective is* `mus2muc` *in enumerating MUCs?*; (***Q3***) *which* `mus2muc` *component affects the performance the most? and* (***Q4***) *how does* `mus2muc` *fare against a domain-agnostic MUC enumeration tool?*

Implementation of `mus2muc`. `mus2muc` is implemented in Python 3.12 following Algorithm 2. It uses the ASP solver `wasp` as a MUS generator, and the LTL_f solver `aaltaf` as a satisfiability solver. More in detail, `wasp` performs state-of-the-art MUS enumeration (Alviano et al. 2023) over the probe described in the previous section. As soon as a candidate k-MUC (i.e., a MUS of the probe) becomes available, an instance of the LTL_f solver is invoked as a certifier, in a typical producer-consumer architecture. Furthermore, since multiple k-probes (for increasing value of k) are used, it is possible for k-MUCs to be produced multiple times (for different values of k). To avoid redundant calls to the LTL_f solver, we adopt a caching strategy on the MUS generator side. As stated before, our system is anytime, and outputs MUCs as soon as they are certified.

Benchmarks. In our experiments we consider a benchmark suite consisting of common formulae families used in LTL and LTL_f literature to evaluate solvers. We use all unsatisfiable formulae that appear in (Schuppan and Darmawan 2011), and randomly generated formulae from (Li et al. 2020). These formulae have been previously used by (Niu et al. 2023) to benchmark single MUC computation, and by (Roveri et al. 2024) for single UC (with no minimality guarantee) extraction. These benchmarks contain unsatisfiable instances from 15 different applications domains, each with different formula features. They comprise instances from applications (13 domains) and randomly generated (2 domains). The random instances were proposed by Vardi et.al. (Li et al. 2020) and consist of conjunctions of Declare patterns (Di Ciccio and Montali 2022). In total, the suite contains 2079 unsatisfiable instances, obtained from (Schuppan and Darmawan 2011). We transform all instances in conjunctive form by traversing the formula tree in a top-down fashion, stopping whenever formulae are not conjunctions. This is consistent with how these instances have been handled by Niu et al. and Roveri et al.. All formulae are interpreted as LTL_f formulae.

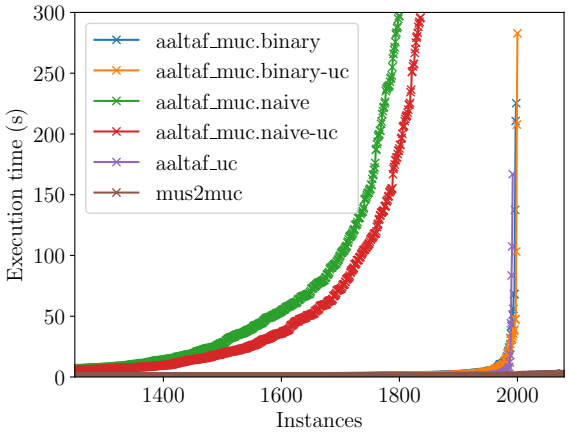


Figure 1: Computation of a single MUC or UC for hardest instances.

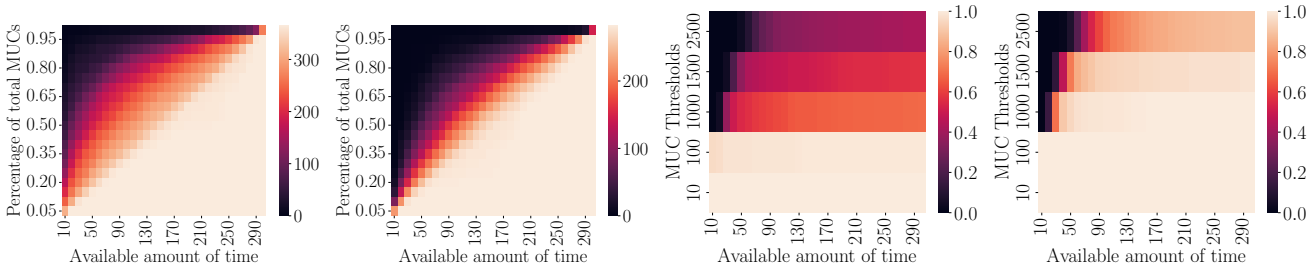
Benchmark	Instances	Full Enum.	Sum of MUCs	
			Full Enum.	Timeout
acacia.demo-v3	11	11	77	-
alaska.lift	129	129	8310	-
forobots.forobots	38	38	38	-
schuppan.O1formula	27	27	27	-
schuppan.O2formula	27	27	27	-
trp.N12x	400	400	14380	-
trp.N5x	240	240	4210	-
anzu.amba	34	2	362	20642
anzu.genbuf	36	6	1043	16119
rozier.counter	76	62	514	35
schuppan.phltl	13	9	219	2760
trp.N12y	67	3	2514	116365
trp.N5y	46	33	121704	287380
rand.C100	500	134	15636	1295793
rand.V20	435	152	64049	1619065

Table 1: MUC enumeration statistics. A benchmark $x.y$ denotes that the set of formulae y is a *family* of benchmark x.

Execution environment. The experiments were run on a system with 2.30GHz Intel(R) Xeon(R) Gold 5118 CPU and 512GB of RAM with Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-137-generic x86_64). For each run in the experiment, memory and time were limited to 8GB and 300s of real time, 700s of CPU time respectively.

Q1: Extraction of a Single MUC. First of all we assess the performance of our implementation in the computation of a single MUC. We compare to algorithms introduced by (Niu et al. 2023), namely `aaltaf_muc.naive` and `aaltaf_muc.binary`, as well as two heuristic variants `aaltaf_muc.naive-uc` and `aaltaf_muc.binary-uc`. (For details on these systems see (Niu et al. 2023).) We also include in the comparison the best performing approach introduced in (Roveri et al. 2024), namely `aaltaf-uc`. Notice that, differently from our approach and the approaches of (Niu et al. 2023), `aaltaf-uc` does not provide any minimality guarantees, but only provides an unsatisfiable core for the input formula.

Figure 1 reports the performance of different systems. In particular, the left tail of the cactus plot has been trun-



(a) C100 Rand class. (b) V20 Rand class. (c) C100 Rand class. (d) V20 Rand class.

Figure 2: (a)-(b): Percentage of not-fully-enumerated instances (among the formula families C100 Rand, V20 Rand) that are able to enumerate at least a given number of MUCs in a certain amount of time. A cell (i, j) in the heatmap represents the percentage of timed-out instances that are able to enumerate j MUCs in i seconds. (c)-(d): Number of not-fully-enumerated instances (among the formula families rand C100, rand V20) that are able to enumerate the y percent of found MUCs within x seconds of runtime. A cell (i, j) in the heatmap represents how many instances in the given formula family can enumerate i percent of the found MUCs found in 300s (up to timeout) in j seconds.

cated, since it consists of the trivial instances (i.e. found an MUC/UC in negligible time) for each system. Overall, we can observe that the `aaltaf-muc.binary` and `aaltaf-muc.binary-uc` are faster than `aaltaf-uc`, although the task solved by `aaltaf-uc` is easier (since it does not provide minimality guarantees on the UC). These results match the experimental results in (Niu et al. 2023). Overall, `mus2muc` outperforms all systems in this task.

Q2: Enumeration of MUCs. Our second experiment consists in evaluating `mus2muc` effectiveness in *enumerating* MUCs of the formulae in the benchmark suite. Table 1 reports the number of fully enumerated instances, total number found MUCs partitioned across fully-enumerated instances and timed-out instances. Families showcase very heterogeneous behavior. Overall, `mus2muc` is effective in providing MUCs for all considered instances; 1273 instances out of 2079 across all families can be fully enumerated. Among families that are not fully enumerated, second block of Table 1, the system was still able to provide a significant number of MUCs (e.g., more than 1M MUCS in the `randV20` family overall). The `schuppan.phltl` family consists of mostly fully enumerated instances (9 out of 13), yet these amount to only 7% of the total number of MUCs found in the family. Viceversa, `rozier.counter` features as well most of the instances solved (62 out of 76), but here the solved instances amount to 93% of all the MUCs found within timeout, for an average of about 8 MUCs per solved instance. The `anzu.amba` and `anzu.genbuf` families are characterized by a huge number of MUCs, in the order of 10^4, with almost none of the instances being fully-enumerated; similarly, the `trp` and `rand` families feature an even higher amount of MUCs, in the order of 10^5 and 10^6. We deepen our analysis studying how fast `mus2mc` extracts MUCs, focusing on those families which contain the most MUCs—`rand`. The heatmaps in Figure 2a and 2b report, for distinct families, in a cell (x, y) the percentage of instances for which `mus2muc` can produce at least y MUCs in at most x seconds. Even on these "hard" formulae, our approach is able to output a considerable amount of MUCs in short time, albeit not able to fully enumerate them within the time limit. Conversely, the heatmaps in Figures 2c- 2d report how MUCs are "temporally distributed" within the timeout. For distinct families, a cell (x, y) contains the number of instances where it is possible to find y percent of found MUCs within x seconds. We can see that, in the majority of these instances, MUCs are computed in a steady fashion and MUCs become available within seconds of runtime. Indeed, these families are characterized by a huge number of MUCs that cannot be realistically inspected. It is worth noting that, even in this scenario, our approach can provide a significant number of MUCs within few seconds.

Q3: Performance of MUS generation and MUC certification. In the `mus2muc` system, as outlined in Algorithm 2, each (unique) MUS extracted from the probe is checked for satisfiability using an LTL_f solver. The MUS is either *certified* (i.e., determined to be unsatisfiable) or *disproved* (i.e., a satisfying trace is found with a length exceeding the current probe depth). Thus, it might be interesting to study which component (among generation and checking) affects runtime the most. In particular, when performing MUC extraction over an instance F, a certain amount of seconds due to MUS generation and MUS certification are accrued. The scatter plot in Figure 3 compares total CPU time spent on MUS generation and MUC certification, for fully-enumerated instances. Different families exhibit heterogeneous behavior; hence, we focus on some representative examples. Instances of the `rand.V100` and `rand.V20` families spend roughly the same amount of time between the two components, as they lie close to the bisector. On the other hand, instances of the `alaska.lift` family require a more intensive certification phase than generation one, as they lie above the bisector. Overall, almost no instance requires significantly higher generation time than certification time, see the sparsely populated area below the bisector.

Q4: Domain-agnostic MUCs enumeration techniques. As far as we know, no publicly available systems work out of the box to enumerate MUCs of LTL_f formulae. However,

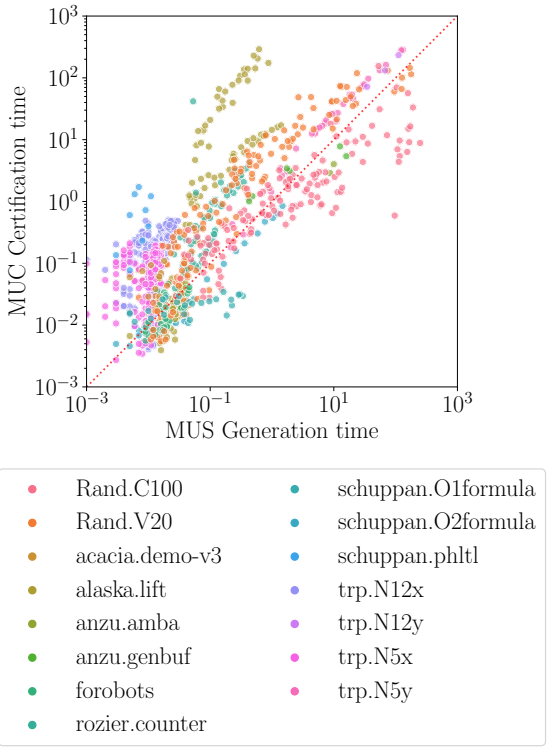


Figure 3: A point (x, y) in the scatter plot represents that a fully-enumerated instance has spent x CPU seconds generating k-MUCs (e.g., ASP MUS enumeration) and y CPU seconds certifying k-MUCs (e.g., an LTL_f solver running to prove unsatisfiability). Each color denotes a formula family.

a number of *general purpose*, *domain-agnostic* MUC extraction algorithms (which also support LTL as a domain, using SPOT (Duret-Lutz et al. 2016) as a satisfiability backend) are available (Bendík and Cerná 2020). The `must` tool supports several of such algorithms, and so, in what follows we compare our MUC enumeration technique with the ones supported by `must`. Note that, since `must` supports the LTL domain but not LTL_f, we patch `must` by applying the well-known LTL-to-LTL_f transformation presented by De Giacomo and Vardi (2013). Figure 4 compares the number of found MUCs of each `must` variant wrt `mus2muc`. A point (x, y) in Figure 4, denotes that for a given instance in the benchmarks suite `mus2muc` has computed x MUCs whereas a `must` algorithm has computed y MUCs. Colors distinguish different `must` algorithms. Overall, `mus2muc` enumerates more MUCs than any of the `must` variants. In some extreme cases, `mus2muc` computes order of magnitude more MUCs then `must` algorithms (see the points that lie on the x-axis). This is however expected as `mus2muc` MUS generation phase is aimed at detecting k-MUCs, while `must` is a general tool.

Conclusions

Satisfiability of temporal specifications expressed in LTL_f play an important role in several artificial intelligence ap-

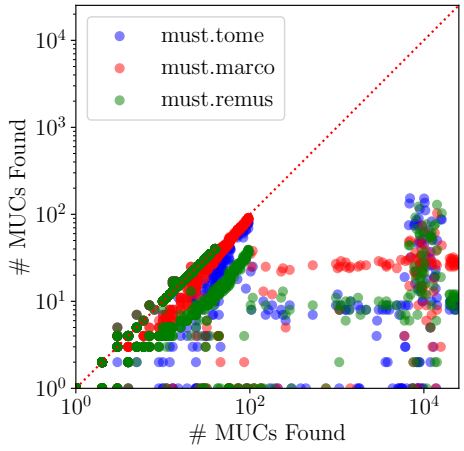


Figure 4: Number of MUCs enumerated within timeout for each instance.

plication domains (Bacchus and Kabanza 1998; Calvanese, De Giacomo, and Vardi 2002; De Giacomo et al. 2016; De Giacomo and Vardi 1999; De Giacomo et al. 2019). Therefore, in case of unsatisfiable specifications, detecting reasons for unsatisfiability—computing its minimal unsatisfiable cores—is of particular interest. This is especially true whenever the specification under analysis *is expected to be satisfiable*. Recent works (Niu et al. 2023; Roveri et al. 2024) propose several approaches for single MUC computation, but do not investigate enumeration techniques for MUCs. However, enumerating MUCs for LTL_f specifications is pivotal to enabling several reasoning services, such as some explainability tasks (Miller 2019), as it is the case for propositional logic (Marques-Silva 2010; Marques-Silva, Janota, and Mencía 2017). We propose an approach for characterizing MUCs of LTL_f formulae as minimal unsatisfiable subprograms (MUS) of suitable logic programs, introducing the notion of probe. This enables to implement LTL_f MUC enumeration techniques by exploiting off-the-shelf ASP and LTL_f reasoners, similarly to SAT-based domain agnostic MUC enumeration techniques à la Bendík and Cerná. Our approach is modular wrt ASP and LTL_f reasoners, which essentially constitute two sub-modules of the system, and wrt the logic program that is used to extract MUCs via its MUSes. We implement this strategy in `mus2muc`, using the ASP solver `wasp` and the LTL_f solver `aaltaf`. Our experiments show that `mus2muc` is effective at enumerating MUCs of unsatisfiable formulae that are commonly used in LTL_f literature as benchmarks, as well as being competitive with the state-of-the-art for single MUC computation. To our knowledge, this represent the first attempt to address this task in LTL_f. For future work, we are interested in studying how the choice of probes affects MUCs computation in our setting, as well as providing ad-hoc implementations for closely related LTL_f tasks, such explaining and repairing incosistent Declare specification in the realm of process mining.

Acknowledgments

This work was partially supported by the Italian Ministries MIMIT, under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005, project ASVIN n. F/360050/01-02/X75 CUP B29J2400020000; and MUR, under projects: PNRR FAIR - Spoke 9 - WP 9.1 and WP 9.2 CUP H23C22000860006, Tech4You CUP H23C22000370006, and PRIN PINPOINT CUP H23C22000280006.

References

Alviano, M.; Dodaro, C.; Fiorentino, S.; Previti, A.; and Ricca, F. 2023. ASP and subset minimality: Enumeration, cautious reasoning and MUSes. *Artif. Intell.*, 320: 103931.

Audemard, G.; Koriche, F.; and Marquis, P. 2020. On Tractable XAI Queries based on Compiled Representations. In *KR*, 838–849.

Baader, F.; and Peñaloza, R. 2010. Automata-Based Axiom Pinpointing. *J. Autom. Reason.*, 45(2): 91–129.

Bacchus, F.; and Kabanza, F. 1998. Planning for Temporally Extended Goals. *Ann. Math. Artif. Intell.*, 22(1-2): 5–27.

Bendík, J.; and Cerná, I. 2020. MUST: Minimal Unsatisfiable Subsets Enumeration Tool. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, 135–152. Springer.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM*, 54(12): 92–103.

Brewka, G.; Thimm, M.; and Ulbricht, M. 2019. Strong inconsistency. *Artif. Intell.*, 267: 78–117.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2): 294–309.

Calvanese, D.; De Giacomo, G.; and Vardi, M. Y. 2002. Reasoning about Actions and Planning in LTL Action Theories. In *KR*, 593–602.

De Giacomo, G.; Iocchi, L.; Favorito, M.; and Patrizi, F. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *ICAPS*, 128–136. AAAI Press.

De Giacomo, G.; Maggi, F. M.; Marrella, A.; and Sardiña, S. 2016. Computing Trace Alignment against Declarative Process Models through Planning. In *ICAPS*, 367–375.

De Giacomo, G.; and Vardi, M. Y. 1999. Automata-Theoretic Approach to Planning for Temporally Extended Goals. In *ECP*, volume 1809 of *LNCS*, 226–238.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*, 854–860. IJCAI/AAAI.

Di Ciccio, C.; and Montali, M. 2022. Declarative Process Specifications: Reasoning, Discovery, Monitoring. In van der Aalst, W. M. P.; and Carmona, J., eds., *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, 108–152. Springer.

Duret-Lutz, A.; Lewkowicz, A.; Fauchille, A.; Michaud, T.; Renault, E.; and Xu, L. 2016. Spot 2.0 - A Framework for LTL and \omega -Automata Manipulation. In *ATVA*, volume 9938 of *Lecture Notes in Computer Science*, 122–129.

Fionda, V.; and Greco, G. 2018. LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner. *J. Artif. Intell. Res.*, 63: 557–623.

Fionda, V.; Ielo, A.; and Ricca, F. 2024. LTLf2ASP: LTLf Bounded Satisfiability in ASP. In *LPNMR*, volume 15245 of *Lecture Notes in Computer Science*, 373–386. Springer.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Gener. Comput.*, 9(3/4): 365–386.

Kuhlmann, I.; and Corea, C. 2024. Inconsistency Measurement in LTL $_{\mathrm{f}}$ Based on Minimal Inconsistent Sets and Minimal Correction Sets. In *SUM*, volume 15350 of *Lecture Notes in Computer Science*, 217–232. Springer.

Li, J.; Pu, G.; Zhang, Y.; Vardi, M. Y.; and Rozier, K. Y. 2020. SAT-based explicit LTLf satisfiability checking. *Artif. Intell.*, 289: 103369.

Liffiton, M. H.; Previti, A.; Malik, A.; and Marques-Silva, J. 2016. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2): 223–250.

Liffiton, M. H.; and Sakallah, K. A. 2008. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reason.*, 40(1): 1–33.

Maggi, F. M.; Montali, M.; and Peñaloza, R. 2020. Temporal Logics Over Finite Traces with Uncertainty. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, 10218–10225. AAAI Press.

Marques-Silva, J. 2010. Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper). In *ISMVL*, 9–14. IEEE Computer Society.

Marques-Silva, J.; Janota, M.; and Mencía, C. 2017. Minimal sets on propositional formulae. Problems and reductions. *Artif. Intell.*, 252: 22–50.

Meliou, A.; Roy, S.; and Suciu, D. 2014. Causality and Explanations in Databases. *Proc. VLDB Endow.*, 7(13): 1715–1716.

Mencía, C.; and Marques-Silva, J. 2014. Efficient Relaxations of Over-constrained CSPs. In *Proceedings of 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, 725–732. IEEE Computer Society.

Mencía, C.; and Marques-Silva, J. 2020. Reasoning About Strong Inconsistency in ASP. In *SAT*, volume 12178 of *Lecture Notes in Computer Science*, 332–342. Springer.

Miller, T. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artif. Intell.*, 267: 1–38.

Morgado, A.; Heras, F.; Liffiton, M. H.; Planes, J.; and Marques-Silva, J. 2013. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints An Int. J.*, 18(4): 478–534.

Niu, T.; Xiao, S.; Zhang, X.; Li, J.; Huang, Y.; and Shi, J. 2023. Computing minimal unsatisfiable core for LTL over finite traces. *Journal of Logic and Computation*, exad049.

Peñaloza, R. 2019. Explaining Axiom Pinpointing. In Lutz, C.; Sattler, U.; Tinelli, C.; Turhan, A.; and Wolter, F., eds., *Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday*, volume 11560 of *Lecture Notes in Computer Science*, 475–496. Springer.

Peñaloza, R. 2020. Axiom Pinpointing. In Cota, G.; Daquino, M.; and Pozzato, G. L., eds., *Applications and Practices in Ontology Design, Extraction, and Reasoning*, volume 49 of *Studies on the Semantic Web*, 162–177. IOS Press.

Peñaloza, R.; and Ricca, F. 2022. Pinpointing Axioms in Ontologies via ASP. In Gottlob, G.; Inclezan, D.; and Maratea, M., eds., *Proceedings of LPNMR 2022*, volume 13416 of *Lecture Notes in Computer Science*, 315–321. Springer.

Pesic, M.; Schonenberg, H.; and van der Aalst, W. M. P. 2007. DECLARE: Full Support for Loosely-Structured Processes. In *Proceedings of EDOC 2007*, 287–300. IEEE Computer Society.

Pnueli, A. 1977. The Temporal Logic of Programs. In *FOCS*, 46–57. IEEE Computer Society.

Roveri, M.; Ciccio, C. D.; Francescomarino, C. D.; and Ghidini, C. 2024. Computing Unsatisfiable Cores for LTLf Specifications. *J. Artif. Intell. Res.*, 80: 517–558.

Schlobach, S.; and Cornet, R. 2003. Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies. In Gottlob, G.; and Walsh, T., eds., *Proceedings of IJCAI'03*, 355–362. Morgan Kaufmann.

Schuppan, V.; and Darmawan, L. 2011. Evaluating LTL Satisfiability Solvers. In *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, 397–413. Springer.

Sebastiani, R.; and Vescovi, M. 2009. Axiom Pinpointing in Lightweight Description Logics via Horn-SAT Encoding and Conflict Analysis. In Schmidt, R. A., ed., *Procedding of the 22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, 84–99. Springer.