Towards ASP-based Minimal Unsatisfiable Cores Enumeration for LTLf

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

[1]DeMaCS, University of Calabria, Italy
[2]DISCo, University of Milano-Bicocca, Italy

Abstract

Linear Temporal Logic over Finite Traces (LTL_f) is a widely used formalism with applications in Artificial Intelligence (AI), process mining, model checking, and more. The primary reasoning task for LTL_f is satisfiability checking. However, the recent focus on explainable AI has increased interest in analyzing inconsistent formulas, making the enumeration of minimal explanations for infeasibility a relevant task for LTL_f. This paper introduces a novel technique for enumerating minimal unsatisfiable cores 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 of the ASP program directly correspond to the minimal unsatisfiable cores of the original LTL_f specification. Leveraging recent advancements in ASP solving yields a minimal unsatisfiable cores enumerator achieving good performance in experiments conducted on established benchmarks from the literature.

Keywords

Answer Set Programming, Linear Temporal Logic over Finite Traces, Minimal Unsatisfiable Cores

1. Introduction

Linear temporal logic over Finite Traces (LTL_f) [1] is a simple, yet powerful language for expressing and reasoning about temporal specifications, that is known to be particularly well-suited for applications in Artificial Intelligence (AI) [2, 3, 4, 5].

Perhaps its most widely recognized use to-date is as the logic underlying temporal process modeling languages such as Declare [6]. Very briefly, a Declare specification is a set of constraints on the potential evolution of a process, which is expressed through a syntactic variant of a subclass of LTL_f formulas. The full specification can thus be seen as a conjunction of LTL_f formulas. As specifications become bigger—especially when they are automatically mined from event logs [7]—, it is not uncommon to encounter inconsistencies (i.e., business process models that are intrinsically contradictory) or other errors.

To understand and correct these errors, it is thus important to highlight the sets of formulas in the specification that are responsible for them [8, 9]. Specifically, we are interested in computing the *minimal unsatisfiable cores* (MUCs): subset-minimal sets of formulas (from the original specification) that are collectively inconsistent [10, 8, 9]. 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 [11, 12]). Thus, a system capable of efficiently enumerating MUCs would be of significant value.

A similar problem has been studied in the field of answer set programming (ASP) [13, 14], where the goal is to find *minimal unsatisfiable subsets* (MUSes) of atoms that make an ASP program incoherent [15, 16, 17]. In recent years, efficient implementations of MUS enumerators have been presented [17].

Our goal in this paper is to take advantage of both ASP declarativity and ASP systems efficiency to enumerate MUCs of LTL_f formulas. Hence, we present a new transformation that constructs, given a

OVERLAY 24: 6th International Workshop on Artificial Intelligence and fOrmal VERification, Logic, Automata, and sYnthesis
antonio.ielo@unical.it (A. Ielo); giuseppe.mazzotta@unical.it (G. Mazzotta); francesco.ricca@unical.it (F. Ricca); rafael.penaloza@unimib.it (R. Peñaloza)
0009-0006-9644-7975 (A. Ielo); 0000-0003-0125-0477 (G. Mazzotta); 0000-0001-8218-3178 (F. Ricca); 0000-0002-2693-5790 (R. Peñaloza)

set of LTL_f formulas, an ASP program whose MUSes are in a bijection with the MUCs of the original specification. Importantly, although we base our reduction on a well-known encoding of LTL_f bounded satisfiability [18, 19], the idea is general enough; it can be applied to other decision procedures, as long as they can be expressed in ASP. To improve its efficiency, our enumerator checks for unsatisfiability iteratively by considering traces of increasing length based on a progression strategy [20]. To the best of our knowledge, we provide the first MUC enumerator for LTL_f.

We empirically compared our implementation with the domain-agnostic MUC enumeration tool `must` [21][1]. Our results suggests ASP is a promising solution for the LTL_f MUCs enumeration task.

Related works. The task of computing MUCs has been considered, under different names, for several representation languages including propositional logic [22], constraint satisfaction problems [23], databases [24], description logics [25], and ASP [17] among many others. For a general overview of the task and known approaches to solve it, see [26].

Although the task was briefly studied for LTL (over infinite traces) in [27], it was only recently considered for the specific case of LTL_f [8, 9]. 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-fletched 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 [28] and later on using ASP solvers [29]. 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 [30], alongside an incremental approach that verifies the (non-)existence of models up to a certain length [18].

2. Preliminaries

We assume the reader to be familiar with syntax and semantics of Linear Temporal Logic over Finite Traces (LTL_f) [1]. In the rest of the paper, we assume all LTL_f formulae to be in conjunctive form, e.g. $\varphi = \bigwedge \phi_i$ for some set of LTL_f formulae $\{\phi_1, \ldots, \phi_k\}$. With a slight abuse of notation we refer to a formula in conjunctive form as the set of its conjunts; thus, for example, given $\varphi = \phi_1 \wedge \phi_2 \wedge \phi_3$, the subformula $\psi = \phi_1 \wedge \phi_3$ is denoted by the set $\{\phi_1, \phi_3\} \subseteq \{\phi_1, \phi_2, \phi_3\}$. Recall that given an unsatisfiable LTL_f formula $\varphi = \bigwedge \phi_i$ in conjunctive form, a *minimal unsatisfiable core* (MUC) of φ is an unsatisfiable formula $\psi \subseteq \varphi$ which is minimal (w.r.t. set inclusion); i.e., removing any conjunct from ψ yields a satisfiable formula [8]. Complexity-wise, it is known that a single formula may have exponentially many MUCs, but computing one MUC requires only polynomial space; just as deciding satisfiability [31, 26].

The next subsection recaps required notions of Answer Set Programming (ASP) [13]. We assume familiarity with ASP. The interested reader can refer to [32, 33] for an introduction to these notions.

2.1. Answer Set Programming

Syntax and semantics. A *term* is either a *variable* or a *constant*, where *variables* are alphanumeric strings starting with uppercase letter, while *constants* are either integer numbers or alphanumeric strings starting with lowercase letter. 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

[1] https://github.com/jar-ben/mustool

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 use also choice rules, which abbreviate complex expressions [32]. 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 $h_i \leftarrow l_1^i, \ldots, l_{k_i}^i, b_1, \ldots, b_n, not\, nh_i$; $nh_i \leftarrow l_1^i, \ldots, l_{k_i}^i, b_1, \ldots, b_n, not\, h_i$, for each $i \in 1, \ldots, m$ where e_i are of the form $h_i : l_1^i, \ldots, l_{k_i}^i$ and nh_i is a fresh atom not appearing anywhere else.

Given a program P, the *Herbrand Universe* of P, $\mathcal{U}_P$, denotes the set of constants that appear in P, while the *Herbrand Base*, $\mathcal{B}_P$, denotes the set of ground atoms obtained from predicates in P and constants in $\mathcal{U}_P$. Given a program P, and $r \in P$, $ground(r)$ denotes the set of ground instantiations of r obtained by replacing variables in r with constants in $\mathcal{U}_P$. Given a program P, $ground(P)$ denotes the union of ground instantiations of rules in P. An *interpretation* $I \subseteq \mathcal{B}_P$ is a set of atoms. Given an interpretation I, a positive (resp. negative) literal l is true w.r.t. I if $l \in I$ (resp. $\bar{l} \notin I$); otherwise it is false. A conjunction of literal is true w.r.t. I if all its literals are true w.r.t. I. An interpretation I is a *model* of P if for every rule $r \in ground(P)$, H_r is true whenever B_r is true. Given a program P and an interpretation I, the (Gelfond-Lifschitz) reduct [14], denoted by P^I, is defined as the set of rules obtained from $ground(P)$ by deleting those rules whose body is false w.r.t. I and removing all negative literals that are true w.r.t. I from the body of remaining rules. Given a program P, and a model I, then I is also an *answer set* of P if no such $I' \subseteq I$ exists such that I' is a model of P^I. For a program P, let AS(P) denotes the set of answer sets of P, then P is said to be *coherent* if $AS \neq \emptyset$, otherwise it is *incoherent*.

MUSes and MSMs [17]. Consider a program P and a set of objective atoms $O \subseteq \mathcal{B}_P$. For $S \subseteq O$, we denote by enforce(P, O, S) 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) denotes an augmentation of the program P in which the objective atoms can be arbitrarily choosen (i.e. either as true or false) but the atoms in S are *enforced* to be true.

An *unsatisfiable subset* for P w.r.t. the set of objective atoms O is a set of atoms $U \subseteq O$ such that enforce(P, O, U) is incoherent. US(P, O) denotes the set of unsatisfiable subsets of P w.r.t. O. An unsastisfiable subset $U \in$ US(P, O) is a *minimal unsatisfiable subset* (MUS) of P w.r.t. O iff for every $U' \subset U$, $U' \notin$ US(P, O). Analogously, an answet set $M \in$ AS(P) is a *minimal stable model* (MSM) of P w.r.t. the set of objective atoms O if there is no answer set $M' \in$ AS(P) with $(M' \cap O) \subset (M \cap O)$.

3. Method

Our idea, inspired by and most closely related to domain agnostic approaches to MUC enumeration developed in the SAT community [21], is to leverage ASP minimal unsatisfiable subprograms enumeration techniques to enumerate LTL_f formulae MUCs.

In particular, our starting point is the bounded satisfiability approach described in [19]. Given an LTL_f formula φ and a positive integer k, we can write a logic program P such that answer sets of P are in one-to-one correspondance with satisfying traces of φ with length up to k.

Example 1. *Consider the formula $\varphi = (F\, a) \wedge (F\, b) \wedge G\,(a \rightarrow X\, b) \wedge G\,(b \rightarrow X\, a)$. Applying the encoding proposed in [19], we can encode bounded satisfiability of φ with the following logic program P:*

```
% Directed acyclic graph reification of the input formula.
root(0). conjunction(0, 1). conjunction(0, 2). conjunction(0, 3).
conjunction(0, 4). eventually(1, 3). atom(3, a). eventually(2, 4).
atom(4, b). always(3, 5). always(4, 6). implies(5, 3, 8).
implies(6, 4, 10). next(8, 4). next(10, 3).
```

```
% Guess a state for each time-point t
time(0..k-1).
{ trace(T,A): atom(_,A) } :- time(T).

% Discard traces that are not models
:- root(X), not holds(X,0).

% Rules to evaluate extension of holds/2
holds(T,X) :- trace(T,A), atom(X,A).
holds(T,X) :- holds(T+1,F), next(X,F), time(T+1).
holds(T,X) :- holds(T,F), eventually(X,F).
holds(T,X) :- eventually(X,_), ... .
holds(T,X) :- trace(T,_), not trace(T+1,_), always(X,F), holds(T,F).
holds(T,X) :- always(X,F), holds(T,F), holds(T+1,F).
holds(T,X) :- conjunction(X,_), time(T), holds(T,F): conjunction(X,F).
```

where k is a runtime constant that is passed as input to the ASP system.

By applying standard rewriting techniques that are used to debug ASP programs [17], the logic program P can be transformed in a logic program P' whose *minimal unsatisfiable subprograms* with respect to a set of freshly-introduced objective atoms matching signature $phi/1$, correspond to subsets of φ that are either minimal unsatisfiable cores or satisfiable subformulas, whose shortest model exceeds the length k. We refer to the latter case as a k-MUC for φ.

Example 2. *Applying the rewriting sketched in [17] to the encoding of the previous example, we replacing the set of facts $conjunction(0, \_)$ with the following rules:*

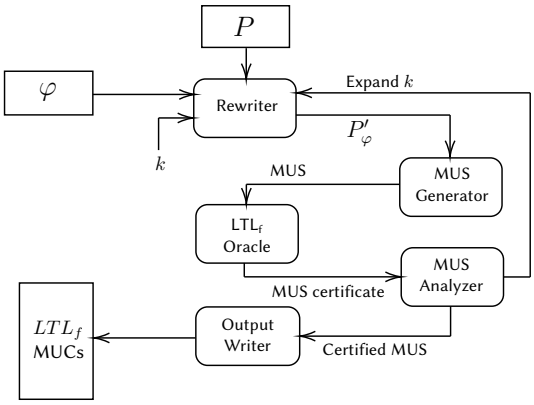
```
{ phi(1) }. { phi(2) }. { phi(3) }. { phi(4) }.
conjunction(0, 1) :- phi(1).
conjunction(0, 2) :- phi(2).
conjunction(0, 3) :- phi(3).
conjunction(0, 4) :- phi(4).
```

obtaining as a result the program P'. The MUSes of P' (with $k > 1$) are $\{phi(1),\ phi(3),\ phi(4)\}$, $\{phi(2),\ phi(3),\ phi(4)\}$. Indeed, this corresponds to the formulae $M_1 = (F\, a) \wedge G\,(a \rightarrow X\, b) \wedge G\,(b \rightarrow X\, a)$, $M_2 = (F\, b) \wedge G\,(a \rightarrow X\, b) \wedge G\,(b \rightarrow X\, a)$, which are the MUCs of φ.

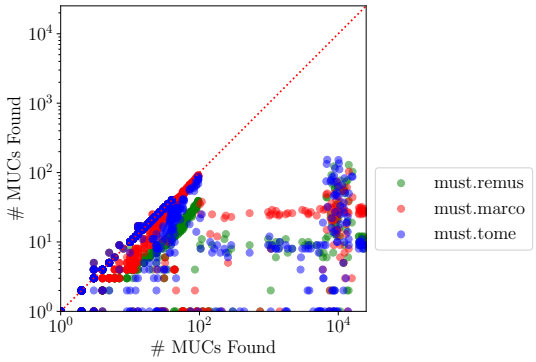
In general, it is not true that *all* MUSes of P' correspond to MUCs of φ, since the approach of [19] is not *complete* but encodes a semi-decision procedure for LTL_f satisfiability. In order to check that a k-MUC is MUC (i.e., it is unsatisfiable) we propose the usage of an off-the-shelf LTL_f satisfiability solver. Thus, we propose the architecture sketched in Figure 1a. We refer to k as the *search horizon* in the enumeration procedure; starting with $k = 1$, MUSes that correspond to LTL_f formulae that are found to be satisfiable are used to expand the value of k; subformulae that are found to be unsatisfiable returned as MUCs of φ. The enumeration procedure stops at a given k if all MUSes of P' corresponds to unsatisfiable subformulae of φ. Further details are provided in the extended version [34].

4. Preliminary Experiment

To validate our approach, we conduct a preliminary experiment comparing our prototype implementation with `must` [21], in the task of enumerating MUCs of LTL_f specifications in conjunctive form.



(a) A prototypical architecture leveraging an off-the-shelf ASP solver (*MUS Generator component*) and an off-the-shelf LTL$_f$ satisfiability checker (*LTL_f Oracle* component) to enumerate MUCs of an input formula φ.



(b) Number of MUCs computed using our prototype and `must`. Our prototype is able to enumerate more MUCs in all the instances.

Data and Execution environment. We collect unsatisfiable instances from the datasets [35, 36] (interpreted as LTL$_f$ formulae), which are standard datasets in LTL$_f$ satisfiability literature, for a total of 2079 unsatisfiable instances. As execution environment we use 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). Memory and time were limited to 8GB and 300s of real time, 700s of CPU time respectively.

The `must` system implements several *domain agnostic* MUC enumeration algorithms (ReMUS, TOME, and MARCO), and among many possible domains it supports also the LTL domain. We patch `must` according to the well-known LTL$_f$-to-LTL translation [1] in order to support our use case. As far as we know, `must` is the only publicly available system for enumerating MUCs supporting the LTL domain. The scatter plot in Figure 1b reports the results of our experiment. A point (x, y) corresponds to an LTL$_f$ instance where our prototype enumerates x MUCs and `must` enumerates y MUCs within timeout.

Overall, in all the instances, our approach performs better than any of the enumeration algorithms implemented in `must`. Further experiments are provided in the extended version.

5. Conclusions

Satisfiability of temporal specifications expressed in LTL$_f$ is crucial in several artificial intelligence application domains [2, 3, 4, 5]. Therefore, in case of unsatisfiable specifications, detecting reasons for unsatisfiability — e.g., computing its minimal unsatisfiable cores — is of particular interest. Specifically, this is essential whenever a specification ought to be satisfiable.

Recent works [8, 9] propose several approaches for single MUC computation but do not investigate enumeration techniques. However, enumerating MUCs for LTL$_f$ specifications is pivotal for enabling several reasoning services, such as explainability tasks [11], as in the propositional case [37, 38].

To tackle this issue, we propose an ASP-based "generate and check" approach to LTL$_f$ MUC enumeration, inspired by the domain agnostic MUC enumeration of [21]. We implement a prototype using `wasp` [39] and its MUS enumeration techniques [17]. Our preliminary experiment, featuring standard formulae in LTL$_f$ satisfiability benchmarking, shows promising results.

Concerning future works, we are interested in extending our experimental analysis and the theoretical framework behind the proposed approach.

Acknowledgments

This work was partially supported by MUR under the PRIN project PINPOINT Prot. 2020FNEB27, CUP H23C22000280006 and H45E21000210001.

References

[1] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: IJCAI, IJCAI/AAAI, 2013, pp. 854–860.

[2] F. Bacchus, F. Kabanza, Planning for temporally extended goals, Ann. Math. Artif. Intell. 22 (1998) 5–27.

[3] D. Calvanese, G. De Giacomo, M. Y. Vardi, Reasoning about actions and planning in LTL action theories, in: KR, 2002, pp. 593–602.

[4] G. De Giacomo, F. M. Maggi, A. Marrella, S. Sardiña, Computing trace alignment against declarative process models through planning, in: ICAPS, 2016, pp. 367–375.

[5] G. De Giacomo, M. Y. Vardi, Automata-theoretic approach to planning for temporally extended goals, in: ECP, volume 1809 of *LNCS*, 1999, pp. 226–238.

[6] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, Declare: Full support for loosely-structured processes, in: Proceedings of EDOC 2007, IEEE Computer Society, 2007, pp. 287–300.

[7] C. Di Ciccio, M. Montali, Declarative process specifications: Reasoning, discovery, monitoring, in: W. M. P. van der Aalst, J. Carmona (Eds.), Process Mining Handbook, volume 448 of *Lecture Notes in Business Information Processing*, Springer, 2022, pp. 108–152. doi:10.1007/978-3-031-08848-3_4.

[8] T. Niu, S. Xiao, X. Zhang, J. Li, Y. Huang, J. Shi, Computing minimal unsatisfiable core for LTL over finite traces, Journal of Logic and Computation (2023) exad049. URL: https://doi.org/10.1093/logcom/exad049. doi:10.1093/logcom/exad049.

[9] M. Roveri, C. Di Ciccio, C. Di Francescomarino, C. Ghidini, Computing unsatisfiable cores for ltlf specifications, J. Artif. Intell. Res. 80 (2024) 517–558.

[10] M. H. Liffiton, A. Previti, A. Malik, J. Marques-Silva, Fast, flexible MUS enumeration, Constraints An Int. J. 21 (2016) 223–250.

[11] T. Miller, Explanation in artificial intelligence: Insights from the social sciences, Artif. Intell. 267 (2019) 1–38.

[12] G. Audemard, F. Koriche, P. Marquis, On tractable XAI queries based on compiled representations, in: KR, 2020, pp. 838–849.

[13] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103.

[14] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (1991) 365–386.

[15] G. Brewka, M. Thimm, M. Ulbricht, Strong inconsistency, Artif. Intell. 267 (2019) 78–117.

[16] C. Mencía, J. Marques-Silva, Reasoning about strong inconsistency in ASP, in: SAT, volume 12178 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 332–342.

[17] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, F. Ricca, ASP and subset minimality: Enumeration, cautious reasoning and muses, Artif. Intell. 320 (2023) 103931.

[18] V. Fionda, G. Greco, LTL on finite and process traces: Complexity results and a practical reasoner, J. Artif. Intell. Res. 63 (2018) 557–623.

[19] V. Fionda, A. Ielo, F. Ricca, Ltlf2asp: Ltlf bounded satisfiability in asp, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, 2024, pp. 373–386.

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

[21] J. Bendík, I. Cerna, Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets, in: LPAR, volume 57 of *EPiC Series in Computing*, EasyChair, 2018, pp. 131–142.

[22] M. H. Liffiton, K. A. Sakallah, Algorithms for computing minimal unsatisfiable subsets of constraints, J. Autom. Reason. 40 (2008) 1–33. doi:10.1007/S10817-007-9084-Z.

[23] C. Mencía, J. Marques-Silva, Efficient relaxations of over-constrained csps, in: Proceedings of 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, IEEE Computer Society, 2014, pp. 725–732. doi:10.1109/ICTAI.2014.113.

[24] A. Meliou, S. Roy, D. Suciu, Causality and explanations in databases, Proc. VLDB Endow. 7 (2014) 1715–1716. doi:10.14778/2733004.2733070.
[25] S. Schlobach, R. Cornet, Non-standard reasoning services for the debugging of description logic terminologies, in: G. Gottlob, T. Walsh (Eds.), Proceedings of IJCAI'03, Morgan Kaufmann, 2003, pp. 355–362. URL: http://ijcai.org/Proceedings/03/Papers/053.pdf.
[26] R. Peñaloza, Axiom pinpointing, in: G. Cota, M. Daquino, G. L. Pozzato (Eds.), Applications and Practices in Ontology Design, Extraction, and Reasoning, volume 49 of *Studies on the Semantic Web*, IOS Press, 2020, pp. 162–177. URL: https://doi.org/10.3233/SSW200042. doi:10.3233/SSW200042.
[27] F. Baader, R. Peñaloza, Automata-based axiom pinpointing, J. Autom. Reason. 45 (2010) 91–129. URL: https://doi.org/10.1007/s10817-010-9181-2. doi:10.1007/S10817-010-9181-2.
[28] R. Sebastiani, M. Vescovi, Axiom pinpointing in lightweight description logics via horn-sat encoding and conflict analysis, in: R. A. Schmidt (Ed.), Procedding of the 22nd International Conference on Automated Deduction, volume 5663 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 84–99. doi:10.1007/978-3-642-02959-2_6.
[29] R. Peñaloza, F. Ricca, Pinpointing axioms in ontologies via ASP, in: G. Gottlob, D. Inclezan, M. Maratea (Eds.), Proceedings of LPNMR 2022, volume 13416 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 315–321. doi:10.1007/978-3-031-15707-3_24.
[30] J. Li, G. Pu, Y. Zhang, M. Y. Vardi, K. Y. Rozier, Sat-based explicit ltlf satisfiability checking, Artif. Intell. 289 (2020) 103369. doi:10.1016/J.ARTINT.2020.103369.
[31] R. Peñaloza, Explaining axiom pinpointing, in: C. Lutz, U. Sattler, C. Tinelli, A. Turhan, F. Wolter (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*, Springer, 2019, pp. 475–496. URL: https://doi.org/10.1007/978-3-030-22102-7_22. doi:10.1007/978-3-030-22102-7_22.
[32] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309. URL: https://doi.org/10.1017/S1471068419000450. doi:10.1017/S1471068419000450.
[33] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012.
[34] A. Ielo, G. Mazzotta, R. Peñaloza, F. Ricca, Enumerating minimal unsatisfiable cores of ltlf formulas, CoRR abs/2409.09485 (2024).
[35] V. Schuppan, L. Darmawan, Evaluating LTL satisfiability solvers, in: ATVA, volume 6996 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 397–413.
[36] J. Li, G. Pu, Y. Zhang, M. Y. Vardi, K. Y. Rozier, Sat-based explicit ltlf satisfiability checking, Artif. Intell. 289 (2020) 103369.
[37] J. Marques-Silva, Minimal unsatisfiability: Models, algorithms and applications (invited paper), in: ISMVL, IEEE Computer Society, 2010, pp. 9–14.
[38] J. Marques-Silva, M. Janota, C. Mencía, Minimal sets on propositional formulae. problems and reductions, Artif. Intell. 252 (2017) 22–50.
[39] M. Alviano, C. Dodaro, N. Leone, F. Ricca, Advances in WASP, in: LPNMR, volume 9345 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 40–54.

6. Online Resources

An extended, work-in-progress version of this work is available here.