Computing Syntax Tree-based Minimal Unsatisfiable Cores of LTLf Formulas

Valeria Fionda,Antonio Ielo,Francesco Ricca

University of Calabria, Rende, Italy
valeria.fionda@unical.it, antonio.ielo@unical.it, francesco.ricca@unical.it

Abstract

Linear Temporal Logic on Finite Traces (LTL_f) is a popular logic to express declarative specifications in Artificial Intelligence (AI). The recent call for explainable AI tools has made relevant the problem of computing efficiently minimal unsatisfiable cores (MUCs) and minimal correction sets (MCSes) of LTL_f formulas. Recent work has focused on the extraction of MUCs on formulas in conjunctive form. In this paper, we present a method that operates on arbitrary formulas and computes a more refined notion of MUCs, as introduced by Schuppan, along with the corresponding notion of MCSes. Experiments show that our system, based on Answer Set Programming, outperforms available tools.

Introduction

Linear Temporal Logic on Finite Traces (LTL_f) (De Giacomo and Vardi 2013) has emerged as a powerful formalism for specifying temporal properties in Artificial Intelligence applications, particularly in areas such as planning, business process modeling, and formal verification (Bacchus and Kabanza 1998; Di Ciccio and Montali 2022). In many practical settings, LTL_f specifications can become inconsistent, due to either conflicting requirements or unintended interactions between temporal constraints. Identifying and resolving such inconsistencies is critical for ensuring the reliability and interpretability of temporal models (Corea et al. 2024). In practice, specifications are often evaluated under *bounded* semantics, where traces are restricted to a fixed length k, particularly when embedded in symbolic or bounded model checking frameworks (Latvala et al. 2004). This bounded-LTL setting is both practically useful and algorithmically more efficient, as it reduces satisfiability and verification tasks to bounded searches over finite traces.

Minimal unsatisfiable cores (MUCs) (Lynce and Marques-Silva 2004) play a central role in explaining and debugging inconsistencies in specifications, notably in propositional and temporal logics. Recent LTL_f research has focused primarily on extracting MUCs from conjunctions of formulas, where each conjunct is treated as an indivisible unit (Roveri et al. 2024; Niu et al. 2023). While effective in many cases, this coarse-grained perspective fails to capture inconsistencies that arise from the internal structure of individual formulas. In contrast, the notion of syntax tree-based unsatisfiable cores (Schuppan 2012) offers a more fine-grained view, identifying precisely which subformula occurrences contribute to unsatisfiability. These tree-based approaches enable the identification of minimal sets of subformula occurrences that cause unsatisfiability, offering a more precise explanation of the conflict *within* arbitrary formulas, rather than just among sets of conjuncts.

In addition to minimal unsatisfiable cores, the dual concept of minimal correction sets (MCSes) is equally important for addressing inconsistencies (Janota and Marques-Silva 2016). A minimal correction set identifies the smallest set of subformulas whose removal or abstraction restores satisfiability of an otherwise inconsistent specification. In practical applications such as process mining, planning, and declarative workflow modeling, specifications are often expressed as LTL_f formulas encoding constraints over finite executions (Fuggitti and De Giacomo 2018), which frequently become inconsistent due to evolving policies or conflicting requirements. In these settings, identifying minimal unsatisfiable cores helps isolate precisely which subformula occurrences cause inconsistencies, while minimal correction sets indicate how specifications can be minimally repaired. By combining the identification of MUCs with MCSes, it becomes possible not only to explain the sources of inconsistency but also to guide effective resolution strategies.

Example 1 (Patient Monitoring)**.** *In a hospital ward, a monitoring protocol requires: (i) Each patient must receive at least one nurse bedside check-in (p) and one doctor update (q) in the electronic health record $\varphi_1 = \mathsf{F}\, p \wedge \mathsf{F}\, q$; (ii) After any nurse check-in, a doctor update must immediately occur $\varphi_2 = \mathsf{G}\,(p \rightarrow \mathsf{X}\, q)$; (iii) After any doctor update, a nurse check-in must immediately occur $\varphi_3 = \mathsf{G}\,(q \rightarrow \mathsf{X}\, p)$. The combination $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ is unsatisfiable over any finite patient records, as it forces an infinite alternation between nurse check-ins and doctor updates. Intuitively, the two subformulas $\mathsf{F}\, p \wedge \varphi_2 \wedge \varphi_3$ and $\mathsf{F}\, q \wedge \varphi_2 \wedge \varphi_3$ are two MUCs, since each selects different combinations of critical subformulas that together enforce the contradiction. Conversely, one possible MCS consists of abstracting both p and q in φ_1, thus avoiding the start of the cyclic dependencies.*

In this work, we introduce a technique to compute a refined notion unsatisfiable cores that is tree-based, where

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

abstracted subformulas are replaced with fresh propositional variables rather than fixed truth values as proposed in (Schuppan 2012). This yields a richer space where abstractions can vary across interpretations and trace states, enhancing flexibility and expressiveness. Our method works under bounded LTL_f semantics, where the evaluation is restricted to traces of bounded length (Latvala et al. 2004). In addition, we propose a novel Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011) encoding that represents bounded LTL_f satisfiability under arbitrary abstractions. Our method reduces the computation of syntax tree MUCs to the identification of minimal unsatisfiable subprograms and the extraction of MCSes to the enumeration of minimal answer sets, which can be efficiently computed using modern ASP solvers (Alviano et al. 2023). Experimental results demonstrate that our approach outperforms existing enumerative methods.

Related Work. Unsatisfiable cores (UCs) have been extensively studied in propositional logic and constraint solving (Liffiton and Sakallah 2008). For temporal logics, and in particular Linear Temporal Logic (LTL), the problem is more challenging due to the temporal structure of formulas. Schuppan (2012) introduced several notions of UCs for LTL, including syntax tree–based cores. Recent work has adapted UC computation to LTL over finite traces (LTL_f). Roveri et al. (2024) proposed algorithms leveraging satisfiability checking techniques, while Niu et al. (2023) studied the complexity of computing minimal unsatisfiable cores (MUCs). Both rely on the aaltaf solver (Li et al. 2020) and focus on formulas in conjunctive normal form. Other approaches have used Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Brewka, Eiter, and Truszczynski 2011). For example, there is an ASP-based technique for enumerating MUCs of LTL_f formulas (Ielo et al. 2024); and also MUC enumeration was proposed as a mean to quantify the degree of inconsistency of a specification (Kuhlmann and Corea 2024). These methods view inputs as conjunctions of formulas and identify UCs as subsets of conjuncts, which can miss inconsistencies within individual formulas.

Schuppan's syntax tree notion enables finer-grained identification of unsatisfiable cores in arbitrary LTL_f formulas. While his definition replaces abstracted subformulas with constants ($\top$ or $\bot$) based on polarity, our approach substitutes them with fresh propositional variables that can take different truth values across traces, yielding a more general notion. This interpretation aligns with the behavior of the LTL_f solver BLACK (Geatti et al. 2024), which also supports syntax tree core extraction.

Preliminaries

This section provides a brief overview of Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Brewka, Eiter, and Truszczynski 2011; Lifschitz 2019; Calimeri et al. 2020) and Linear Temporal Logic over Finite Traces (LTL_f) (De Giacomo and Vardi 2013).

Answer Set Programming. In ASP, a *term* is either a *variable* or a *constant*. Variables, following logic programming conventions, are alphanumeric strings starting with uppercase letters, whereas *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, $t_1, \ldots, t_n$ are terms, and n is known as *arity* of the predicate. An atom is *ground* if all its terms are constants. 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 opposite of l, $\bar{l} = a$ if $l = not\, a$, otherwise $\bar{l} = not\, a$. A *normal rule* r 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 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 $\{h : l_1, \ldots, l_k\} \leftarrow b_1, \ldots, b_n$, which is a shorthand for the set of normal rules $h_i \leftarrow l_1, \ldots, l_k, b_1, \ldots, b_n, not\, nh_i$; $nh_i \leftarrow l_1, \ldots, l_k, b_1, \ldots, b_n, not\, h_i$; where 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, given a program P, the Herbrand's base of P, denoted by $\mathcal{B}_P$, is the set of atoms constructible from constants and predicate names occurring in P. An *interpretation* I for P is a subset of $\mathcal{B}_P$, which is an answer set of P if (i) I is a model, i.e., 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 (ii) I is a minimal model of its GL-reduct (Gelfond and Lifschitz 1991).[1]

Consider a program P and a set of objective atoms $O \subseteq \mathcal{B}_P$. For $S \subseteq O$, 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, enforce(P, O, S) augments the program P in such a way 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 enforce(P, O, U) is incoherent (Alviano et al. 2023). $US(P, O)$ denotes the set of unsatisfiable subsets of P wrt O. An unsatisfiable subset $U \in US(P, O)$ is a *minimal unsatisfiable subset* (MUS) of P wrt O iff for every $U' \subset U, U' \notin US(P, O)$. An answer set M of P is minimal wrt $O \subset \mathcal{B}_\mathcal{P}$ iff there exist no other answer set M' of P such that $(M' \cap O) \subset (M' \cap O)$ (Alviano et al. 2023).

Linear Temporal Logic on Finite Traces. Linear Temporal Logic on Finite Traces (LTL_f) (De Giacomo and Vardi

[1]For more details we refer the reader to dedicated literature (Gelfond and Lifschitz 1991; Gebser et al. 2012).

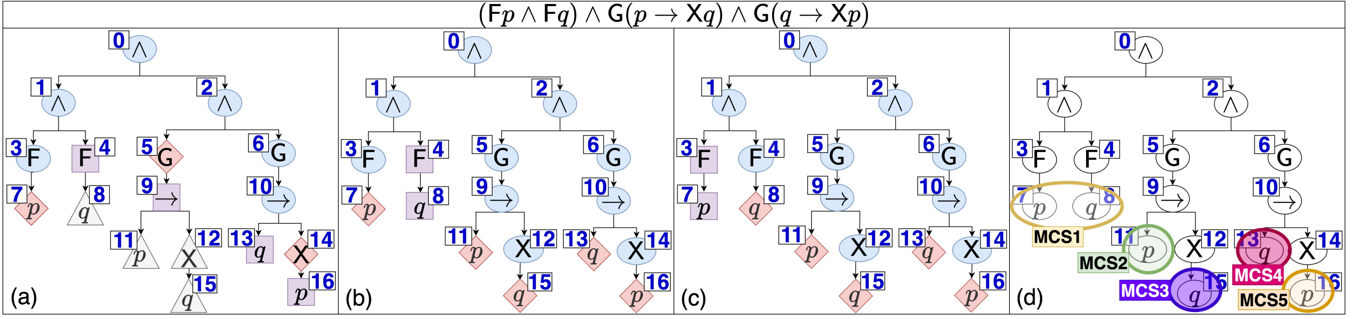

Figure 1: (a) Stumped ◯-shaped, abstracted □-shaped and detached △-shaped nodes corresponding to the set $X = \{7, 5, 14\}$ (highlighted in red ◇-shaped); (b)-(c) Minimal Unsatisfiable Cores; (d) Minimal Correction Sets.

2013) is a variant of Linear Temporal Logic (LTL) (Pnueli 1977) where the models, called *traces*, are finite sequences of states, as opposed to the infinite ones used in standard LTL. Let $\mathcal{A}$ be a finite set of atomic propositions. An LTL_f formula over $\mathcal{A}$ is defined by the following grammar:

$$\varphi ::= a \in \mathcal{A} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

where X ("Next") and U ("Until") are temporal operators. The following derived temporal operators are also commonly used: *Eventually* ($\mathsf{F}\ \varphi$): defined as $\top\ \mathsf{U}\ \varphi$; *Always* ($\mathsf{G}\ \varphi$) defined as $\neg\mathsf{F}\ \neg\varphi$; *Weak Next* ($\mathsf{X}_w\ \varphi$) defined as $\neg\mathsf{X}\ \neg\varphi$; and *Release* ($\varphi_1\ \mathsf{R}\ \varphi_2$) defined as $\neg(\neg\varphi_1\ \mathsf{U}\ \neg\varphi_2)$. Moreover, standard boolean abbreviations $\top$, $\bot$, $\vee$, $\rightarrow$, $\leftrightarrow$ also hold. An interpretation in LTL_f is a finite trace $\pi = \pi_0\pi_1 \cdots \pi_{n-1}$, where each $\pi_i \subseteq \mathcal{A}$ is the set of propositions that hold *true* at the i-th state. The trace length is denoted as $|\pi| = n$, and the i-th state as $\pi(i) = \pi_i$. The satisfaction relation $\pi, i \models \varphi$ is defined inductively as follows:

- $\pi, i \models a$ iff $a \in \pi(i)$
- $\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$
- $\pi, i \models \varphi \wedge \psi$ iff $\pi, i \models \varphi$ and $\pi, i \models \psi$
- $\pi, i \models \mathsf{X}\,\varphi$ iff $i + 1 < |\pi|$ and $\pi, i + 1 \models \varphi$
- $\pi, i \models \varphi\,\mathsf{U}\,\psi$ iff there exists k with $i \leq k < |\pi|$ such that $\pi, k \models \psi$ and for all j with $i \leq j < k$, $\pi, j \models \varphi$

A trace π is a *model* of φ if $\pi, 0 \models \varphi$, written $\pi \models \varphi$. The satisfiability problem for LTL_f, i.e., deciding whether there exists a trace π such that $\pi \models \varphi$, is known to be PSPACE-complete (De Giacomo and Vardi 2013). Complexity drops to NP-complete for bounded trace semantics, i.e. by enforcing an upper bound on trace length (Fionda and Greco 2018).

Syntax Tree-based Unsatisfiable Cores

We adopt a *syntactic perspective* to reason about the structure of LTL_f formulas, as proposed in (Schuppan 2012). Given an LTL_f formula φ, we define its *syntax tree*, denoted $\mathsf{Tree}(\varphi)$, as a rooted, ordered tree that reflects the syntactic derivation of φ according to the grammar of LTL_f and where each node represents a unique syntactic occurrence of a subformula of φ. Each node in $\mathsf{Tree}(\varphi)$ is uniquely identified by a non-negative integer i and is labeled with a syntactic construct derived from the grammar. In particular, leaf nodes are always labeled with propositional variables, and internal nodes are labeled with Boolean connectives or temporal operators. We assume, without loss of generality, that the root of the tree is assigned the identifier 0. Subformulas that appear multiple times in φ result in distinct subtrees, each corresponding to a uniquely numbered occurrence. We write ϕ_i to denote the subformula encoded by the subtree rooted at node i of $\mathsf{Tree}(\varphi)$–in the following, we may use ϕ as a shorthand for the formula encoded by the subtree rooted at node i, when the context is unambiguous.

Let φ be an LTL_f formula, and let $\mathsf{Tree}(\varphi)$ be its syntax tree. We define the *closure* of φ, denoted $\mathsf{cls}(\varphi)$, as the set of all subformula occurrences, that is, all distinct syntactic instances of subformulas within φ. Formally, each subformula occurrence corresponds to a unique node in the syntax tree, representing the root of the subtree that encodes that specific instance. In the following, we will refer to subformula occurrences of φ and nodes in its syntax tree interchangeably.

Example 2. *Consider the formula* $(\mathsf{F}\,p \wedge \mathsf{F}\,q) \wedge \mathsf{G}\,(p \rightarrow \mathsf{X}\,q) \wedge \mathsf{G}\,(q \rightarrow \mathsf{X}\,p)$*. Its syntax tree is reported in Figure 1 (a) and contains 17 nodes, each representing a subformula occurrence. Notably, nodes 7, 11, and 16 all correspond to the atomic subformula p, but they are distinct occurrences. In fact, they appear in different positions within the syntax tree and play different semantic roles within the formula.*

Let X be a subset of nodes of the syntax tree $\mathsf{Tree}(\varphi)$ of φ. To formalise the notion of syntax tree unsatisfiable core, we introduce the notion of *stump*.

Definition 1 (Stump). *Let φ be an LTL_f formula, and $X \subseteq cls(\varphi)$. The X-stump of φ, denoted by $\mathsf{Stump}(X, \varphi)$, is the subtree of $\mathsf{Tree}(\varphi)$ that contains all the nodes in X, all the ancestors of each node in X, and the edges connecting them.*

In general, $\mathsf{Stump}(X, \varphi)$ is not a syntactically valid LTL_f formula. We say that a node x of $\mathsf{Stump}(X, \varphi)$ is *void* if x has fewer children nodes in $\mathsf{Stump}(X, \varphi)$ than in the full $\mathsf{Tree}(\varphi)$. Formally, we define $gap(x) = \{y \in Nodes(\mathsf{Tree}(\varphi)) : (x, y) \in \mathsf{Tree}(\varphi), (x, y) \notin \mathsf{Stump}(X, \varphi)\}$.

Definition 2 (Anchor). *The X-anchor of φ is a LTL_f formula obtained by appending fresh propositional symbols to void nodes of $\mathsf{Stump}(X, \varphi)$.*

More precisely, for each node $x \in \mathsf{Stump}(X, \varphi)$ and each node $y \in gap(x)$, we append a node labeled by a fresh propositional symbol o_y to x. Each such y is referred to as an *abstracted node*. If x is an abstracted node, we refer to nodes in $cls(x) \setminus \{x\}$ as *detached nodes*.

Example 3. *Consider the formula* $(\mathsf{F}\, p \wedge \mathsf{F}\, q) \wedge \mathsf{G}\,(p \rightarrow \mathsf{X}\, q) \wedge \mathsf{G}\,(q \rightarrow \mathsf{X}\, p)$*. Figure 1 (a) depicts its parse tree. Let* $X = \{5, 7, 14\}$ *that corresponds to the formulas* $\{p, \mathsf{G}\,(p \rightarrow \mathsf{X}\, q), \mathsf{X}\, p\}$*. The corresponding X-anchor would be the formula* $(\mathsf{F}\, p \wedge o_{\mathsf{F}\, q}) \wedge \mathsf{G}\,(o_{p \rightarrow \mathsf{X}\, q}) \wedge \mathsf{G}\,(o_q \rightarrow \mathsf{X}\, o_p)$*, where* o_ϕ *abstracts the subformula* ϕ*.*

Informally, the X-anchor captures the idea of "keeping the syntax tree untouched up to the nodes in X", while abstracting away everything beyond them. Notably, when X corresponds to the set of all leaves of φ, the X-anchor is φ itself. Our framework interprets unsatisfiable cores as subsets of $cls(\varphi)$ whose anchor yield an unsatisfiable formula.

Definition 3 (Tree Unsatisfiable Core)**.** *Let φ be an unsatisfiable LTL$_f$ formula. We say that $U \subset cls(\varphi)$ is an* unsatisfiable core *if the U-anchor of φ is unsatisfiable. An unsatisfiable core is* minimal *if all its proper subsets are not unsatisfiable cores.*

Example 4. *Consider again the formula* $(\mathsf{F}\, p \wedge \mathsf{F}\, q) \wedge \mathsf{G}\,(p \rightarrow \mathsf{X}\, q) \wedge \mathsf{G}\,(q \rightarrow \mathsf{X}\, p)$*. The set* $X = \{7, 11, 13, 15, 16\}$ *(see Figure 1 (b)) is an unsatisfiable core. Indeed, its X-anchor, i.e., the formula* $(\mathsf{F}\, p \wedge o_{\mathsf{F}\, q}) \wedge \mathsf{G}\,(p \rightarrow \mathsf{X}\, q) \wedge \mathsf{G}\,(q \rightarrow \mathsf{X}\, p)$*, is unsatisfiable. Moreover, X is minimal, since every proper subset X' yields a satisfiable anchor formula. Note that the set* $X = \{8, 11, 13, 15, 16\}$ *is also a minimal unsatisfiable core (see Figure 1 (c)).*

Characterizing syntax tree unsatisfiable cores as sets of anchored subformulas also naturally leads to a simple definition of their dual notion, the correction set.

Definition 4 (Tree Correction Sets)**.** *Let φ be an unsatisfiable LTL$_f$ formula. We say that $S \subset cls(\varphi)$ is a* correction set *if anchoring the complement of their closures* $cls(\varphi) \setminus \bigcup_{s \in S} cls(s)$ *yields a satisfiable formula. A correction set S is* minimal *if no strict subset of its closure is itself a correction set; formally, there is no* $S' \subset \bigcup_{s \in S} cls(s)$ *such that S' is a correction set.*

Intuitively, this corresponds to anchoring as much of the original formula as possible while restoring satisfiability.

Example 5. *Consider again the formula* $(\mathsf{F}\, p \wedge \mathsf{F}\, q) \wedge \mathsf{G}\,(p \rightarrow \mathsf{X}\, q) \wedge \mathsf{G}\,(q \rightarrow \mathsf{X}\, p)$*. Figure 1 (d) shows the minimal correction sets, which correspond to the hitting sets of the two minimal unsatisfiable cores shown in Figure 1 (b) and (c).*

Searching Tree MUCs with ASP

We propose an ASP-based approach for computing (minimal) unsatisfiable syntax tree cores of LTL$_f$ formulas. Our solution extends the encoding proposed in (Fionda, Ielo, and Ricca 2024) to handle the notion of anchors during model search. We define a logic program whose answer sets represent satisfiable anchors of a formula φ, and use its minimal unsatisfiable subprograms to identify anchors that yield unsatisfiable subformulas, i.e., syntax tree unsatisfiable cores.

In the remainder of this section, φ denotes an (unsatisfiable) LTL$_f$ formula and k is a positive integer representing the horizon in bounded semantics. The encoding is written in the input language of the ASP system `clingo` (Gebser et al. 2019). For background on `clingo`, we refer the reader to standard references (Gebser et al. 2012) and the online `clingo` user guide.

LTL$_f$ Bounded Satisfiability

We briefly recap the approach of Fionda et. al., which provides *(i)* a uniform encoding of LTL$_f$ formulas as sets of ASP facts, and *(ii)* a logic program $P_{\text{LTL}_f} \cup P_{search}$ that encodes k-bounded satisfiability (Fionda, Ielo, and Ricca 2024). The input formula φ is translated into a set of facts by mapping the nodes of its syntax tree to atoms, where the predicate name matches the label of the node and terms match the unique identifier of the node and of its children. We denote by $[\varphi]$ the set of facts (called reification) that encode φ.

Example 6. *Consider the formula* $\varphi = \mathsf{G}\,(a) \wedge \mathsf{F}\,(\neg a)$*. It is encoded by means of the following facts*[2]*:*

```
root(5). atom(0,a). atom(2,a). eventually(1,0).
negate(3,2). always(4,3). conjunction(5,4,1).
```

The logic program P^k_{search} generates candidate traces of length at most k by means of choice rules.

```
#const k.
sym(A) :- atom(_,A).

{ time(T): T=0..k-1 }.
time(T-1) :- time(T), T > 0.

{ trace(T,A): sym(A) } :- time(T).
last_state(T) :- time(T), not time(T+1).
:- root(X), not holds(0,X).
```

The first choice rule non-deterministically selects the trace length (up to the bound k), and the predicate $time/1$ models available time-points, enforcing no gaps between available time points. The predicate `sym/1` collects all propositional symbols that appear in the input formula, and the second choice rule expresses traces as sequences over the available propositional symbols, thereby constructing candidate traces. In particular, the atom $trace(t, a)$ models that $a \in \pi(t)$. The atom $holds(t, x)$ models that $\pi, t \models \varphi_x$, where π is the guessed trace. The constraint discards guessed traces that are not models of the input formula.

The $holds/2$ predicate captures the semantics of LTL$_f$ operators, and is defined in the program P_{LTL_f}. In detail, P_{LTL_f} encodes the semantics of temporal and Boolean operators using normal rules, as follows:

```
holds(T,X) :- atom(X,A), trace(T,A).

holds(T,X) :- until(X,LHS,RHS),
  holds(T,LHS), holds(T+1,X).

holds(T,X) :- until(X,LHS,RHS),
  holds(T, RHS), time(T).
```

[2]Subformulas that appear twice share their index in (Fionda, Ielo, and Ricca 2024), while here the formula is a plain tree.

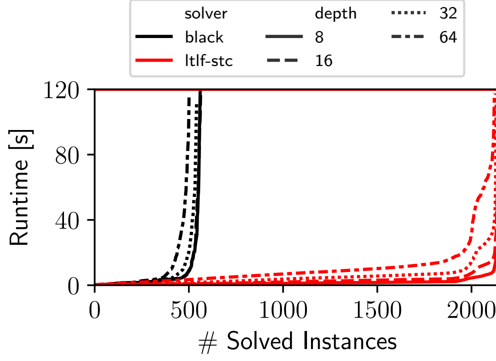

Figure 2: Computing a MUC. Cumulative runtime comparison between BLACK and tool. A point (x, y) denotes that there exist x instances that can be solved in up to y seconds.

```
holds(T,X) :- next(X,F),
  holds(T+1,F), time(T).
holds(T,X) :- conjunction(X,A,B),
  holds(T,A), holds(T,B).
holds(T,X) :- negate(X,F), time(T),
  not holds(T,F).
```

The models of the program $P_{\mathrm{LTL_f}} \cup [\varphi] \cup P_{search}$ are in one-to-one correspondence with traces of length at most k that satisfy φ (Fionda, Ielo, and Ricca 2024). If the program has no stable models, then the formula φ is unsatisfiable under k-bound semantics.

Modeling anchors. Now we manipulate the reification of $[\varphi]$, such that sources of unsatisfiability of $P_{\mathrm{LTL_f}} \cup [\varphi] \cup P_{search}$ (that is, its minimal unsatisfiable subprograms) yield the syntax tree unsatisfiable cores of φ. For each edge (i, j) in the syntax tree, we add (to the reification of φ) a fact $tree(i, j)$, and for each node in the syntax tree, we add a fact $subformula(i)$. Let α_1, α_2 be placeholders for the predicate names of respectively unary and binary operators. We replace facts of the form $\alpha_1(i, f) \in [\varphi]$ with rules $\alpha_1(i, f) \leftarrow fix(i)$, and facts of the form $\alpha_2(i, lhs, rhs) \leftarrow fix(i)$, where $fix/1$ will be used as *objective atoms* for MUS computation, and consequently are under free choice.

Example 7. *The facts of Example 6 are replaced by:*

```
atom(0,a) :- fix(0).
atom(2,a) :- fix(2).
eventually(1,0) :- fix(1).
tree(1,0).
negate(3,2) :- fix(3).
tree(3,2).
always(4,3) :- fix(4).
tree(4,3).
conjunction(5,4,1) :- fix(5).
tree(5,4). tree(4,1).
{ fix(X) } :- subformula(X).
```

Following the Definition 1, we model the concept of stump of φ wrt the set of nodes identified by the predicate $fix/1$. We define it as the ancestors of fixed nodes. We define abstracted nodes, $abs/1$ predicate, as children of stumps that are not stumps themselves.

Family	#	Solved Instances							
		BLACK				ASP			
		8	16	32	64	8	16	32	64
acacia	11	11	11	11	10	11	11	11	11
alaska	129	16	16	17	16	129	128	120	114
anzu	70	34	34	34	34	70	70	70	68
forobots	38	0	0	0	0	38	38	38	38
random	935	142	141	136	125	935	935	935	935
rozier	147	71	73	71	70	147	147	147	147
schuppan	67	38	38	38	38	61	59	58	57
trp	753	250	250	235	209	753	753	753	753

Table 1: Computing a MUC. Solved instances for BLACK and ASP for different bounds $k \in \{8, 16, 32, 64\}$

```
stump(X) :- fix(X).
stump(Y) :- stump(X), tree(X,Y).
abs(Y) :- stump(X), tree(X,Y), not stump(Y).
```

We also assume, by convention, that if the stump is empty, then we are abstracting the root node of the tree:

```
abs(X) :- root(X), not stump(_).
```

Finally, to apply the "replacement" of abstracted nodes with fresh propositional variables, we introduce the last rule:

```
atom(X, o(X)) :- abs(X).
```

Standard reification occurs when a node is stumped. Otherwise, the reification atom is replaced by a fresh propositional symbol, i.e., the node is abstracted. In cases where the node is neither stumped nor abstracted, the atom is 'detached' and simply ignored. We denote the reification of φ modified as described above as $[\varphi]^*$. Each answer set M of $P_{\mathrm{LTL_f}} \cup [\varphi]^* \cup P_{search}$ can be decoded into a (X, π) pair where anchoring $X = \{x : fix(x) \in M\}$ yields a satisfiable formula that admits π as a model under k-bound semantics. Thus, the minimal unsatisfiable subprograms wrt $fix/1$ provide syntax tree minimal unsatisfiable cores of φ. Consequently, due to the duality relationship between MUSes and minimal answer sets (Alviano et al. 2023), minimal answer sets wrt $fix/1$ enables us to compute syntax tree minimal correction sets.

Experiments

In this section, we present the results of an experimental evaluation of our approach for computing syntax tree–based minimal unsatisfiable cores (MUCs) and minimal correction sets (MCSes) under k-bounded semantics. Experiments were executed on an Intel(R) Xeon(R) CPU E7-8880 v4 @ 2.20 GHz server with 500 GB RAM and 88 physical cores, running up to 32 concurrent jobs via GNU Parallel. Each run was limited to a timeout of 120 CPU seconds. We used the MUS enumerator based on `wasp` (Alviano et al. 2023) (version 2.0) in combination with the grounder `gringo` (Gebser, Schaub, and Thiele 2007) (version 5.4.1).

Code to reproduce our experiments and prototype are available at https://www.github.com/ainnoot/ltlf-stc and https://osf.io/hb36x.

We consider a benchmark suite of 2150 unsatisfiable formulas widely adopted in the $\mathrm{LTL_f}$ literature (Li et al. 2014;

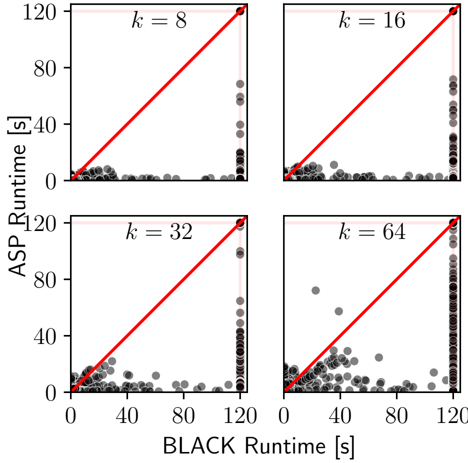

Figure 3: Computing a MUC. Instance-wise comparison between BLACK and `ltlf-stc`. A point (x, y) denotes an instance where BLACK yields the first MUC in x seconds, and our tool in y seconds.

k	Family	MUCs μ	MUCs σ	MCSes μ	MCSes σ
8	random	0.892	0.638	3.070	10.185
	acacia	0.427	0.074	1.654	1.052
	alaska	5.441	9.991	5.169	8.057
	anzu	11.083	17.557	6.180	4.802
	forobots	0.516	0.033	1.141	0.046
	rozier	0.335	0.065	1.138	4.368
	schuppan	2.096	5.474	3.367	9.514
	trp	1.186	1.217	1.552	1.479
16	random	4.036	9.921	4.007	9.544
	acacia	0.785	0.406	1.847	0.982
	alaska	17.672	27.695	9.292	13.512
	anzu	26.661	27.207	7.912	5.150
	forobots	0.676	0.055	1.838	0.060
	rozier	0.383	0.102	2.176	10.452
	schuppan	3.695	9.573	3.696	9.733
	trp	9.206	18.303	2.390	2.999
32	random	7.750	15.455	5.759	7.934
	acacia	2.574	2.267	2.488	1.074
	alaska	18.135	28.984	13.682	18.205
	anzu	39.511	31.889	12.828	8.780
	forobots	1.091	0.083	3.356	0.211
	rozier	0.510	0.212	2.215	8.276
	schuppan	7.084	20.796	4.573	9.476
	trp	16.825	25.407	4.551	7.619
64	random	12.223	18.079	9.647	9.608
	acacia	16.785	16.872	5.117	2.532
	alaska	13.573	21.410	24.688	25.284
	anzu	48.160	19.838	26.209	18.274
	forobots	2.416	0.075	7.548	0.885
	rozier	0.793	0.504	2.889	8.981
	schuppan	5.374	12.490	8.266	18.155
	trp	17.946	27.901	9.274	15.528

Table 2: Average (μ) and standard deviation (σ) of time required to compute a single MUC and a single MCS for different families and bounds.

Schuppan and Darmawan 2011). These formulas have also been used in benchmarking minimal unsatisfiable core extraction methods for conjunctive specifications (Roveri et al. 2024; Niu et al. 2023; Ielo et al. 2024). The considered formulas (i.e., those from (Li et al. 2014)) also include specifications of Declare patterns (Pesic, Schonenberg, and van der Aalst 2007), a major applications of LTL_f to business process modeling (Di Ciccio and Montali 2022). We analyse the results in three tasks: single MUC computation, single MCS computation, MUC enumeration, and MCS enumeration. In the first task, our approach, labelled `ltlf-stc`, is compared to BLACK (Geatti, Gigante, and Montanari 2019), patched to support k-bounded semantics (by interpreting `UNKNOWN` output state as `UNSATISFIABLE` upon reaching depth k during the tableaux expansion).

Computing a single MUC and MCS

We start by evaluating the effectiveness of our approach, comparing it against the BLACK solver. BLACK computes cores by enumeratively checking subsets of subformulas from the input specification, thus representing a natural baseline for the task.

Comparison with BLACK. Table 1 shows the number of solved instances (that is, formulas for which a single MUC was successfully computed within the given timeout) for each method for increasing values of the bound k. Our ASP-based approach consistently solves significantly more instances than BLACK. This is more evident in Figure 2, which reports the cumulative execution time needed by each system to extract one MUC across all tested formulas. The instance-wise runtime comparison reported in Figure 3 confirms that our method outperforms BLACK.

Single MUC and MCS runtimes. Table 2 reports, for different formula families, the average (μ) and standard deviation (σ) of the runtime required to compute a single MUC and a single MCS for various values of the bound k. Overall, we observe that runtime increases with higher values of k, as expected, although the behavior differs significantly across families. In particular, for challenging families such as `anzu` and `alaska`, the variability (σ) remains consistently large, indicating substantial differences in individual instance difficulty. Conversely, families such as `forobots` and `rozier` exhibit lower runtimes and significantly smaller standard deviations, highlighting their relative ease and stability across instances.

Impact of Formula and Alphabet Size. We now investigate how key factors influencing the search space size affect runtime performance. Figure 4 compares the runtime of our approach against the BLACK solver as a function of formula size $|\varphi|$ (top row) and alphabet size $|\mathcal{A}|$ (bottom row) on a single MUC computation. We report the results for the formula families `anzu`, `alaska`, `random`, and `trp` for $k = 64$, which show paradigmatic behaviour. As expected, increasing values for $|\varphi|$ and $|\mathcal{A}|$ generally result in higher runtimes for both systems.

In the task of computing a single minimal correction set (MCS) at the fixed depth $k = 64$ we show results for the same four representative formula families. The upper plots show the influence of formula size, whereas the lower plots assess the alphabet size impact. While a general correlation between increased formula and alphabet size and higher runtimes is observed (particularly evident in the `random` fam-

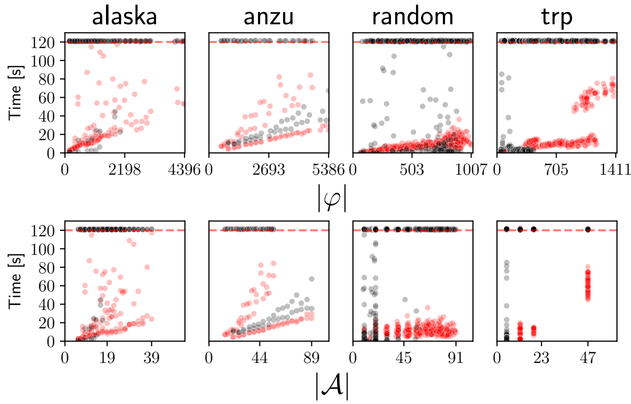

Figure 4: Computing a MUC. Runtime comparison between BLACK (in black) and our tool (in red), at depth $k = 64$. Top: impact of formula size; bot: impact of alphabet size.

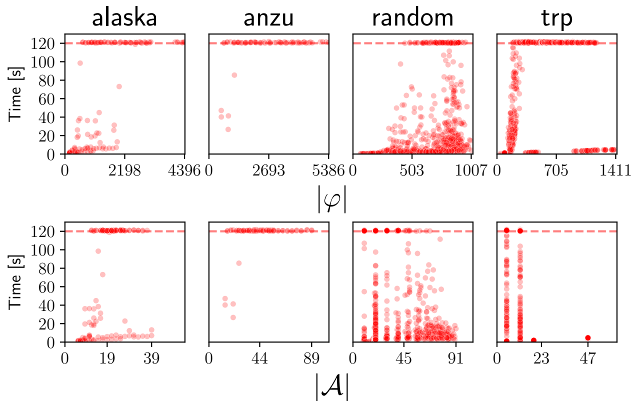

Figure 5: Computing a MCS. Runtime of our tool, partitioned among families at depth $k = 64$. Top: impact of formula size; bot: impact of alphabet size.

ily), it is noteworthy that some formulas from families such as `alaska` and `anzu` exhibit high runtimes or timeouts even at relatively small sizes. This behavior suggests that intrinsic semantic complexity and internal structure of the formulas substantially contribute to their difficulty, beyond straightforward metrics like formula or alphabet size alone.

Enumerating MUCs and MCSes

Enumerating MUCs can be achieved by leveraging algorithms for enumerating minimal unsatisfiable subsets, rather than stopping after identifying a single solution, as in previous experiments. Analogously, enumerating minimal correction sets corresponds to computing maximal stable models of the logic program wrt the objective atoms `fix/1`.

Table 3 presents quantile-based statistics on the number of MUCs and MCSes enumerated within a timeout of 120 seconds, for varying bounds $k \in \{8, 16, 32, 64\}$. As expected, higher values of k generally result in increased computational difficulty, leading to fewer enumerated MUCs and MCSes within the same timeout. Nevertheless, our results

k	Family	MUCs $q_{0.25}$	MUCs $q_{0.50}$	MUCs $q_{0.75}$	MUCs $q_{0.90}$	MCSes $q_{0.25}$	MCSes $q_{0.50}$	MCSes $q_{0.75}$	MCSes $q_{0.90}$
8	random	2888	3922	4805	6790	194	266	340	488
	acacia	1250	7525	10108	11258	102	142	166	170
	alaska	598	1184	2048	3160	37	68	134	316
	anzu	195	694	1503	2682	147	224	316	415
	forobots	3	3	3	3	1	1	1	1
	rozier	3	36	289	9389	2	87	2233	3652
	schuppan	2	12	290	1807	1	1	1	65
	trp	2401	5472	8914	11252	106	212	441	688
16	random	2040	3371	4080	5412	97	125	157	213
	acacia	1250	4522	6276	7459	99	136	172	192
	alaska	6	101	848	1470	11	38	133	181
	anzu	0	0	264	1282	41	88	147	207
	forobots	3	3	3	3	1	1	1	1
	rozier	3	15	2802	5167	2	10	816	1428
	schuppan	2	8	174	1190	1	1	1	44
	trp	1452	3110	5098	6309	90	184	433	592
32	random	803	2312	3137	3950	44	59	75	103
	acacia	1004	2475	3373	3781	98	107	138	173
	alaska	0	2	28	353	0	7	39	128
	anzu	0	0	0	47	0	0	50	92
	forobots	3	3	3	3	1	1	1	1
	rozier	2	4	59	895	2	2	122	341
	schuppan	2	4	115	1152	1	1	1	26
	trp	0	1146	2245	2949	73	169	322	393
64	random	61	1048	2142	2595	16	20	34	44
	acacia	6	125	839	1333	98	104	124	131
	alaska	0	0	3	35	0	0	0	33
	anzu	0	0	0	0	0	0	0	24
	forobots	3	3	3	3	1	1	1	1
	rozier	0	3	9	17	2	2	54	135
	schuppan	2	2	43	176	1	1	1	11
	trp	0	0	441	780	46	108	196	214

Table 3: MUCs and MCSes quantiles over different formula families and depths $k \in \{8, 16, 32, 64\}$.

confirm that a substantial number of MUCs and MCSes can be consistently enumerated for most formula families, even at higher reasoning depths. The table highlights notable variability among formula families.

The `random` and `acacia` families consistently yield large numbers of enumerated MUCs and MCSes across all quantiles. In contrast, families such as `anzu` and `alaska` become significantly more challenging as k increases, drastically reducing the number of enumerated solutions. Furthermore, the `forobots` family consistently yields a minimal number of solutions across all depths, suggesting an inherent limitation in the number of unsatisfiable cores and correction sets of these formulas. We observe also that enumeration performance depends on the semantic structure and intrinsic complexity of each formula family.

Conclusion and Future Work

In this paper, we introduced a novel approach to computing syntax tree-based minimal unsatisfiable cores (MUCs) and minimal correction sets (MCSes) for Linear Temporal Logic over finite traces (LTL_f). Our method is grounded in an ASP encoding that models bounded LTL_f satisfiability. Our experimental evaluation clearly demonstrates that the ASP-based approach can handle substantially large and complex formulas and significantly outperforms baseline methods.

Future research directions include integrating our core extraction approach to develop interactive debugging and model repair tools (Corea et al. 2021) for declarative specifications written in LTL_f.

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, and Tech4You CUP H23C22000370006; and European Union – Next Generation EU through the MUR PRIN 2022-PNRR project DISTORT (CUP: H53D23008170001) under the Italian PNRR Mission 4 Component 1.

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.

Bacchus, F.; and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22: 5–27.

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

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.

Corea, C.; Kuhlmann, I.; Thimm, M.; and Grant, J. 2024. Paraconsistent reasoning for inconsistency measurement in declarative process specifications. *Inf. Syst.*, 122: 102347.

Corea, C.; Nagel, S.; Mendling, J.; and Delfmann, P. 2021. Interactive and Minimal Repair of Declarative Process Models. In *BPM (Forum)*, volume 427 of *Lecture Notes in Business Information Processing*, 3–19. Springer.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, 2013*, 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.

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.

Fuggitti, F.; and De Giacomo, G. 2018. *LTL and past LTL on finite traces for planning and declarative process mining*. Ph.D. thesis, Master's thesis, DIAG, Sapienza Univ. Rome.

Geatti, L.; Gigante, N.; and Montanari, A. 2019. A SAT-Based Encoding of the One-Pass and Tree-Shaped Tableau System for LTL. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings*, volume 11714 of *Lecture Notes in Computer Science*, 3–20. Springer.

Geatti, L.; Gigante, N.; Montanari, A.; and Venturato, G. 2024. SAT Meets Tableaux for Linear Temporal Logic Satisfiability. *J. Autom. Reason.*, 68(2): 6.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1): 27–82.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. GrinGo : A New Grounder for Answer Set Programming. In Baral, C.; Brewka, G.; and Schlipf, J. S., eds., *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, 266–271. Springer.

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

Ielo, A.; Mazzotta, G.; Peñaloza, R.; and Ricca, F. 2024. Enumerating Minimal Unsatisfiable Cores of LTLf formulas. *CoRR*, abs/2409.09485.

Janota, M.; and Marques-Silva, J. 2016. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233: 73–83.

Kuhlmann, I.; and Corea, C. 2024. Inconsistency Measurement in LTL_f Based on Minimal Inconsistent Sets and Minimal Correction Sets. In *SUM*, volume 15350 of *Lecture Notes in Computer Science*, 217–232. Springer.

Latvala, T.; Biere, A.; Heljanko, K.; and Junttila, T. 2004. Simple bounded LTL model checking. In *International Conference on Formal Methods in Computer-Aided Design*, 186–200. 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.

Li, J.; Zhang, L.; Pu, G.; Vardi, M. Y.; and He, J. 2014. LTLf Satisfiability Checking. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 513–518. IOS Press.

Liffiton, M. H.; and Sakallah, K. A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40: 1–33.

Lifschitz, V. 2019. *Answer Set Programming*. Springer. ISBN 978-3-030-24657-0.

Lynce, I.; and Marques-Silva, J. 2004. On Computing Minimum Unsatisfiable Cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*.

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.

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 *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, 46–57. IEEE Computer Society.

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

Schuppan, V. 2012. Towards a notion of unsatisfiable and unrealizable cores for LTL. *Science of Computer Programming*, 77(7-8): 908–939.

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