La borsa di dottorato è stata cofinanziata con risorse del
Programma Operativo Nazionale Ricerca e Innovazione 2014-202 (CCI 2014IT16M2OP005)
Fondo Sociale Europeo, Azione I.1 “Dottorati Innovativi con caratterizzazione Industriale”

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in

Matematica e Informatica

Con il contributo di (Ente finanziatore)

Revelis srl

CICLO

XXXVII

Applications of Answer Set Programming to Linear Temporal Logic over Finite Traces

Settore Scientifico Disciplinare INF/01

Coordinatore: Ch.mo Prof. Giorgio Terracina

Giorgio Terracina
08.01.2025
11:01:02
GMT+02:00

Supervisori: Ch.mo Prof. Francesco Ricca

FRANCESCO RICCA
08.01.2025 11:16:50
GMT+02:00

Prof. Carmine Dodaro

Carmine
Dodaro

Dottorando: Dott. Antonio Ielo

Antonio Ielo
08.01.2025 11:04:44
GMT+02:00

A mamma e papà,
per avermi sempre sostenuto e incoraggiato.

Abstract

Linear Temporal Logic over Finite Traces (LTL_f) is a popular logic to express high level specifications that require reasoning about time. It was introduced in the seminal work of De Giacomo and Vardi, motivated by the fact that many Artificial Intelligence applications that require temporal reasoning are more intuitively described as finite execution sequences, rather than infinite execution sequences as in Linear Temporal Logic.

Due to the high complexity that characterizes LTL_f-related problems, efficienty systems are required. Reasoning tasks in LTL_f are usually tackled by automata theoretic techniques or by SAT-based approaches.

This thesis proposes an alternative approach to LTL_f-related reasoning problems that is based on Answer Set Programming (ASP). Answer Set Programming is a declarative logic programming paradigm that stems from research in stable model semantics for logic programs and knowledge representation and reasoning.

ASP is more expressive than SAT, thus, the solutions we obtain are more compact, intuitive and readable than SAT counterparts; moreover the usage of ASP does not result in a compromise on efficiency and effectiveness. Indeed, the approach presented in this thesis often obtains better performance with respect to existing SAT-based solutions; also, ASP enabled the development of novel solutions for unsatisfiable core enumeration and query checking in declarative process mining contexts.

Sommario

La *Logica Temporale Lineare su Tracce Finite* (LTL_f) è una logica molto popolare per esprimere specifiche dichiarative che riguardano aspetti temporali.

È stata introdotta da De Giacomo e Vardi, motivata dal fatto che molte applicazioni di Intelligenza Artificiale che richiedono ragionamento temporale sono più intuitivamente descritte come sequenze di esecuzione finite, piuttosto che sequenze di esecuzione infinite (come in LTL).

A causa dell'alta complessità che caratterizza i problemi relativi a LTL_f, sono necessari sistemi efficienti. I compiti di ragionamento in LTL_f sono solitamente affrontati con tecniche basate sugli automi a stati finiti o con approcci basati su SAT.

Questa tesi riguarda le applicazioni dell'Answer Set Programming ai problemi di ragionamento legati a LTL_f. Answer Set Programming (ASP) è un paradigma di programmazione dichiarativa che nasce dalla ricerca sulla semantica dei modelli stabili dei programmi logici e sulla rappresentazione e il ragionamento della conoscenza.

L'ASP è più espressivo di SAT, pertanto le soluzioni che otteniamo sono più compatte, intuitive e leggibili rispetto alle controparti basate su SAT; inoltre, l'uso dell'ASP non comporta compromessi in termini di efficienza ed efficacia. Infatti, gli approcci presentati in questa tesi ottengono spesso prestazioni migliori rispetto alle soluzioni esistenti basate su SAT; inoltre, l'ASP ha permesso lo sviluppo di nuove soluzioni per i problemi dell'enumerazione di "unsatisfiable cores" e il "query checking" nel contesto del process mining dichiarativo.

Contents

Chapter 1

Introduction

1.1 Context & Motivation

Temporal reasoning is a key facet of Artificial Intelligence [152], which is relevant to areas such as knowledge representation, planning and multi-agent systems. Many logic-based formalisms have been proposed to deal with the notion of time in Computer Science [56]. Among them, Linear Temporal Logic (LTL) has been first proposed to reason about programs' execution [138] and reactive systems [137]. It has since become a staple in the fields of verification and model checking [136, 45].

LTL extends propositional logic by means of *temporal operators*. In this logic, interpretations are infinite sequences — *"traces"* — of propositional symbols — *"states"* —, which are to be interpreted as "a propositional interpretation evolving over time". Propositional logic does not provide a notion of time, and temporal aspects of specifications must be "manually dealt with". The typical approach is to define multiple variables that model the same concept, but over different time-points. Indeed, such a construction formally requires a known upper bound on the "number of time-points" one considers. On the other hand, in LTL, temporal operators allow a natural representation of and reasoning about time, that does not require such an upper bound.

As an example, the formula $\mathsf{G}(a \rightarrow b \wedge \mathsf{X}c)$ reads as *whenever a is true, b is true and c will be true in the next state*. Intuitively, the "always" operator G requires the implication $a \rightarrow b \wedge \mathsf{X}c$ to be true in *all* states of a trace, that is whenever a belongs to a certain state, it must be that b belongs to the same state and c belongs to the *next* state. In particular, the notion of "next state" implicitly refers

to states that follow a state that contains an a; there's no explicit mention of time-points. An example of violating trace would be $\{a,b\} \cdot \{c\} \cdot \{a\} \cdot \{c\} \cdots$; since the second occurrence of a, is followed by a c, but there is no b in the same state. An example of trace satisfying the formula could be the two states $\{a,b\} \cdot \{c\}$ alternating—traces must be infinite sequences.

LTL enables to perform temporal reasoning which is *qualitative* in nature, dealing essentially with *relative order of events*; further extensions of LTL, such as metric temporal logic [96], also take into account *quantitative* aspects of temporal reasoning. This makes LTL an appealing formalism to write specifications in a high-level, declarative and composable fashion [11, 101].

While in LTL interpretations are infinite sequences (which are suitable for reactive systems, that are expected to run "forever"), many applications in Artificial Intelligence are better described by means of *finite execution traces* [55]. Prominent examples are robotics [83], planning [85, 14, 54], agent-based reasoning [84], and business process management [43] . Linear Temporal Logic *over finite traces* (LTL_f) is a variant of LTL introduced by De Giacomo and Vardi [55]. LTL_f shares the same syntax of LTL, but interpretations are limited to finite sequences.

This thesis provides several contributions in this context. In particular, we address the following tasks: (i) bounded satisfiability (e.g., does a specification admit a model of a given length?); (ii) minimal unsatisfiable core extraction (what subformulae make a specification unsatisfiable?); (iii) passive learning (is there a specification that separates two kinds of behavior?); and (iv) applications to declarative process mining. All these tasks of LTL_f, as in standard LTL, are usually tackled in terms of propositional satisfiability [78, 107] or automata-theoretic techniques [51].

In this thesis, we propose a novel perspective for tackling the above-mentioned tasks, and propose the usage of Answer Set Programming [21] as a tool for modeling and solving reasoning problems in LTL_f.

Answer Set Programming (ASP) [21, 131, 82] is a declarative programming paradigm, developed in the area of knowledge representation and reasoning [90], with roots in the stable model semantics of logic programs [82]. ASP is particularly well-suited to solve combinatorial optimization problems and knowledge-intensive NP-complete problems [65, 66]. ASP also allows to model problems up to Σ_2^P complexity class [62] in the polynomial hierarchy, using disjunctive extensions . Limiting ourselves to normal rules and non-recursive aggregates, one can solve problems of complexity up to NP. Moreover, being a superset of Datalog [33, 32], ASP is also suitable as query language for deductive knowledge bases and related tasks [64, 17, 116, 18].

Since ASP is more expressive than SAT, SAT-based techniques for LTL_f can be effectively lifted to ASP. Due to this expressiveness, the solutions we obtain with ASP can be more compact, intuitive, and readable compared to their SAT counterparts. Importantly, this enhanced clarity does not come at the cost of efficiency or effectiveness.

In the following, we introduce each one of the aforementioned contributions we provide in this thesis.

1.1.1 Bounded Satisfiability

As in any logic-based formalism, *satisfiability* is a central problem. For LTL_f, satisfiability is the problem of deciding whether a formula admits satisfying traces (and possibly to provide such a trace).

In typical application scenarios, LTL_f specifications describe requirements for a system, while traces represent how the state of such a system evolves over time, each state of the trace providing a "propositional snapshot" of the system in a given point in time.

Satisfiability of a specification guarantees that the given requirements are compatible, and possibly exhibits a trace that satisfies the requirements. On the other hand, unsatisfiable specifications are not interesting: they describe behavior that is not attainable by the system. Thus, LTL_f satisfiability problem can be used first and foremost to understand whether desired behavior is feasible; possibly, to get examples of such behavior.

Furthermore, inspired by bounded model checking [20, 19] procedures, satisfiability can also be used to *search for witnesses of undesired (or unexpected) behavior*: if we are interested in whether our system (formalized as a formula φ) satisfies a property, satisfiability of $\neg\varphi$ provides a *counterexample* of the system exhibiting such property.

Complexity-wise, finiteness does not make the problem any easier, as LTL_f satisfiability is a PSPACE-complete problem as in standard LTL [55]. Given this high complexity, addressing satisfiability problem calls for efficient systems.

Our contribution is an ASP-based approach for *bounded satisfiability*, that assuming a known upper bound on model length, which improves upon other publicly available systems for the same task. Our experiments show that the approach is particularly well suited for (satisfiable) specifications characterized by having "lengthy shortest models", e.g., that are satisfiable but do not admit short models.

1.1.2 Minimal Unsatisfiable Cores Extraction

Due to the compositional nature of LTL_f, specifications are typically written by conjoining multiple subformulae together. As specifications become bigger and more complex, they easily (and sometimes, unexpectedly!) become unsatisfiable.

In this case, whenever a formula is unsatisfiable it is of interest to understand *why* it is so (e.g., which subformulae). This is particularly relevant whenever a specification is hand-written and *believed to be satisfiable*. The process that leads modelers to understand why a should-be-satisfiable specification is unsatisfiable is, essentially, the declarative equivalent of "debugging" procedural programs [148].

In propositional logic, this problem is often framed as extracting *minimal unsatisfiable cores* [113] — subset-minimal sets of formulae whose conjunction is unsatisfiable. Similar approaches have been developed in LTL_f; Roveri et al. [144] provide a detailed survey of several techniques for single unsatisfiable core extraction (with no minimality guarantee), while Niu et al. [132] focus on a single minimal unsatisfiable core extraction, using SAT-based techniques. To the best of our knowledge, there is no publicly available tool to *enumerate* minimal unsatisfiable cores.

Our contribution here is an ASP-based approach to *enumerate* minimal unsatisfiable cores (MUCs) of LTL_f specification in conjunctive form. Surprisingly, our approach is also more effective on single MUC extraction with respect to the state-of-the-art, despite being designed for a harder task.

1.1.3 Passive Learning

Composability, readability and declarativity make LTL_f an interesting language for writing specifications; the same characteristics make LTL_f an interesting formalism to *describe* and *explain* behaviors (described by "examples", provided as sets of traces exhibiting/not exhibiting the temporal property) in terms of temporal properties [29].

It is in fact well-known in verification literature that many interesting patterns that arise in the specification of systems can be modeled by means of small, highly composable LTL_f formulae [61].

In applications, LTL_f specifications typically describe a system's behavior. Developing manual specifications of system behaviors is not an easy endevour, and automatic approaches for inferring LTL_f specifications from data (e.g., execution traces) are becoming increasingly important [72].

In this context, *passive learning* refers to the problem of deciding whether there exists a formula that "distinguishes" two sets of traces by the temporal behavior they exhibit. More formally, given two disjoint sets of traces P and N, we are interested in a formula φ such that all traces $\pi \in P$ are models of φ, and none of the traces in N are models of φ, that is $\tau \not\models \varphi$ for all $\tau \in N$.

Thus, intuitively, traces in P are examples of the behavior φ describes, while N do not exhibit such property.

Passive learning enables to learn models that explain observed behavior that is expressive, human-interpretable [29] yet equipped with a formal semantics.

Our contribution is an ASP-based framework for LTL_f passive learning, which makes use of the *learning from answer set* [102] inductive logic programming [48] setting. The approach is based on an abduction-like approach that searches over LTL_f formulae syntax trees, and outperforms similar SAT-based techniques [130, 73], while being more declarative and easier to read and modify (e.g., to add inductive biases on searched formulae).

1.1.4 Applications to Declarative Process Mining

Although LTL_f is expressive and intuitive, it also comes with all the classic *downsides* of declarative specification languages, especially when applied by people that do not have a strong background in logic or computer science. First and foremost, although LTL_f provides first-class support for temporal concepts, it sometimes leads to "un-intuitive behavior" — a recent survey classifies and highlights several "classic mistakes" in the use of LTL and LTL_f [89].

For this reason, especially in application domains where such skills "are not expected", LTL_f specifications are not used in a direct, "free-form" fashion, but in a more "controlled fashion" through the usage of a small set of intuitive, well understood *patterns*. A field where LTL_f has proven effective with such an approach is declarative process mining [57]. *Declare* [135] is a process modeling language based on a set of LTL_f "templates", chosen among properties that frequently appear in verification and model checking literature [61], that describe the temporal relationship between events in a process execution trace.

Although Declare has not been originally defined in terms of LTL_f templates, but dates back to literature on verification and model checking, the formalization of Declare as LTL_f has started a successful and rich stream of research [57, 126], which lead to results in process monitoring, process discovery and query checking, which make Declare the de-facto language for declarative process mining. Thus, in practice, the Declare approach enables process mining practitioners to

benefit from LTL_f automated reasoning techniques, but requries little to no formal background, since Declare patterns can be easily understood without mentioning their LTL_f semantics.

Recently, there has been a surge of interest in the application of ASP to Declare-based reasoning tasks [35, 40, 99]. A straightforward way to tackle Declare reasoning tasks in ASP is to consider the LTL_f definition of Declare templates, and/or to exploit the link between such LTL_f formulae and its equivalent finite state machine [51, 40]. However, since Declare is simpler than full-fledge LTL_f (e.g., most Declare templates do not involve nested temporal operators), it is expected to be possible to provide some ad-hoc encodings in ASP.

Our contribution is a *direct* ASP encoding for Declare, targeting the conformance checking and query checking tasks (i.e., that does not rely on translation to automata nor LTL_f). We show that such an encoding is better (time- and memory-wise) than other available ASP encodings, and is also competitive with recent Python libraries for Declare.

1.2 Publications

Summarizing, this thesis provides several contributions in the context of LTL_f, that can be summarized as follows: (i) improve scalability of LTL_f semi-decision procedures; (ii) a novel approach to LTL_f MUC enumeration; (iii) an Inductive Logic Programming-based technique for LTL_f passive learning; (iv) ASP-based approach for Declare-based process mining tasks. These contributions have led to a number of scientific publications:

- Valeria Fionda, Antonio Ielo, Francesco Ricca: LTLf2ASP: LTLf bounded satisfiability in ASP. LPNMR 2024: 373-386
- Francesco Chiariello, Valeria Fionda, Antonio Ielo, Francesco Ricca: A Direct ASP Encoding for Declare. PADL 2024: 116-133. *Distinguished paper at the 26th International Symposium on Practical Aspects of Declarative Languages conference*
- Francesco Chiariello, Valeria Fionda, Antonio Ielo, Francesco Ricca: A Direct ASP Encoding for Declare. TPLP (to appear)
- Antonio Ielo, Mark Law, Valeria Fionda, Francesco Ricca, Giuseppe De Giacomo, Alessandra Russo: Towards ILP-Based LTLf Passive Learning.

ILP 2023: 30-45. *This paper was awarded the Best Paper Award at the 23rd International Conference on Inductive Logic Programming conference.*

- Antonio Ielo, Giuseppe Mazzotta, Rafael Peñaloza, Francesco Ricca: Towards ASP-based Minimal Unsatisfiable Cores Enumeration for LTLf. OVERLAY 2024 (to appear)

Not strictly related to the contents of this thesis, during my PhD I also contributed to the following papers and projects:

- Francesco Chiariello, Antonio Ielo, Alice Tarzariol: ILP-based Petri net model repair. LPNMR 2024 85-97
- Valeria Fionda, Antonio Ielo, Francesco Ricca: Logic-based Composition of Business Process Models. KR 2023: 272-281
- Antonio Ielo, Salvatore Falco, Salvatore Iiritano, Patrizia Piro, Ada Polizzi and Francesco Ricca: An ASP-based approach to water distribution system reconstruction. LPNMR 2024 140-153
- Mario Alviano, Antonio Ielo and Francesco Ricca: Efficient compliance computation in Probabilistic Declarative Specifications. PLP 2024 (ICLP Workshops, to appear)
- Giuseppe De Giacomo, Valeria Fionda, Nicola Gigante, Antonio Ielo, Francesco Ricca and Alessandra Russo: A Declarative Framework for Temporal Reasoning in Green-Aware Applications (GreenAware AI 2024, AIxIA Workshops, to appear)

In particular, "An ASP-based approach to water distribution system reconstruction" was developed in the context of PON-RI 2020 (Azione IV.5 "Green") by Revelis srl, which partially funds my PhD scholarship on the *Artificial Intelligence for water network control optimisation* theme.

1.3 Thesis structure

The rest of this thesis is organized as follows:

- Part I introduces preliminary notions of Answer Set Programming (Chapter 2), Linear Temporal Logic over Finite Traces (Chapter 3) and declarative process mining (Chapter 4) for the rest of the thesis.

- Part II is about applications of Answer Set Programming in the domain of LTL_f satisfiability, describing the `ltlf2asp` and `mus2muc` systems:
  - Chapter 5 introduces the ASP encoding of LTL_f semantics, which is the basis for most of the encodings presented in this thesis. It also links the encoding with the SAT-based approach of LTL2SAT [68].
  - Chapter 6 introduces an ASP-based framework for LTL_f MUCs enumeration which leverages ASP minimal unsatisfiable subprograms enumeration algorithms as implemented in the `wasp` [7] solver.
- Part III deals with applications
  - Chapter 7 describes a direct ASP encoding for the main constraints of the Declare process modeling language, which improve over "general purpose" encodings for Declare, and is competitive with recent tools in Declare-based process mining [60].
  - Chapter 8 describes an ASP-based approach to learn LTL_f formulae from labeled examples, in the *learning from answer sets* inductive logic programming framework.
- Chapter 9 discusses works related to the contents of this thesis.
- Chapter 10 sums up the contributions of this thesis, as well as discuss possible future directions for this line of research.

Part I

Preliminaries

Chapter 2

Answer Set Programming

This chapter briefly introduces minimal notions of Answer Set Programming (ASP) which will be referred to throughout the thesis. The first two sections introduce syntax and semantics. The third section consists of an example, which showcases the typical usage workflow whenever applying ASP to solve problems. The fourth section introduces the Learning from Answer Sets learning task in Inductive Logic Programming.

2.1 Syntax

A *term* is either a *variable* or a *constant*, where *variables* are alphanumeric strings starting with uppercase letter, while *constants* are either integer number or alphanumeric string starting with lowercase letter. An *atom* is an expression of the form $p(t_1,\ldots,t_n)$ where p is a predicate of arity 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 code listings, $\leftarrow$ in rules will be typeset as `:-`. 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 occurring in r appears in at least

one positive literal in the body of r.

A *program* is a finite set of safe normal rules. In the rest of the thesis we will also use also *choice rules*, which abbreviate complex expressions [27]. 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 $\{e_1; \ldots; e_m\} \leftarrow b_1, \ldots, b_n$, where each e_i ($i \in 1, \ldots, m$) is of the form $h_i : l_1^i, \ldots, l_{k_i}^i$, and it represents 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 $i \in 1, \ldots, m$ and nh^i is a fresh atom not appearing anywhere else.

2.2 Semantics

Given a program P, the *Herbrand Universe* of P, $\mathscr{U}_P$, denotes the set of constants that appear in P, while the *Herbrand Base*, $\mathscr{B}_P$, denotes the set of ground atoms obtained from predicates in P and constants in $\mathscr{U}_P$. 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 $\mathscr{U}_P$. Given a program P, $ground(P)$ denotes the union of ground instantiations of rules in P. An *interpretation* $I \subseteq \mathscr{B}_P$ is a set of atoms.

Example 2.2.1. *Consider the non-ground logic program P:*

```
a(0). b(0). a(1).
c(X) :- a(X), b(X).
```

Then $\mathscr{U}_P = \{0,1\}$*, and* $\mathscr{B}_P = \{a(0), b(0), a(1), b(1), c(0), c(1)\}$*. Thus, the* (theoretical) grounding *of P would be the logic program* $ground(P)$*:*

```
a(0). b(0). a(1).
c(0) :- a(0), b(0).
c(1) :- a(1), b(1).
```

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 [81], 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 from the body of remaining rules.

Given a program P, and a model I, then I is also a *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*. Given an answer set S and a signature σ, the *projection of S on* σ is the set $S_{|\sigma} = \{\alpha \in S : \alpha \text{ matches } \sigma\}$.

Example 2.2.2. *Consider the logic program:*

```
a :- not b.
b :- not a.
```

Consider its four possible interpretations, $I_1 = \{\}$, $I_2 = \{a\}$, $I_3 = \{b\}$ *and* $I_4 = \{a,b\}$. *We start by computing the reduct* P^{I_2}, *thus* deleting *the second rule and simplifying the first rule down to the fact a:*

```
a.
```

Indeed, a is the minimal model of P^{I_2}, *thus* $\{a\}$ *is a stable model. Since* $I_2 = \{a\}$ *is a stable model,* I_4 cannot *be a stable model, since* $I_2 \subset I_4$. *By symmetry, we obtain that also* $\{b\}$ *is a stable model. Finally, if* I_1 *were to be a stable model,* I_2 *and* I_3 *could not be stable models, since* $I_1 \subset I_2$ *and* $I_1 \subset I_3$. *Indeed, computing* P^{I_1} *would yield the logic program:*

```
a. b.
```

that does not admit an empty minimal model.

Minimal Unsatisfiable Subprograms (MUSes). Consider a program P and a set of objective atoms $O \subseteq \mathscr{B}_P$, with $|O| = |P|$. For $S \subseteq O$, we denote by $\mathsf{enforce}(P,O,S)$ the program obtained from P by adding (i) a choice rule over atoms in O (i.e., $\{o_1;\ldots;o_n\} \leftarrow$); (ii) a set of constraints of the form $\leftarrow not\, o$, for every $o \in S$ and (iii) adding the objective atom o_i to the positive body of rule $\rho_i \in P$. Intuitively, $\mathsf{enforce}(P,O,S)$ denotes an augmentation of the program P in which the objective atoms can be arbitrarily chosen (i.e., either as true or false) but the atoms in S are required to be true.

An *unsatisfiable subset* for P w.r.t. the set of (fresh) objective atoms O is a set of atoms $U \subseteq O$ such that $\mathsf{enforce}(P,O,U)$ 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 $U' \subset U$, $U' \notin US(P,O)$.

We provide one example, which showcases how MUSes can be exploited to diagnose reasons for incoherence.

Example 2.2.3. *Consider the (incoherent) logic program P:*

```
a :- not b.
c :- not a.
b :- not c.
e :- not c.
f :- not e.
c :- not f.
```

Let $O = \{o(0), o(1), o(2), o(3), o(4), o(5)\}$*, where we assume that the atom* $o(i)$ *is "matched" to the i-th rule in P as it appears in the listing above. Then, this is the logic program* $\mathsf{enforce}(P, O, \{\})$*:*

```
a :- not b, o(0).
c :- not a, o(1).
b :- not c, o(2).
e :- not c, o(3).
f :- not e, o(4).
c :- not f, o(5).
{ o(0); o(1); o(2); o(3); o(4); o(5) }.
```

Notice that $\mathsf{enforce}(P, O, S)$ *for* $S \subseteq O$ *is obtained by adding constraints (involving atoms in O) to* $\mathsf{enforce}(P, O, \{\})$*. Given the logic program* $\mathsf{enforce}(P, O, \{\})$*, we can invoke an MUS enumeration algorithm (such as the ones implemented in* `wasp` *[59]) to compute the MUSes of P wrt O, namely:*

$$\{o(0), o(1), o(2), o(4)\}$$

$$\{o(0), o(3), o(4), o(5)\}$$

which can be decoded *into two rule-wise minimal incoherent subprograms:*

```
a :- not b.
c :- not a.
b :- not c.
f :- not e.
```

and

```
a :- not b.
e :- not c.
f :- not e.
c :- not f.
```

2.3 Solving Problems with ASP: An Example

The typical way to use ASP to solve problems is to write a logic program P whose answer sets map to solutions of the original problem. P is usually written in a *uniform* fashion, as a non-ground logic program which models the problem, assuming it will be jointly evaluated with a suitable set of *input facts* that model a particular instance. We provide an example using the well-known graph coloring problem. In this example, and in the rest of the thesis, we use the input language of the CLINGO system. The reader can refer to Gebser et al. for a complete reference [79].

Example 2.3.1. *Let G be an undirected graph, with vertex set V and edge set E. We say that G is k-colorable if there exists a function $\lambda : V \mapsto \{1,\ldots,k\}$ such that $(v,w) \in E$ implies that $\lambda(v) \neq \lambda(w)$. Given an input graph G, determining whether λ exists is known as the k-colorability problem. It is a well-known NP-complete problem for $k \geq 3$.*

First, problem instances are graphs. We represent them by means of an $edge/2$ predicate, modeling a graph G by the set of facts $\{edge(v,w) : (v,w) \in E\}$. The following logic program Π_C models 3-colorability:

```
color(0..k-1).
node(X;Y) :- edge(X,Y).
{ assign(C,X): color(C) } = 1 :- node(X).
:- assign(C,X), assign(C,Y), edge(X,Y).
```

It follows the classic guess & check *approach, where choice rules are used to generate the search space, and constraints are used to prune unfit candidate solutions. Given an answer set of $\Pi_C \cup G$, we decode a solution by projecting on the $assign/2$ predicate. In particular, if $assign(c,x)$ belongs to the answer set, it is to be interpreted as $\lambda(x) = c$.*

2.4 Learning from Answer Sets

Inductive Logic Programming (ILP) is a symbolic machine learning technique that uses logic programs as inductive bias [48]. That is, the output of ILP algorithms are logic programs, under a given semantics. ILASP is a state-of-the-art ILP system to learn ASP programs [102] from sets of examples.

A *learning from answer sets* (LAS) task is a tuple $\mathscr{T} = (H,B,E^+,E^-)$, where H is the *hypothesis space*, defining a set of candidate rules, B is an ASP program

called *background knowledge*, while E^+ and E^- are set of *positive* and *negative examples*, respectively. Each example is denoted with a triple (Inc, Exc, C), where Inc and Exc are sets of atoms, called *inclusion* and *exclusion* sets respectively, and C is an ASP program, called *context*.

Example 2.4.1. *Suppose we are interested in learning logic programs composed of rules of normal rules of length (e.g., number of literals) using the atoms a and b. Then, our hypothesis space would be composed of the following rules (one per line):*

```
:- a.
:- b.
b.
a.
:- not b.
:- not a.
:- a, b.
b :- a.
a :- b.
:- a, not b.
a :- not b.
:- b, not a.
b :- not a.
:- not a, not b.
```

The ILASP system admits different ways to define H: explicitly listing all its elements, *mode biases*, an annotation-like syntax which declaratively describes what literals are allowed in the head and body of a rule $h \in H$, and meta-programming approaches where candidate rules are answer sets of a logic program. We refer the reader to the ILASP manual [93] for further details and examples.

Example 2.4.2. *Using* mode biases*, the hypothesis space of Example 2.4.1 can be compactly denoted by the directive:*

```
#modeh(a). #modeh(b).
#modeb(a). #modeb(b).
```

where the `#modeh` *(*`#modeb`*) directive states that an atom can belong to the head (body) of a normal rule.*

We say that an interpretation $\mathscr{I}$ is an *accepting answer set* of an example (Inc, Exc, C) with respect to a program P if $\mathscr{I} \in AS(B \cup C)$ such that $Inc \subseteq \mathscr{I}$ and $Exc \cap \mathscr{I} = \emptyset$.

Given a LAS task $\mathscr{T}$, ILASP searches for an *inductive solution* $h \subseteq H$ such that (i) for each positive example $e \in E^+$, there is some accepting answer set of e with respect to $B \cup h$, and (ii) for any negative example $e \in E^-$, there is no accepting answer set of e with respect to $B \cup h$. Together with $\mathscr{T}$, ILASP allows the definition of a scoring function σ to be used as optimization criteria, which by default is the length of the solution. ILASP returns an inductive solution h that is optimal wrt σ, i.e., there is no other $h' \subseteq H$ such that $\sigma(h') < \sigma(h)$.

Example 2.4.3. *Suppose we wish to learn the logic program in Example 2.2.2, using the hypothesis space in Example 2.4.1 and an empty background knowledge. Using the following example:*

```
#pos({b},{a},{}).
```

We can read this example as there should be an answer set that contains b but not a. *Intuitively, the shortest solution satisfying this requirement is the fact b, which we would obtain running ILASP. Indeed, it admits a unique answer set which extends $\{b\}$, and is disjoint from $\{a\}$, thus it is an accepting answer set for the given example. To refine our program, we might consider adding the negative example:*

```
#neg({b},{},{a.}).
```

The inductive solution computed by the ILASP system is now:

```
b :- not a.
```

Finally, we might add a new positive example:

```
#pos({a},{b},{}).
```

Now, running ILASP, we obtain as a solution the target logic program shown in Example 2.2.2. Notice that the two positive examples would have been sufficient to learn the logic program, and the negative example is redundant.

Chapter 3

Linear Temporal Logic over Finite Traces

This chapter briefly introduces minimal notions of Linear Temporal Logic over Finite Traces (LTL_f) which will be referred to throughout the thesis. The first section recaps syntax and semantics of LTL_f. The second section defines the satisfiability problem and defines the notion of minimal unsatisfiable core for LTL_f specifications. The last two sections describe the concept of *propositionalization* of an LTL_f formula and the relationship between finite state machines and LTL_f.

3.1 Syntax & Semantics

Let $\mathscr{A}$ be a finite set of symbols, also referred to as *atomic formulae* or *atoms*. An LTL_f formula φ over $\mathscr{A}$ is defined according to the following grammar:

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi_1 \mathsf{U} \varphi_2,$$

where $a \in \mathscr{A}$. The *temporal* operators X and U are called respectively *next* and *until*.

A *finite trace* is a sequence $\pi = \pi_0 \cdots \pi_{n-1}$ of n propositional interpretation $\pi_i \subseteq \mathscr{A}$, $i = 0, \ldots, n-1$, for some $n \in \mathbb{N}$. The interpretation π_i is also called a *state*, and $|\pi| = n$ denotes the trace *length*. We refer to the *size* of a formula φ, denoted by $|\varphi|$ as the number of its subformulae.

Let φ be an LTL_f formula, π a finite trace, $0 \leq i < |\pi|$ an integer, then the *satisfaction* relation, denoted by $\pi, i \models \varphi$, is defined recursively as follows:

- $\pi, i \models \top$;
- $\pi, i \models a$ iff $a \in \pi_i$;
- $\pi, i \models \neg\varphi$ iff $\pi, i \models \varphi$ does not hold;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \mathsf{X}\varphi$ iff $i < |\pi| - 1$ and $\pi, i+1 \models \varphi$;
- $\pi, i \models \varphi_1 \mathsf{U} \varphi_2$ iff $\exists j$ with $i \leq j \leq |\pi| - 1$ s.t. $\pi, j \models \varphi_2$ and $\forall k$ with $i \leq k < j$, $\pi, k \models \varphi_1$.

We say that π is a *model* for φ if $\pi, 0 \models \varphi$, denoted as $\pi \models \varphi$.

An $\mathrm{LTL_f}$ formula φ expresses temporal concepts by means of combining temporal and propositional operators, that (inductively) define a set of *constraints* on the states of a trace π—the set of conditions such that $\pi \models \varphi$. Intuitively, the formula $\mathsf{X}\varphi$ denotes that on the *next state* φ holds, while the formula $\varphi \mathsf{U} \psi$ denotes that *at some point in the future* ψ holds, and up to that point φ holds.

Beyond the operators introduced, we also assume common propositional operators ($\vee$, $\rightarrow$, $\longleftrightarrow$, etc.) with the usual meaning. Notice that propositional operators' semantics does not involve *multiple* states.

It is also common to define further temporal operators. In particular, for temporal operators, we define the *eventually* operator $\mathsf{F}\varphi \equiv \top \mathsf{U} \varphi$ ("φ holds at least once in the future"), the *always* operator $\mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\phi$ ("φ always holds, that is it false that its opposite $\neg\varphi$ holds at least once"). The *weak until* operator $\varphi \mathsf{W} \varphi' \equiv \mathsf{G}\varphi \vee \varphi \mathsf{U} \varphi'$, which relaxes the standard until operator, allowing φ' to *never occur*) and *weak next* operator $\mathsf{X}_\mathsf{w}\varphi \equiv \neg\mathsf{X}\neg\varphi \equiv \mathsf{X}\varphi \vee \neg\mathsf{X}\top$, which relaxes the standard next operator allowing for *no subsequent states*. Lastly, another common operator is the *release* operator $\phi \mathsf{R} \psi$ is defined as the dual operator of until, $\phi \mathsf{R} \psi \equiv \neg(\neg\phi \mathsf{U} \neg\psi)$.

Although $\mathrm{LTL_f}$ and LTL share the same syntax, interpreting a formula over finite traces results in very different properties. As an example, consider the *fairness* constraint of the form $\mathsf{GF}\varphi$. In LTL, this kind of formulae means that *it is always true that in the future φ holds*. However, in $\mathrm{LTL_f}$, this is equivalent to stating that φ *holds in the last state of the trace*. Thus, while the formula admits only infinite traces as counterexamples when interpreted in LTL, it admits finite counterexamples when interpreted as $\mathrm{LTL_f}$. It holds more generally that *(sic) direct nesting of temporal operators yields un-interesting formulae in* LTL_f [55].

3.2 Satisfiability & Unsatisfiability

The main computational problem in LTL_f is the *satisfiability problem*, that is to determine whether an LTL_f formula φ admits at least one model. If φ admits at least one model we say it is *satisfiable*; if it admits none it is *unsatisfiable*; if all traces are models of φ, we say φ is a *valid* formula.

The satisfiability problem for LTL_f is known to be PSPACE-complete [55], as for standard LTL [138], although fixing an upper bound on model length (also known as *fixed traces semantics*) yields NP-completeness for the problem [68]. We refer to satisfiability under upper bounded model lengths as *bounded satisfiability*.

Let φ be a conjunction of LTL_f formulae, $\varphi = \phi_1 \wedge \cdots \wedge \phi_n$. We say that φ is in *conjunctive form*. With a slight abuse of notation, we will treat formulae in conjunctive form as the set of its conjuncts, that is the set $\{\phi_1, \ldots, \phi_n\}$. If φ is unsatisfiable, we refer to any of its unsatisfiable subsets as an *unsatisfiable core*. An unsatisfiable core is *minimal* whenever all its proper subsets are satisfiable.

Example 3.2.1. *Consider the following formulae:* $\varphi_1 = F(a)$, $\varphi_2 = F(b)$, $\varphi_3 = G(a \rightarrow X(b))$, $\varphi_4 = G(b \rightarrow X(a))$. *It is easily shown that* $\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$ *is unsatisfiable:* φ_1 *and* φ_2 *require a, b to appear at least once in a satisfying trace. However, once a (resp. b) appears, it is required for b (resp. a) to appear, by* φ_3 *(resp.* φ_4*). No finite trace can satisfy such behavior. Thus, the conjunction is unsatisfiable. Notice the same formula would be satisfiable in LTL. Here, unsatisfiability stems from the interaction of* φ_3 *and* φ_4 *with either* φ_1 *(or* φ_2*). It is easy to verify that* $\{\varphi_1, \varphi_3, \varphi_4\}$ *and* $\{\varphi_2, \varphi_3, \varphi_4\}$ *are minimal unsatisfiable cores of the specification. Notice that, once minimal unsatisfiable cores are known, we can* repair *the specification by computing (and removing) minimal hitting sets on the minimal unsatisfiable cores. As an example, removing one between* $\{\varphi_3, \varphi_4\}$ *would be sufficient to make the specification satisfiable, as it would avoid a and b "chasing each other".*

As in other constraint systems, minimal unsatisfiable cores play an important role in debugging and diagnosis of declarative specifications.

3.3 Propositionalization

A formula φ is said to be in *negation normal form* if negations involve only atomic subformulae of φ. Informally, φ can be translated in negation normal form by

"pushing down the negation" exploiting dualities of logic operators.

A formula φ is said to be in *next normal form* (xnf) if it is in negation normal form and all its until (and release) operators appear under the scope of a next operator. An LTL_f formula can be converted into next normal form in linear time, applying the following transformation:

- $xnf(\alpha) = \alpha$, for $\alpha \in \mathscr{A}$;
- $xnf(\mathsf{X}\varphi) = \mathsf{X}\varphi$;
- $xnf(\mathsf{X}_\mathsf{w}\varphi) = \mathsf{X}_\mathsf{w}\varphi$;
- $xnf(\varphi \wedge \psi) = xnf(\varphi) \wedge xnf(\psi)$;
- $xnf(\varphi\ \mathsf{U}\ \psi) = xnf(\psi) \vee (xnf(\varphi) \wedge \mathsf{X}(\varphi \mathsf{U} \psi))$;
- $xnf(\varphi\ \mathsf{R}\ \psi) = xnf(\psi) \wedge (xnf(\varphi) \vee \mathsf{X}_\mathsf{w}(\varphi\ \mathsf{R}\ \psi))$;

An important property is that an LTL_f formula is satisfiable if and only if its next normal form is satisfiable [107]. SAT-based techniques for LTL_f satisfiability rely on the notion of *propositionalization*, that is encoding an LTL_f formula into a propositional formula whose atoms are subformulae of the original LTL_f formula. SAT-based approach to LTL_f satisfiability rely on next normal form to propositionalize formulae [107, 78].

3.4 Relation to automata

There exists a deep and rich relationship between automata theory and temporal logics. For what we are concerned in this thesis, we will be interested only in the notion of *deterministic finite state machine associated to an* LTL_f *formula*. For each LTL_f formula φ over $\mathscr{A}$ there exists a minimal finite-state automaton $\mathscr{M}(\varphi)$ over the alphabet $2^{\mathscr{A}}$ such that for any trace π it holds that $\pi \models \varphi$ iff π is accepted by $\mathscr{M}(\varphi)$ [51, 55]. Such an automaton can be obtained by applying standard techniques in the literature [51], which are doubly exponential in the worst case but quite effective in practice. In the rest of the thesis, whenever we refer to automata, we implicity assume they are deterministic finite state machines.

Under the assumption that each state of π contains exactly one symbol, $\mathscr{M}(\varphi)$ is an automaton over $\mathscr{A}$. (e.g., over an exponentially smaller alphabet). This assumption is extensively used in declarative process mining literature, where it is

referred to as *simplicity assumption* [68] or *Declare assumption*. We refer to $\mathrm{LTL_f}$ under simplicity assumption as *linear temporal logic over process traces* [68], $\mathrm{LTL_p}$.

Fionda and Greco [68] provide a comprehensive analysis of the complexity landscape for *syntactical fragments* of $\mathrm{LTL_f}$ and $\mathrm{LTL_p}$, that is, how the complexity of the satisfiability problem is affected whenever considering only formulae over a given set of temporal operators. We refer the interested reader to Theorems 27-35 of Fionda and Greco [68].

In particular, for what we are concerned in this thesis, it is important to notice that for arbitrary formulae (that is, unrestricted usage of temporal operators), $\mathrm{LTL_p}$ satisfiability reduces to $\mathrm{LTL_f}$ satisfiability [68]. Indeed, $\mathrm{LTL_p}$ simplicity assumption can be encoded into plain $\mathrm{LTL_f}$ by means of *"exactly one"* constraints over allowed atoms in each state. More formally, given an $\mathrm{LTL_p}$ formula φ over the alphabet $\mathscr{A}$, we can obtain the equisatisfiable $\mathrm{LTL_f}$ formula φ' defined as follows:

$$\varphi' = \varphi \wedge \underbrace{\Big(\bigwedge_{\{x,y\} \subset \mathscr{A}} \mathsf{G}(x \rightarrow \neg y) \wedge \mathsf{G}(y \rightarrow \neg x) \Big)}_{A} \wedge \underbrace{\mathsf{G}\Big(\bigvee_{a \in \mathscr{A}} a \Big)}_{B}$$

where subformula B ensures the absence of empty states (that is, each state must contain at least one atom in $\mathscr{A}$) and subformula A ensures that whenever an atom x appears in a state, $y \in \mathscr{A} \setminus \{x\}$ does not appear in the same state.

Chapter 4

Declarative Process Mining

This chapter provides some context on Process Mining. The first section discusses Process Mining, while the second section introduces Declare, the de facto standard language for declarative process mining.

4.1 Process Mining

Process Mining [1] is a research area at the intersection of Process Science and Data Science. It leverages data-driven techniques to extract valuable insights from operational processes by analyzing event data (i.e., event logs) collected during their execution. A *process* can be seen as a sequence of activities that collectively achieve a specific goal. A *trace* is a concrete execution of a process recording the exact sequence of events and decisions taken in a specific instance.

Process Mining plays a significant role in Business Process Management [155], by providing data-driven approaches for the analysis of *events logs* directly extracted from enterprise information systems. In this context, an *event* refers to the most fine-grained, atomic information that is available about the execution of a process, and associated metadata. Thus, a trace is a sequence of events that belongs to the same enactment of the process, and an event log is a collection of traces.

A *process model* is any mathematical abstraction which can be used to model a process. Several formalisms can be used in process modeling, with Petri nets [2] and BPMN [42] being among the most widely used, both following an imperative paradigm. Imperative process models describe all the valid process executions and can be impractical when the process under consideration is excessively intricate

Template	LTL_p
$Choice(a,b)$	$\mathsf{F}(a \vee b)$
$ExclusiveChoice(a,b)$	$\text{Choice(a,b)} \wedge \neg(\mathsf{F}a \wedge \mathsf{F}b)$
$RespondedExistence(a,b)$	$\mathsf{F}a \rightarrow \mathsf{F}b$
$Coexistence(a,b)$	$\text{RespondedExistence(a,b)} \wedge \text{RespondexExistence(b,a)}$
$Response(a,b)$	$\mathsf{G}(a \rightarrow \mathsf{F}b)$
$Precedence(a,b)$	$\neg b \,\mathsf{W}\, a$
$Alt.Response(a,b)$	$\mathsf{G}(a \rightarrow \mathsf{X}(\neg a \mathsf{U} b))$
$Alt.Precedence(a,b)$	$\text{Precedence(a,b)} \wedge \mathsf{G}(b \rightarrow \mathsf{X}_\mathsf{w}\text{Precedence(a,b)})$
$ChainResponse(a,b)$	$\mathsf{G}(a \rightarrow \mathsf{X}b)$
$ChainPrecedence(a,b)$	$\mathsf{G}(\mathsf{X}b \rightarrow a) \wedge \neg b$

Table 4.1: Some Declare templates as LTL_p formulae. We slightly edit the definitions for $ChainPrecedence(a,b)$ and $AlternatePrecedence(a,b)$, to align their semantics to the informal description commonly assumed in Process Mining applications. Changes w.r.t the original source are highlighted in red. The Succession (resp. AlternateSuccession, ChainSuccession) template is defined as the conjunction of (Alternate, Chain) Response and Precedence templates.

(known in literature as "spaghetti process"). In such cases, declarative process modeling [57] is a more appropriate choice. Declarative process models specify the desired properties (in terms of constraints) that each valid process execution must satisfy, rather than prescribing a step-by-step procedural flow. Using declarative modeling approaches allows for easy specification of the desired behaviors: everything that does not violate the rules is allowed.

Typical Process Mining tasks include: *Process Discovery*, which amounts to inferring a process model from an event log; *Conformance Checking* that aims at verifying if a trace is conformant to a specified model and, for logic-based techniques, *Query Checking* that evaluates *queries* (i.e., formulae incorporating variables) against the event log.

4.2 Declare

Declare [3] is a declarative process modeling language that consists of a set of *templates* that express temporal properties of process execution traces. The semantics of each Declare template is defined in terms of an underlying LTL_p formula. Table 4.1 provides the LTL_p definition of some Declare templates, as re-

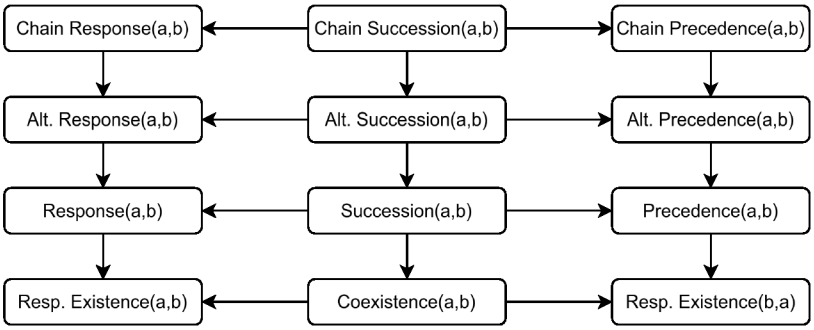


Figure 4.1: *Subsumption hierarchy* between Declare relation and existence templates. The arrow $t_1 \rightarrow t_2$ denotes that $\pi \models t_1 \implies \pi \models t_2$, e.g. t_1 is more specific than t_2. We also say that t_1 *subsumes* t_2.

ported in [49]. Declare templates can be classified into four distinct categories, each addressing different aspects of process behavior: *existence* templates, specifying the necessity or prohibition of executing a particular activity, potentially with constraints on the number of occurrences; *choice* templates, centered around the concept of execution choices as they model scenarios where there is an option regarding which activities may be executed; *relation* templates, establishing a dependency between activities as they dictate that the execution of one activity necessitates the execution of another, often under specific conditions or requirements; *negation* templates, modeling mutual exclusivity or prohibitive conditions in activity execution. In Table 4.1, $Choice(a,b)$ and $ExclusiveChoice(a,b)$ are examples of *choice* templates; while the others fall under the *relation* category. A Declare model is a set of *constraints*, where a constraint is a particular instantiation of a template, over specific activities, called respectively *activation* and *target* for binary constraints. Informally, the activation of a Declare constraint is the activity whose occurrence imposes a constraint over the occurrence of the target on the rest of the trace. A more formal account of activation-target semantics of Declare constraints can be found in Cecconi et al. [30].

Main Declare constraints. Declare constraints are arranged into "chains", that strengthen/weaken the basic relation templates $Response(a,b)$ and $Precedence(a,b)$. The $Response(a,b)$ constraint specifies that whenever a occurs, b must occur after, while the $Precedence(a,b)$ constraint states that b can occur only if a was executed before. The constraints $AlternateResponse(a,b)$ and $AlternatePrecedence(a,b)$ strengthen respectively the $Response(a,b)$ and

Precedence(a,b) constraints, imposing that involved activities must "alternate", e.g. once the activation occurs, the target must occur before the activation re-occurs. The constraint *ChainResponse*(a,b) imposes that the target must immediately follow the activation (that is, $\pi_t = a \rightarrow \pi_{t+1} = b$); the constraint *ChainPrecedence*(a,b) imposes that the target must immediately precede the activation (that is, $\pi_t = b \rightarrow \pi_{t-1} = a$). We can also weaken the temporal relation properties: the *Coexistence* relation states that two activities must occur together; the *ExclusiveChoice* states that exactly one of them must occur; *RespondedExistence* states that one implies the occurrence of the other. These templates do not specify temporal behavior, but only co-occurrences of events. Finally, the *Succession* template family combines, by means of propositional conjunction, *Response* and *Precedence* constraints. This hierarchy of constraints, graphically represented in Figure 4.1, suggests a progression from more general properties, at the bottom of the hierarchy, to more specific ones, at the top. As an example, the *ChainResponse*(a,b) is the most specific constraint in the *Response* chain; it implies *AlternateResponse*(a,b), which in turn implies *Response*(a,b), which implies *RespondedExistence*(a,b).

The following example showcases the informal semantics of the Response template.

Example 4.2.1 (Semantics of the Response template)**.** *The informal semantics for Response*(a,b) *is that* whenever a occurs in the trace, b will appear in the future. *Formally, the template is defined as* $G(a \rightarrow Fb)$. *Thus, if a occurs at time t in a trace* π*, for the constraint to be satisfied, b must appear in the trace suffix* $\pi_{t+1}, \ldots, \pi_n$. *In the context of a customer service process, let's consider the Response template instantiated with* $a =$ customer_complaints *and* $b =$ address_complaint*, corresponding to the template instantiation, i.e., the constraint,*

Response(customer_complaints,address_complaint)

Such constraint imposes that when a customer complaintt is received (activation activity), a follow-up action to address the complaintt (target activity), must be executed. On the other hand, the trace $\pi =$ (customer_complaints, logging_complaint, address_complaint, feedback_ collection) *satisfies the above constraint. In contrast, the trace* $\pi' =$ (customer_complaints, logging_complaint, address_complaint, customer_complaints, feedback_collection) *does not, since the second occurrence of* customer_complaints *is not followed by any* address_complaint *event.*

Another example shows how the *Response* constraint can be further specialized.

Example 4.2.2 (Semantics of the *AlternateResponse* and *ChainResponse* templates)**.** *Consider the execution traces $\pi_1 = aaabc$, $\pi_2 = abacb$, $\pi_3 = abab$. The constraint $Response(a,b)$ is valid over all these traces, while π_1 violates*

$$AlternateResponse(a,b)$$

because a repeats before b occurs after the first occurrence of a; The constraint $ChainResponse(a,b)$ is valid only on trace π_3, since both π_1 and π_2 have an a that is not immediately followed by a b.

Conformance and Query Checking Tasks. Let $\mathscr{L}$ be an event log (a multiset of traces) and $\mathscr{M}$ a Declare model. The *conformance checking* task $(\mathscr{L}, \mathscr{M})$ consists in computing the subset of traces $\mathscr{L}' \subseteq \mathscr{L}$ such that for each $\pi \in \mathscr{L}'$, $\pi \models c$ for all $c \in \mathscr{M}$. Basically, conformance checking determines whether a give process execution is compliant with a process model. This is the most basic form of conformance checking, that distinguishes between compliant and non-compliant traces. More advanced techniques capture the notion of conformance in a more fine-grained way, taking into account for example *how often* a constraint is violated [57].

Let $\mathscr{L}$ be an event log, and c a constraint. The support of c on $\mathscr{L}$, denoted by $\sigma(c, \mathscr{L})$, is defined as the fraction of traces $\pi \in \mathscr{L}$ such that $\pi \models c$. High support for a constraint is usually interpreted as a measure of relevance for the given constraint on the log $\mathscr{L}$. Given a Declare template t and a *support threshold* $s \in (0,1]$, the *query checking* task $(t, \mathscr{L}, s)$ consists in computing variable-activity bindings such that the constraint c we obtain by instantiating t with such bindings has a support greater than s on $\mathscr{L}$. Query checking enables discovering which instantiation of a given template exhibit high (above the threshold) support on an event log, and is a valuable tool to explore and analyze event logs [140].

Example 4.2.3 (Query checking and conformance checking)**.** *Consider the template $Response(a, ?y)$ (where $?y$ is a placeholder for an activity) over the log $L = \{abab, abac, abadabd\}$, and with support $\sigma = 0.5$. All the possible instantiations of the "partial template", which we obtain by substituting the placeholder $?y$ with an activity that appears in the event log, are the constraints $Response(a,b)$, $Response(a,c)$, and $Response(a,d)$, which yield respectively support $1, \frac{1}{3}, \frac{2}{3}$ over*

L. The substitution $?y = a$*, yields an unsatisfiable* LTL_f *formula, hence zero support. Thus, the query checking task* $(Response(a,?y), L, \sigma = 0.5)$ *admits*

$$\{Response(a,b), Response(a,d)\}$$

as answers. In fact, if we perform the conformance checking task of the constraint $Response(a,c)$ *the only compliant trace would be abac, with a support below 0.5.*

The recent survye of Di Ciccio and Montali [57] provides a starting point for Declare-related literature for the interested reader.

Part II

$\text{LTL}_\mathbf{f}$ Satisfiability in ASP

Chapter 5

Bounded Satisfiability

In this chapter we discuss an ASP-based approach to LTL_f bounded satisfiability, that improves on the SAT-based approach of the LTL2SAT system. The chapter closes with an experimental section which assesses the viability of the approach.

5.1 Introduction

When modeling system behavior in LTL_f, ensuring the model is satisfiable is crucial. A satisfiable model admits at least one valid system execution, guaranteeing the system can operate in the real world. Without satisfiability, the model would be considered flawed, as it would imply that the defined behaviors and constraints cannot be fulfilled. The concept of satisfiability in LTL_f translates in the existence of a finite sequence of events that satisfies all the temporal constraints and logical conditions imposed by the LTL_f formula.

Despite the intrinsic complexity of LTL_f satisfiability checking, that is PSPACE-complete for general formulas [55], in the last few years several efforts have been undertaken to design efficient solutions [68, 78, 107]. In fact, in the literature there are various approches to LTL_f satisfiability checking. Early methods relied on reductions to LTL satisfiability [55], but ad-hoc system specifically designed to deal with the finite trace semantics have proven to be faster. The main approaches include tableaux-based techniques and SAT-based methods. While the former are complete but often struggle with satisfiability, the latter are usually incomplete but faster, making them particularly suitable for use in a model checking-like fashion where counterexamples are sought up to a given horizon.

A notable SAT-based system for LTL_f satisfiability is LTL2SAT [68]. The

system encodes LTL formulas into SAT formulas, which are then solved using a SAT solver. The basic idea is to unfold LTL formulas over a finite horizon, by translating temporal operators into a series of propositional constraints that hold over each time step within the horizon.

In this thesis we present an alternative approach to this task that is based on Answer Set Programming (ASP) [21, 81]. As stated before, ASP is a key formalism in logic programming that has been many applications in fields like AI [65, 74], databases [116], and was also applied in industry [66, 88]. The new system, named `ltlf2asp`, is based on the ideas of LTL2SAT. In particular, we provide the following contributions:

- A uniform encoding of LTL_f satisfiability in ASP, that requires as input a factual representation of the LTL_f formula.
- Two optimization strategies not present in LTL2SAT, namely:
  - *(i)* A representation of the input formula as a directed acyclic graph, which avoids grounding the same subformula twice; and
  - *(ii)* An optimized search strategy which exploits the assumption that if a model of the input formula does not exist up to length k, then search is limited to models of length greater than k in the next solver calls.
- An implementation based on the CLINGO API [80].
- An experimental evaluation which demonstrates that `ltlf2asp` outperforms LTL2SAT and other state-of-the-art approaches.

It is worth noting that we propose a pure logic-based approach to LTL_f bounded satisfiability, in the sense that once formulas are rendered in ASP facts no external procedure is needed to generate propositional SAT clauses specific for the given instance (as in LTL2SAT). The first optimization is inspired to similar encoding optimizations used for passive learning of LTL_f formulae [29], and allows to produce a smaller ground program (w.r.t. the equivalent SAT encoding of LTL2SAT). Moreover, the second optimization is implemented in a natural way in ASP (it requires adding a small number of nonground rules), whereas it would require to extend the SAT encoder to add it to LTL2SAT.

5.2 Encoding $\mathrm{LTL_f}$ semantics in SAT

We recall the LTL2SAT [68] approach to bounded satisfiability of $\mathrm{LTL_f}$ formulae. The underlying idea is to encode an $\mathrm{LTL_f}$ formula φ in a propositional formula $\Phi_{\varphi,k}$ which is satisfiable if and only if φ admits a model of length at most k. At an intuitive level, the encoding consists of a bottom-up evaluation of φ, over a trace π which is implicitly defined by specific propositional variables associated with atomic formulae appearing in φ. Whenever $\Phi_{\varphi,k}$ is satisfiable, we can recover the trace π (a witness of satisfiability) from the propositional model of $\Phi_{\varphi,k}$. In the following, we will refer to π as "the trace being evaluated".

Let φ be an $\mathrm{LTL_f}$ formula, $\mathscr{A}$ the set of propositional symbols that appear in φ, k a positive integer (the upper bound on model length). Let $pt(\varphi) = (V, E, \lambda)$ be the (labeled) parse tree of φ, where $\lambda : V \mapsto \{\mathsf{X}, \mathsf{U}, \neg, \wedge\} \cup \mathscr{A}$ maps each node of $pt(\varphi)$ to a propositional symbol or logic operator. With a slight abuse of notation, we will refer to v as the subformula of φ which is rooted at the node v. The encoding makes use of two kinds of propositional variables. One, $s_v[i]$, which will be true if and only if the formula $v \in V$ is satisfiable at time i and one, $\ell[i]$, that will be true if and only if the length of π is less than or equals to i. In particular, the least i such that $\ell[i]$ is true represents the last state of π. Next, by exploiting XNF transformation rules, we define the propositional formula φ_v^i for each $v \in V$, according to the value of $\lambda(v)$:

- if $\lambda(v) = a \in \mathscr{A}$ then $\varphi_v^i = \bigwedge_{v' \in V, \lambda(v') = \lambda(v)} s_{v'}[i]$;
- if $\lambda(v) = \mathsf{X}$, then $\varphi_v^i = s_{v'}[i+1] \wedge \neg\ell[i]$, where v' is the child of v in $pt(\varphi)$ and $s_v[k-1] = false$;
- if $\lambda(v) = \wedge$, then $\varphi_v^i = s_{v_l}[i] \wedge s_{v_r}[i]$, where v_l and v_r are the left and right children of v in $pt(\varphi)$;
- if $\lambda(v) = \mathsf{U}$, then $\varphi_v^i = s_{v_r}[i] \vee (s_{v_l}[i] \wedge (s_v[i+1] \wedge \neg\ell[i]))$ where v_l and v_r are the left and right children of v in $pt(\varphi)$;
- if $\lambda(v) = \neg$ then $\varphi_v^i = \neg s_{v'}[i]$, where v' is the child of v in $pt(\varphi)$

Since the propositional variables $s_v[i]$ model satisfaction of formula v at time i, one can substitute each occurrence of v in the XNF transformation with the corresponding propositional variable $s_v[i]$ (and using $s_v[i+1]$ whenever v is under the scope of an X), and obtain a direct mapping of XNF rules into SAT.

Furthermore, we define the formula $\Phi_v = \bigwedge_{i=0}^{k-1}(s_v[i] \leftrightarrow \varphi_v^i)$. The formula Φ_v aligns the truth value ($\leftrightarrow$) of $s_v[i]$ with temporal semantics of $\lambda(v)$, for each available time-point. Let *root* be the root node of $pt(\varphi)$. We define Φ_k as follows:

$$\Phi_{\varphi,k} = \bigwedge_{v \in V} \Phi_v \wedge s_{root}[0] \wedge \ell[k-1] \wedge \bigwedge_{i=0}^{k-2} \ell[i] \rightarrow \ell[i+1]$$

If $\Phi_{\varphi,k}$ is satisfiable, then φ admits a model of length at most k. In this case, we can also recover the satisfying trace π by projecting the model of Φ_φ, k on the variables $s_v[i]$ such that $\lambda(v) \in \mathscr{A}$, and in particular $\sigma_i = \{x : s_v[i] \in \mathscr{M}\}$. If Φ_φ, k is unsatisfiable, it means that it does not admit a model of length up to k.

5.3 Encoding $\mathrm{LTL_f}$ semantics in ASP

We now report on how to model in ASP the semantics of $\mathrm{LTL_f}$.

Formulae & traces. Consider a formula φ, and suppose each node in $pt(\varphi)$ is assigned a unique integer. We can reify φ into ASP facts by means of predicates $until/3$, $negate/2$, $conjunction/2$, $atomic/2$, $next/2$, $true/1$, where the first term always refers to the unique identifier of the subformula and the other terms refer to identifiers of the most immediate subformulae, and predicate names are self-explanatory. The root of φ is identified by 0. We denote this set of facts as $[\varphi]$. For example, the formula $\neg a \mathsf{U} (\mathsf{X} b)$ would be encoded as: `until(0,1,2). negate(1,3). atomic(3,a). next(2,4). atomic(4,b). root(0).`

A trace π can be encoded by means of facts $trace(t,a)$ if $a \in \sigma_t$, and $time(t)$ if $t < |\pi|$, where $trace(t,a)$ holds if $a \in \pi(t)$, and $time(i)$ if $|\pi| = k$ and $0 \leq i \leq k-1$. For example, the trace $\{a,b\} \cdot \{\} \cdot \{c\}$ is encoded as: `time(0..2). trace(0,a). trace(0,b). trace(2,c).` We denote this set of facts as $[\pi]$.

Semantics of temporal operators. The ASP encoding of temporal operators makes use of predicates with signature $holds/2$, $trace/2$, and $time/1$. An atom $holds(x,t)$ models that the subformula rooted in x is true at time t (it has the same role of variable $s_x[t]$ in the SAT encoding). The atom $time(i)$ states that σ_i is a valid state in the trace, similar to the role of $\ell[i]$. The following logic program Π models the semantics of temporal operators:

```
holds(T,X) :- next(X,F), holds(T+1,F).
holds(T,X) :- conjunction(X,_), time(T), holds(T,F): conjunction(X,F).
```

```
holds(T,X) :- negate(X,F), time(T), not holds(T,F).
holds(T,X) :- until(X,F,G), holds(T,G).
holds(T,X) :- until(X,F,G), holds(T,F), holds(T+1,X).
sat :- holds(X,0), root(X).
```

Observe that each rule closely mirrors the corresponding XNF rule, so correctness follows from previous results [68]; since Π is a locally stratified logic program [79], it admits a unique answer set containing *sat* iff $\pi \models \varphi$.

Search for a model. The above encoding evaluates the formula φ over a trace π, so it is sufficient to replace the set of facts $[\pi]$ by some choice rules that generates all possible traces to lift it for checking bounded satisfiability. Consider the following logic program:

```
symbol(X) :- atomic(_,X).
{ last_instant(T): T=0..k-1 } = 1.
time(0..T) :- last_instant(T).
{ trace(T,A): symbol(A) } :- time(T).
:- not sat.
```

Above, `k` is a runtime CLINGO constant which denotes the search bounds. The first choice rule yields the length of the guessed trace, $last\_instant/1$, whereas the second one generates all possible states (given symbols that appear in φ), for each time point, thus all possible traces of length up to k. The constraint discards traces which are not models of φ. If the above logic program is satisfiable, projecting answer sets on predicates $trace/2$, $time/1$ allows one to obtain satisfying traces.

Example 5.3.1. *The answer set* $\{trace(0,a), trace(0,b), trace(2,c), time(0), time(1), time(2)\}$ *is to be interpreted as the trace* $\pi = \{a,b\} \cdot \{\} \cdot \{c\}$. *The atom* $time(1)$ *identifies the empty state* $\pi_M(1)$ *in the absence of* $trace(1,\_)$ *atoms.*

5.4 Implementation and Experiments

In this section we report on the design of our system, and discuss the result of the experimental evaluation.

5.4.1 Implementation

The tool `ltlf2asp` is implemented in Python 3, using few external dependencies, mostly relying on the CLINGO Python APIs. The architecture is minimal and

streamlined, consisting of a parsing component (which allows to customize a reification strategy to map φ onto a set of facts, possibly applying rewriting to certain temporal operators), and solving module which currently uses CLINGO as a backend solver. Solving is based on a exponential search [128] progression strategy which expands the upper bound on model length k progressively doubling up the horizon at every solving call.[1] This yields the following semi-decision procedure:

```
def solve(f, k):
    # f: Input formula
    # k: Upper bound on model length
    b = 1
    while True:
        # Exceeds horizon upper bound
        if b > k:
            return False
        if solve_up_to_length(f, b):
            return True
        # Double up search horizon
        b = b * 2
```

where `solve_up_to_length` is understood to be a procedure which attempts to find a model of length at most b (such as the SAT or ASP approaches). If such a model is not found, the search horizon expands (doubles). This is performed iteratively, up to the point where b is greater than an input upper bound k. In the worst case, this performs $\lceil \log(k) \rceil$ calls to the subroutine `solve_up_to_length`.

With respect to the original implementation of the LTL2SAT system, we provide the following improvements.

Formula representation. We improve upon LTL2SAT by encoding the input formula as a directed acyclic graph rather than a tree, such that each *unique* subformula is encoded once. As an example, consider the formula $(\mathsf{X}a) \wedge (\mathsf{XX}a)$. Reifying it as a tree yields the facts $root(0)$, $conjunction(0,1)$, $conjunction(0,2)$, $next(1,3)$, $atomic(3,a)$, $next(2,3)$, $next(3,4)$, $atomic(4,a)$; while reifying it as a directed acyclic graph yields the facts $root(0)$, $conjunction(0,1)$, $conjunction(0,2)$, $next(1,3)$, $atomic(3,a)$, $next(2,1)$. Observe that this improvement might also be applied in SAT-based solvers, as it was for example done in [29], however, it is missing from LTL2SAT.

[1] In general, it is not possible to compute a priori such an upper bound k to guarantee being able to prove satisfiability on satisfiable formulae, although this is possible for some syntactical fragments [68] and other restricted conditions.

Optimizing model search. Each time `ltlf2asp` doubles the search horizon it means no solution exists up to the previous upper bound. We exploit this observation by adopting a slight modification of the logic program described in 5.3.

```
{ last_instant(T): T=a..b-1 }.
time(0..T) :- last_instant(T).
{ trace(T,A): symbol(A) } :- time(T).
:- not sat.
```

where `a`, `b` are runtime CLINGO constants, which denotes the minimum length of the search models and its upper bound. Thus, we assume the length of the searched trace is at least a. Although this change can be easily achieved in SAT as well, doing so requires editing the SAT encoder, while in ASP it only requires adding a few (non-ground) rules in a logic program.

Input format & availability. Concerning `ltlf2asp` input format, we take a *pragmatic approach* to ensure compatibility with common LTL_f textual formats. Also, to foster inter-operability with other tools, we adopt a modern JSON output format which aligns with the BLACK solver [78]. The code for our system is publicly available on GitHub, `https://github.com/ainnoot/ltlf2asp`[2].

5.4.2 Empirical analysis

Our experiments wish to investigate whether an ASP-based approach to bounded satisfiability checking is competitive with state-of-the-art SAT-based systems.

Data. We benchmark the systems over three sets of formulae. Two come from well-established LTL_f solving benchmarks in literature [68, 107, 108]. The first, which we refer to as *Patterns*, is composed of frequent *modeling patterns* that arise in model checking and temporal specifications [61]. These formulae are commonly used to assess LTL_f solvers, despite most of them admit models of length one (and, thus are easy). The second one, *Models*, consists of more complex LTL_f specifications which are obtained by combining small formulae (actually, the same ones as in the *Patterns* dataset). For the third set of formulae, we provide a generator (available here, `https://github.com/ainnoot/ltlp-xes-bench`) which yields formulae that are similar to *Models*, but consists of instances with increasing minimum model length. These are obtained by

[2] `ltlf2asp` is under active development. Refer to contents in the `lpnmr2024` branch for artifacts related to this chapter.

adding the top-level conjunct $\mathsf{X}^k\top$ to guarantee models have a minimal length of $k+1$. We use this set to study how systems deal with longer models. Our approach can check satisfiability, thus we focus on satisfiable instances.

Compared systems. From the literature we know that ad-hoc systems for $\mathrm{LTL_f}$ represent the state of the art. For this reason, we consider: AALTAF, BLACK and LTL2SAT. Regarding BLACK, we run it both in semi-decision mode disabling unsatisfiability checks (for a fair comparison with LTL2SAT and `ltlf2asp`), and standard mode which performs unsatisfiability checks [78]. This is to ensure a *fair* comparison between BLACK, that is complete, and ltlf2asp that is a semi-decision procedure.

Execution environment. Experiments are executed on a server running Ubuntu 22.04 LTS, with a i7-13700 CPU and 32GB RAM. All measurements are the average of 10 runs, with a timeout of 5 minutes and 3GB of RAM. All instances were executed sequentially. We use Python 3.12 and CLINGO 5.4, and latest available versions of BLACK 10.6, AALTAF and LTL2SAT systems.

Discussion of the Results. We report results as cactus and scatter plots in Figure 5.3-5.4-5.5. In cactus plots, there is a point (i,y) whenever a solver solves i instances in y seconds. In scatter plots a point (x,y) denotes that a certain instance was solved in x seconds by a solver, and y units by a "reference" solver. In all scatter plots, the reference system is `ltlf2asp` (y-axis), thus, points above the bisector are instances where the alternative system is faster than `ltlf2asp`. Different colors distinguish different solvers.

First we observe from Figure 5.3 that all instances in the *Patterns* are nowadays trivial, resulting in sub-second solving times for all solvers. Minor performance differences (only visible in logscale) are likely due to the implementation language. BLACK and AALTAF are written in C++, and have lower loading times than alternatives based on Python (i.e., `ltlf2asp`) and Java (i.e., LTL2SAT).

On the other hand, in *Models* dataset a more interesting behavior emerges. We see from the cactus plot in Figure 5.4 that AALTAF is overall the slowest system, while other tools are mostly on par with each other, and `ltlf2asp` is slightly faster. Note also that the similar performance of the semi-decision version and standard version of BLACK hints at the fact that indeed these instances are still easy. Manually inspecting the hardest instances in *Models*, we conjecture the hardness is mostly due to the number of involved propositional symbols and

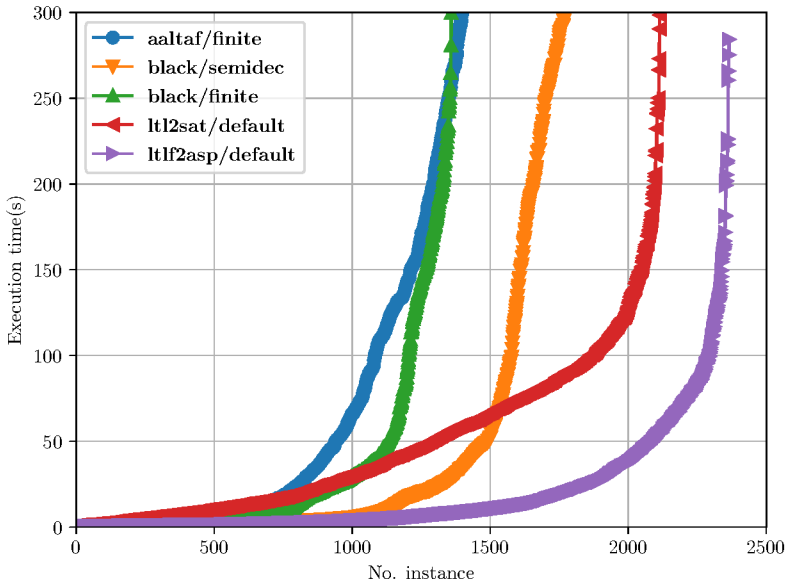


Figure 5.1: Global performance on *Synthetic* formulae. A point (x, y) in the scatter plot denotes that a certain instance was solved in y seconds by the `ltlf2asp` system, and x seconds by the other system (according to the color).

(minimum) model length. In order to better assess scalability of the different systems, we generate synthetic instances in the same way as the *Models*, that is, by considering conjunctions of Declare [43] patterns extracted from real-world event logs, but controlling minimum model length. In particular, given an event log (a collection of traces), we extract a Declare [43] process model using PM4PY [16], which consists of a conjunction of Declare patterns. We sample a set of satisfiable patterns by forcing a lower bound on minimum model length by adding a top level conjunction $\mathsf{X}^k(\top)$ — which forces a model length k. We select values of $k \in \{16, 32, 64, 128, 256, 512\}$. As a reference, the longest model in *Models* has length 64. For each k and event log, we sample 5 different formulae of each length. We generated a total of 2500 formulae.

On *Syntethic* instances SAT-based and ASP-based model search have a clear advantage (see Figure 5.1 and Figure 5.2), both on a global level (Figure 5.1) and instance-wise level (Figure 5.2). We investigated this in detail in Figure 5.5, that compares `ltlf2asp` with other systems, by reporting a plot for each values of $k \in \{16, 32, 64, 128, 256, 512\}$ (i.e., partioning measurement data on minimum

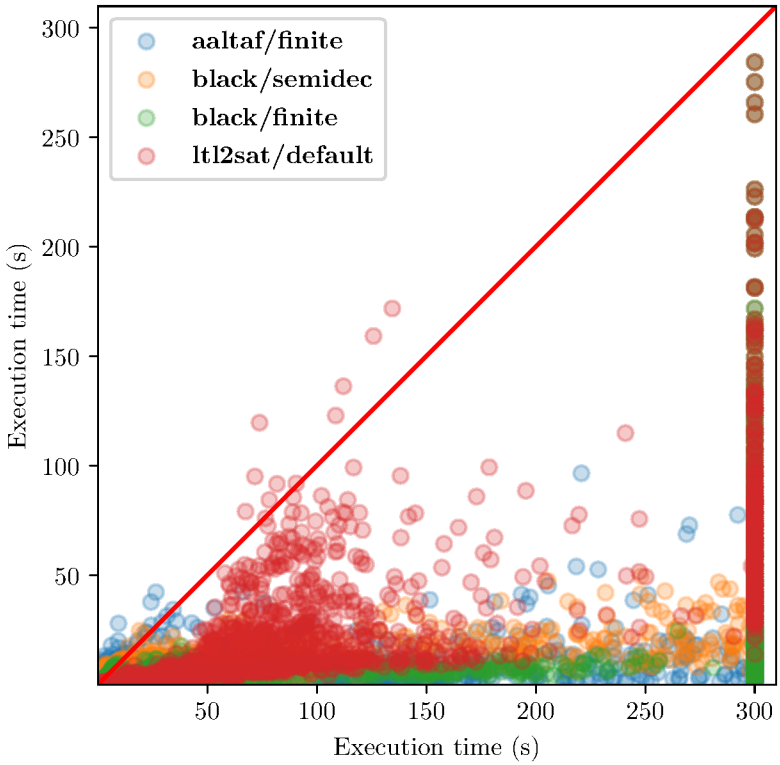


Figure 5.2: Instance-wise comparison on *Synthetic* formulae. A point (x, y) in the scatter plot denotes that a certain instance was solved in y seconds by the `ltlf2asp` system, and x seconds by the other system (according to the color).

enforced model length) for the formulae in *Synthetic*. Expectedly, the performance gap between systems *widens* as the lower bound on model length k increases. Indeed, `ltlf2asp` is preferable already on instances with $k \geq 128$ (cfr., Figure 5.5(d-f)). Overall, `ltlf2asp` is the most efficient, followed by LTL2SAT (cfr., Figure 5.1), and BLACK semidecision.

We conjecture a major component in this performance gap is likely due to the progression-like strategy adopted by LTL2SAT and `ltlf2asp`, which reach an adequate horizon length to find a model in $\lceil \log k \rceil$ solving calls, where AALTAF and BLACK require more calls to the underlying SAT solvers. However, this sort of performance gap is consistent with previous studies [78] and has been highlighted on satisfiable instances. As expected, the semi-decision version of BLACK is

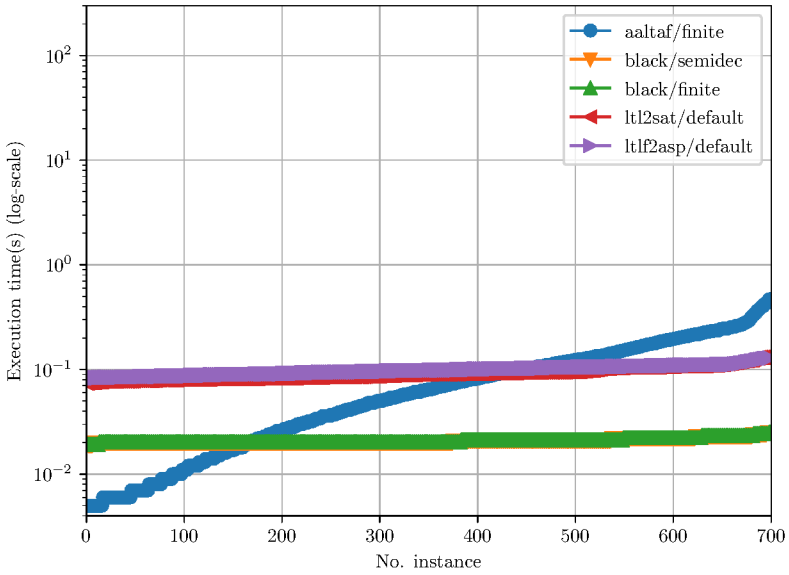


Figure 5.3: Cactus plot on *Patterns*. The x axis reports the number of solved instances within y seconds by a given system (according to the color).

more suitable to (longer) instances, being able to handle some formulae with a model of length 256 (Figure 5.5e, where the BLACK decision version can solve none of these). The advantage of `ltlf2asp` w.r.t. LTL2SAT is most likely due to the DAG-based formula representation that generates less redundant propositional representations and reduce search space avoiding searching for nonexistent short candidate traces.

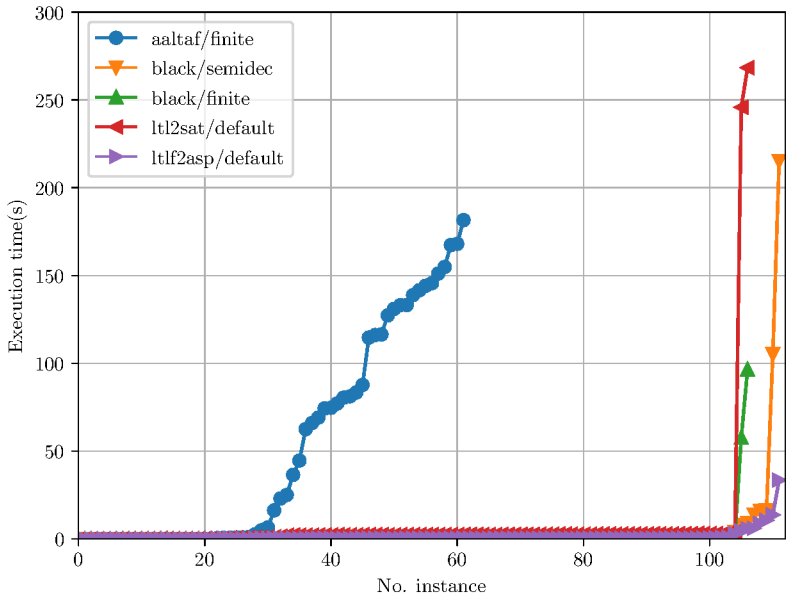
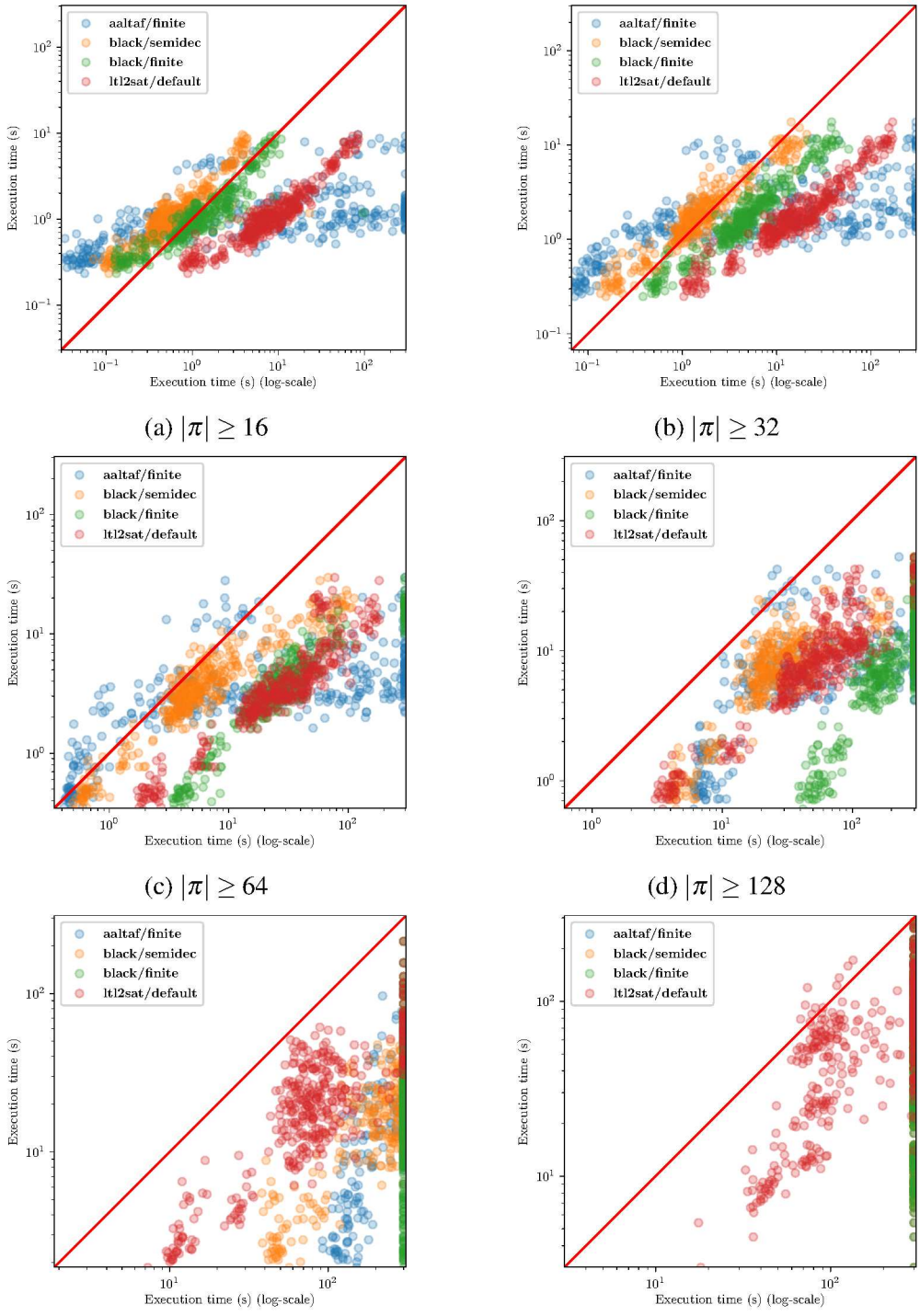


Figure 5.4: Cactus plot on *Models*. The x axis reports the number of solved instances within y seconds by a given system (according to the color).



(a) $|\pi| \geq 16$ (b) $|\pi| \geq 32$

(c) $|\pi| \geq 64$ (d) $|\pi| \geq 128$

(e) $|\pi| \geq 256$ (f) $|\pi| \geq 512$

Figure 5.5: Behavior of the solvers while increasing minimum model length. $|\pi| \geq 2^k$ denotes that $\mathsf{X}^k\top$ is a top-level conjunct of each formula in the set. A point (x,y) in the scatter plot denotes that a certain instance was solved in y seconds by the `ltlf2asp` system, and x seconds by the other system (according to the color).

Chapter 6

Enumerating LTL_f MUSes

In this chapter we propose an ASP-based approach to enumerate minimal unsatisfiable cores of LTL_f formulae in conjunctive form. Our approach is inspired by domain-agnostic unsatisfiable cores enumerations, and exploits the interaction between an ASP solver and an LTL_f solver. We start by formally characterizing our approach, then delve into an experimental analysis on the performances of our system.

6.1 Introduction

As the satisfiability problem plays an important role in LTL_f reasoning, in applications the problem of detecting and addressing *unexpected unsatisfiability* plays an important role. That is, "understanding why" a specification we *assume to be satisfiable*—that could encode, for example, the requirements for a system—is, actually, unsatisfiable. Despite LTL_f expressions being quite readable, as specifications become bigger—especially as they are automatically mined from event logs [1, 57], it is possible 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 formulas in the specification that are responsible for them [132, 144]. Specifically, we are interested in computing the *minimal unsatisfiable cores* (MUCs): subset-minimal subsets of formulas (from the original specification) that are collectively inconsistent [109, 132, 144]. 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 [125, 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) [21, 81], where the goal is to find *minimal unsatisfiable subsets* (MUS) of atoms that make an ASP program incoherent [22, 124, 8]. In recent years, efficient implementations of MUS enumerators have been presented [8].

Our goal in this chapter is to take advantage of both the declarativity of the ASP language and the efficiency of ASP systems to also enumerate MUCs of LTL_f formulas. Hence, we present a new transformation which constructs, given a set of LTL_f formulas, an ASP program whose MUSes are in a biunivocal corrispondence with the MUCs of the original specification. Importantly, although we base our reduction on a well-known encoding of LTL_f bounded satisfiability [68, 70], the idea is general enough to be applicable to other decision procedures, as long as it 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 [128]. To the best of our knowledge, we provide the first MUC enumerator for LTL_f.

We empirically compared our implementation with existing systems designed to produce only *one* MUC (or just one potentially non-minimal unsatisfiable core) [132, 144] and observed that our system—despite being more general—is competitive against those on established benchmarks from the literature. Importantly, MUC enumeration is very efficient as well.

The technique proposed in this chapter 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 an approach, a formal connection between these objects must be established. In this chapter, we introduce the notion of *probe* and *k-MUC* to investigate this relationship. A probe is an abstraction over the class of logic programs—that generalize the bounded satisfiability approach we discussed in Chapter 5—with suitable properties to apply the approach herein presented; *k*-MUCs are a *relaxation* with respect to model length of the concept of MUC, which reveals to be more suitable for ASP-based reasoning.

6.2 MUS and Probes

In the rest of the chapter we adopt the notation introduced in in [132]. 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 first assumption is that there exists a uniform way to encode LTL_f formulae in conjunctive normal form into logic programs. In particular, we are interested in *encodings where original conjuncts of φ can be told apart by means of special atoms*. More formally:

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

Each subset $S \subseteq \mathscr{O}(\varphi)$ uniquely identifies the set of conjuncts $\psi = \{\phi_i : phi(i) \in \mathscr{O}(\varphi)\} \subseteq \varphi$. Therefore, we denote ψ by $\mathsf{Formula}(S)$ and S by $\mathsf{Atoms}(\psi)$.

Reification functions enable to encode LTL_f formulae into logic programs. Since in this chapter we are concerned with notions of *satisfiability*, *unsatisfiability* with respect to *subset minimality* of LTL_f formulae, among all possible reification functions, we are interested in ones that preserve as much information about these properties. In particular, we 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 φ if for each set $S \subseteq \mathscr{O}(\varphi)$ 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 $S = M_{|\mathscr{O}(\varphi)}$.*

There exist multiple ASP encodings that satisfy the definition of probe. Intuitively, one can obtain a probe by adapting any ASP encoding for bounded LTL_f satisfiability [70, 68]. Section 6.4 features an extended and detailed example. Here, we focus on how probes relate to MUCs of φ.

Lemma 6.2.1. *Let ρ be a probe of depth k for φ. Let S be a minimal unsatisfiable subset of $\rho(\varphi)$ wrt the objective atoms $\mathscr{O}(\varphi)$. Then $\mathsf{Formula}(S)$ is either an MUC of φ or it is satisfiable but its shortest satisficing trace has length greater than k.*

Proof. Assume S is a minimal unsatisfiable subset wrt $\mathscr{O}(\varphi)$. 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. □

We provide an example.

Example 6.2.1. *Consider the formula $\varphi = \{X^5\beta, X^5\neg\beta\}$. This formula has a unique MUC, namely $X^5\beta \wedge X^5\neg\beta$. If we consider a probe $\rho_3 = \rho(3,\varphi)$, it has two MUSes, namely $\{X^5\beta\}, \{X^5\neg\beta\}$, since these formulae do not admit models of length at most 3. If we consider instead probes of depth at least 5, it is now possible to detect the MUC through the (unique) MUS $\{X^5\beta, X^5\neg\beta\}$*

Formulae exhibiting the property shown in the statement of Lemma 6.2.1 are the key object that allow us to levarage MUS enumeration to enumerate MUCs. Thus, we introduce a definition:

Definition 3 (k-bound MUC)**.** *Let $k \in \mathbb{N}$. A k-bound MUC (k-MUC) for 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 6.2.2. *Let $S \in \mathsf{MUC}^k(\varphi)$. If S is unsatisfiable, then S is a MUC for φ.*

Proof. Follows from the fact that since $S \in \mathsf{MUC}^k(\varphi)$, it means that any proper subset of S admits a model of length at most k, hence it is satisfiable. If S is also unsatisfiable, it matches the definition of MUC. □

We re-state Lemma 6.2.1 adopting the new definition:

Lemma 6.2.3. *Let φ be a formula, $k \in \mathbb{N}$. S is a minimal unsatisfiable subset of the k-probe $\rho(\varphi)$ if and only if $\mathsf{Formula}(S)$ is a k-MUC for φ.*

Example 6.2.1 highlights an interesting property. The probe at depth $k = 3$ yields two (singleton) MUSes, that intepreted as formulae indeed 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 actually both satisfiable if we consider models of length at most k', but still (jointly) unsatisfiable considering models of length at most k'. Intuitively, this makes the probe at depth k' *more effective*, since it allows to *discard* MUSes that won't lead to a 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 the following theorem, which establishes a bijection between MUSes of complete probes and MUCs of φ:

Theorem 1. *Let ρ be a complete probe for φ. Then S is an MUS of P wrt $\mathscr{O}(\varphi)$ if and only if $\mathsf{Formula}(S)$ is a MUC of φ.*

Proof. ($\rightarrow$) Let S be an MUS of ρ wrt $\mathscr{O}(\varphi)$. By Lemma 6.2.3 $\mathsf{Formula}(S)$ is a k-MUC, thus it is either unsatisfiable (hence it is a MUC); or satisfiable with a satisficing trace with length greater than k—in this latter case, ρ would not be a complete probe. Hence, $\mathsf{Formula}(S)$ must be a 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 $\mathrm{LTL_f}$ 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 $\mathsf{Atoms}(\psi^j)$, but there exists no answer set that extends $\mathsf{Atoms}(\psi)$. Since $\mathsf{Atoms}(\psi^j) = \mathsf{Atoms}(\psi) \setminus \{phi(j)\}$, this shows that $\mathsf{Atoms}(\psi)$ is an MUS for $\rho(\varphi)$ wrt $\mathscr{O}(\varphi)$. □

Theorem 1 characterizes MUCs of φ as MUSes of complete probes for φ. From the standard automata-based procedure for deciding satisfiability in $\mathrm{LTL_f}$ [55, 114] it follows that every satisfiable formula φ has a model of length at most 2^n where n is the number of subformulas of φ. Indeed, completeness threshold is bounded above by the upper bound on model length of φ (due to monotonicity of $\mathrm{LTL_f}$ wrt conjunction —adding a conjunct can only *increase* the length of the shortest model). However, in practice, it can be much smaller, as we will see in the experiments section.

6.3 MUC enumeration by MUS enumeration

Applying Theorem 1 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. However, by Lemma 6.2.2, we also know that *some* k-MUCs, with $k \leq h(\varphi)$ could also be MUCs. These results suggest two anytime algorithms that

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' = 0:
8:             mucs.append(ψ)
9:     return mucs
```

could be useful in the realm of LTL_f MUC enumeration: *(i)* an algorithm (see Algorithm 1) that computes all MUCs among the k-MUCs for a given k and *(ii)* an iterative deepening variant of Algorithm 1 (see Algorithm 2) which expands the probe depth k whenever a k-MUC reveals not to be unsatisfiable. Algorithm 1 and 2 provide pseudo-code for such approaches. Both algorithms make use of the subroutines `probe`, `enumerate_mus`, `to_formula`, `check_satisfiability`, that are explained next.

`probe`(φ, k) builds the logic program from which we will extract k-MUCs. This is the counterpart of $\rho(k, \varphi)$.

`enumerate_mus`(P) invokes an ASP solver to extract MUSes of the probe P wrt the objective atoms Φ;

`to_formula`(x) given an MUS x of P, rebuilds the LTL_f formula $\mathsf{Formula}(x)$;

`check_satisfiability`(ψ) determines wheter an LTL_f formula is satisfiable or not; if ψ is satisfiable, returns the length of a satisfying trace; otherwise, it returns 0;

We remark both algorithms are compatible with any ASP solver that implements MUS enumeration (that is, an implementation of the procedure `enumerate_mus`) and (complete) LTL_f solvers that can *(i)* provide a satisficing 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, that

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 x ∈ mucs:
11:                 skip
12:             k' = check_satisfiability(ψ):
13:             if k' = 0:
14:                 mucs.append(ψ)
15:             else:
16:                 k = k'
17:                 complete = False
18:                 break
19:     return mucs
```

can be *certified* or *disproved* by a call to an $\mathrm{LTL_f}$ satisfiability oracle. Following such a call, we discard *false positives* candidate (that is, k-MUCs that are actually satisfiable) as we encounter them. This approach does not guarantee to 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. Whenever we encounter a false positive k-MUC ψ, this 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.

6.4 A Concrete Example of Probe

Probes can be obtained with slight modifications from any ASP encoding to perform bounded satisfiability of $\mathrm{LTL_f}$ formulae. In this section, we show how to

obtain a probe from the approach proposed in Chapter 5, by means of an example.

Example 6.4.1. *Consider the formula $\varphi = (a \wedge \neg b) \wedge (c \ U \ b)$ with two conjuncts is encoded through the facts:*

```
conjunction(0, 1). conjunction(0, 2).
conjunction(1, 3). conjunction(1, 4).
atom(3, a). negate(4, 5). atom(5, b).
until(2, 6, 5). atom(6, c).
root(0).
```

Without loss of generality, we assume root of formulae to be always assigned the identifier 0, and denote by $[\varphi]$ the set of facts that encodes φ.

The probe. Recall the bounded satisfiability encoding of Chapter 5 evaluates whether φ admits a model of length up to k. In order to comply with Definition 2 of probe, we have to make it so that the logic program yields an answer set *for each subset of φ that admits a model of length up to k*. This is obtained by replacing the facts in $[\varphi]$ corresponding to *top-level conjunctions* with rules with *objective atoms* in the body:

```
conjunction(1, 3). conjunction(1, 4).
atom(3, a). negate(4, 5). atom(5, b).
until(2, 6, 5). atom(6, c).
root(0).

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

As we can see, the only affected rules are the `conjunction` facts at the root level. Intuitively, the additional choice rule over $phi/1$ atoms *enables* or *disables* conjuncts of φ. This is reminiscent of how logic programs under stable model semantics are *annotated* to exploit MUS enumeration for debugging purposes or to compute paraconsistent semantics [8].

Finally, note that this is only an example, and probes may have many different forms. For example, a probe P could encode the tableaux for LTL_f [142, 78], or ad-hoc encodings for syntactical fragments of LTL_f such as Declare [39].

Benchmark	#Inst.	#Compl.	Sum of MUCs		Probe depth			MUC Size		
			Compl.	TO	Min.	Med.	Max.	Min.	Med.	Max.
acacia.demo-v3	11	11	77	-	1	1	1	2	2	2
alaska.lift	129	129	8310	-	1	1	5	1	3	5
forobots.forobots	38	38	38	-	1	1	2	2	2	2
schuppan.O1formula	27	27	27	-	2	2	2	2	2	2
schuppan.O2formula	27	27	27	-	1	1	1	2	60	1000
trp.N12x	400	400	14380	-	1	1	1	1	1	1
trp.N5x	240	240	4210	-	1	1	1	1	1	1
anzu.amba	34	2	362	20642	5	9	9	1	8	41
anzu.genbuf	36	6	1043	16119	6	6	6	1	4	9
rozier.counter	76	62	514	35	4	23	41	2	4	5
schuppan.phltl	13	9	219	2760	1	1	1	2	3	3
trp.N12y	67	3	2514	116365	14	14	14	24	34	42
trp.N5y	46	33	121704	287380	7	7	7	13	16	20
LTLfRandomConjunction.C100	500	134	15636	1295793	1	6	14	2	9	33
LTLfRandomConjunction.V20	435	152	64049	1619065	1	6	14	2	11	26

Table 6.1: Complete MUC enumeration of the different formula families. #Inst is the number of instances for each family. A benchmark *x.y* denotes that the set of formulae *y* is a *family* of benchmark *x*.

6.5 Experiments

This section presents an experiment conducted to empirically evaluate the performance of our system `mus2muc`. We performed different experiments addressing the following issues:

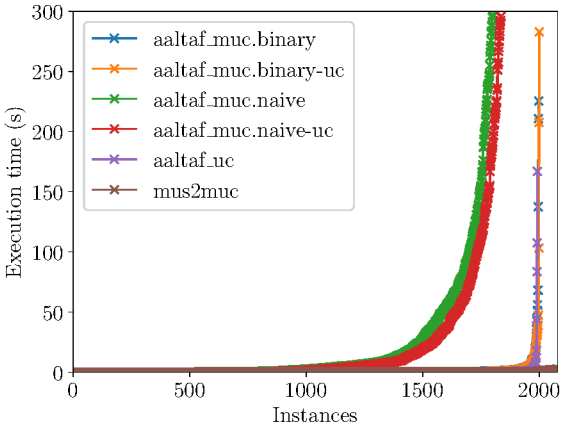
I **Extraction of Single MUC**: *How does `mus2muc` perform in computing a single MUC?*

II **Enumeration of MUCs**: *How effective is `mus2muc` in enumerating* LTL_f *MUCs?*

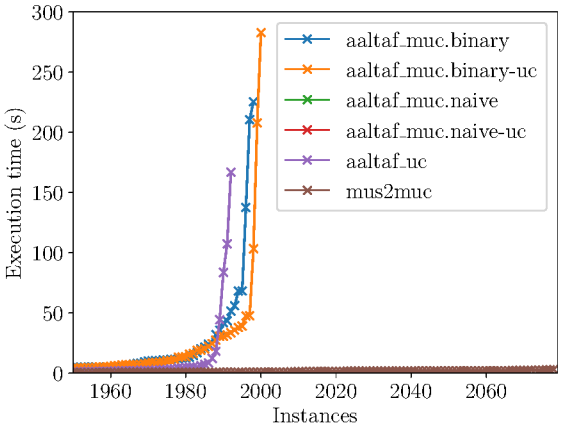
III **Generation vs. Certification**: *How does MUSes generation and* LTL_f *satisfiability checks affect the overall performance of `mus2muc`?*

IV **Domain agnostic MUCs enumeration techniques**: *How does `mus2muc` compare with SAT-based MUCs enumeration techiques, suitably adapted from LTL to* LTL_f *domain?*

In what follows, we describe the implementation of our system, and then shift the attention to an analysis aimed at answering the above questions.

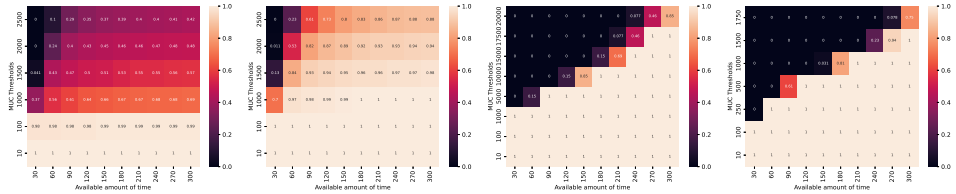


(a) All instances.



(b) Focus on the hardest instances (i.e., $x > 1950$).

Figure 6.1: Computation of a single MUC or UC.



(a) C100 RndConj. (b) V20 RndConj. (c) TRP5y. (d) TRP12y.

Figure 6.2: Percentage of not-fully-enumerated instances (among the formula families RndConj C100, RndConj V20, TRP5y and TRP12y) 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.

Implementation. The implementation of `mus2muc` closely follows the pseudo-code in Algorithm 2. In particular, our implementation uses the ASP solver `wasp` [59, 6] as a MUS Generator, and the LTL_f solver `aaltaf` as a satisfiability solver. In more detail, the solver `wasp` takes as input the probe described in the previous section, and then performs the MUS enumeration. 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 in the previous section, our system is anytime, and outputs MUCs as soon as they are certified at the smallest k that allows to do so. Our implementation[1] uses Python 3.12, the version of `aaltaf`[2] and `wasp`[3] available on authors' public repositories.

Systems. For the single MUC extraction task, we compare with the `aaltaf-muc`[4] system, which computes a single minimal unsatisfiable core, in four configurations as described in [132]. We also include `aaltaf-uc`[5], which

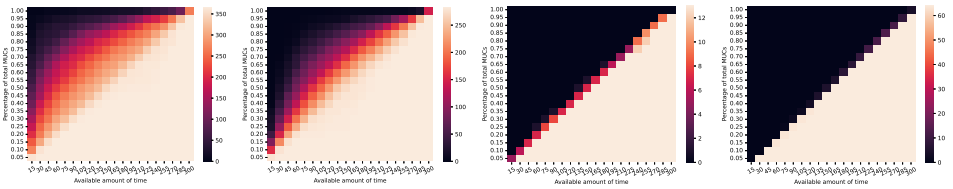
[1] Code will be made available upon request to the authors.

[2] https://github.com/lijwen2748/aaltaf; `858885b`

[3] https://github.com/alviano/wasp; `f3e4c56`. Logging facilities for `mus2muc` require a patch available in our repository.

[4] https://github.com/nuutong/aaltaf-muc; `9b40837`

[5] https://github.com/roveri-marco/aaltaf-uc; `b6aeb5c`



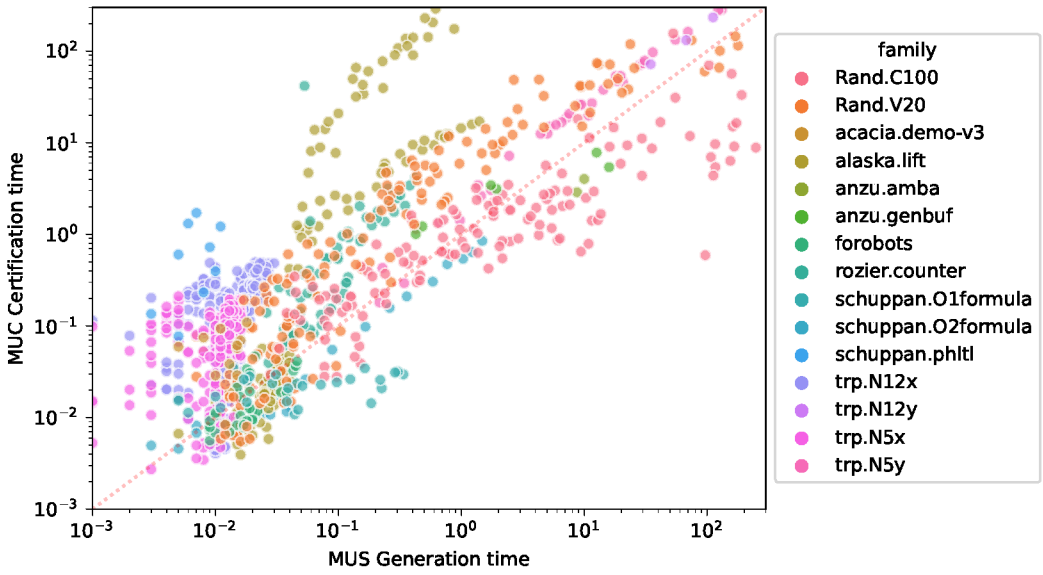
(a) C100 RndConj. (b) V20 RndConj. (c) TRP5y. (d) TRP12y.

Figure 6.3: Number of not-fully-enumerated instances (among the formula families RndConj C100, RndConj V20, TRP5y and TRP12y) that are able to enumerate the y percent of found MUCs within x seconds of runtime. Thus, 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.

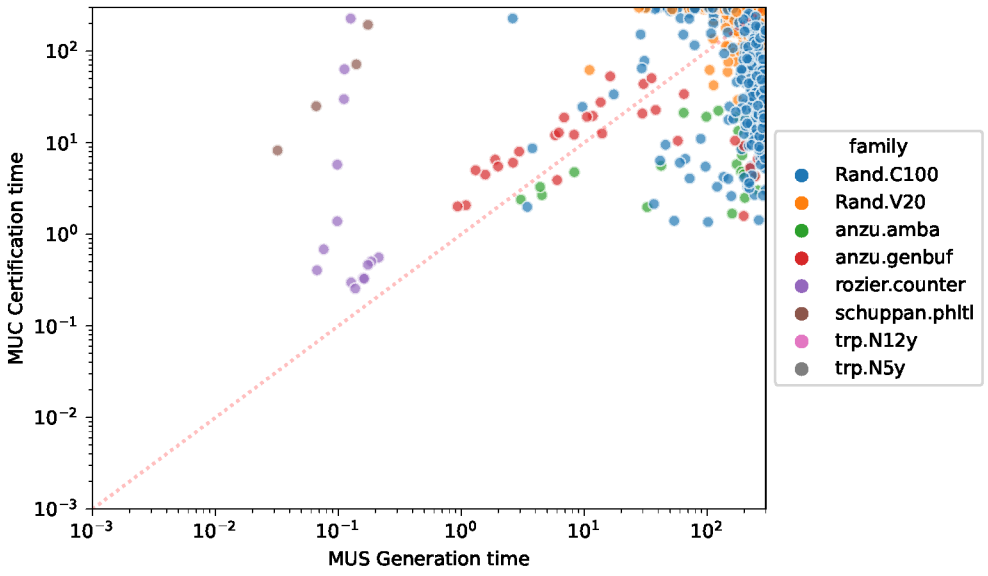
computes a single unsatisfiable core (with no minimality guarantees). Niu et al. [132] provides an in-depth comparison between `aaltaf-uc` and `aaltaf-muc` on this task. For the MUCs enumeration task we consider a general purpose tool `must` [15][6], which supports three LTL MUC enumeration algorithms (namely, `ReMUS`, `MARCO` and `TOME`). Note that, since `must` supports the LTL domain but not the LTL_f domain, we patch `must` by applying the well-known LTL-to-LTL_f transformation presented in [55] before evaluating LTL_f constraints within the MUC enumeration procedure. For further details, we refer the reader to [15].

Benchmarks. In our experiments we consider a benchmark suite consisting of common formulae families used in LTL and LTL_f literature to evaluate solvers. In particular, we use all unsatisfiable formulae that provided by Schuppan and Darmawan [146] and randomly generated formulae provided by Li et al. [107]. These formulae have been previously used to benchmark single MUC computation [132] and single UC (with no minimality guarantee) extraction [144]. This benchmark suite contains unsatisfiable instances from 15 different applications domains, each with different formula shapes and feature. In particular, they comprise both instances from applications (13 domains) and randombly generated (2 domains). In total, the suite [146] contains 2079 unsatisfiable instances. All instances were mapped in conjunctive form by recursively traversing the formula parse tree in a top-down fashion, stopping whenever formulae are not conjunctions. This is con-

[6]https://github.com/jar-ben/mustool; `17fa9f9`



(a) Fully-enumerated instances.



(b) Incomplete instances.

Figure 6.4: A point (x,y) in the scatter plot represents that a certain instance has spent x CPU seconds generating k-MUCs (e.g., ASP MUS enumeration) and y CPU seconds certifieing k-MUCs (e.g., an LTL_f solver running to prove unsatisfiability). Colors represent the formula family an instance belongs to. For a given point, $x+y$ might not sum up to the 300s timeout, because *(i)* these modules run concurrently, thus CPU time can exceed 300s; *(ii)* if the ASP solver or the LTL_f solver times out before yielding control, no event is recorded.

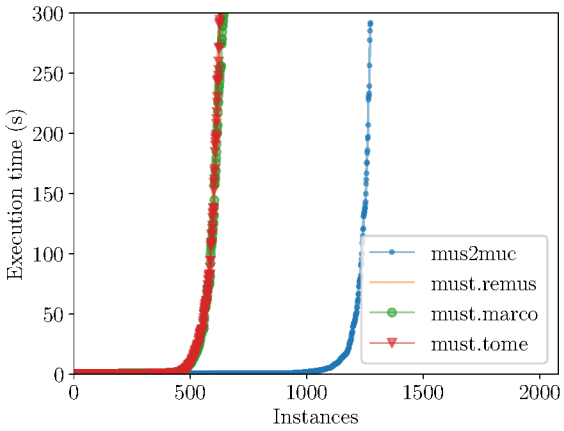


Figure 6.5: Number of fully-enumerated instances.

sistent with how these instances have been handled in literature [132, 144]. All formulae are interpreted as LTL_f formulae.

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 all experiments and systems, over each instance in the benchmark, memory and time were limited to 8GB and 300s of real time, 700s of CPU time respectively.

Extraction of a single MUC. First of all we assess the performance of our implementation in the computation of a single MUC. Niu et al. [132] provide two SAT-based approaches for single MUC extraction, called *NaiveMUC* and *BinaryMUC*, as well as two heuristic variants called *NaiveMUC+UC* and *BinaryMUC+UC* which augment the approach with techniques used in boolean unsatisfiable cores extraction. For an in-depth analysis of the techniques, we refer the reader to the original paper [132].

A related subtask is that of single unsatisfiable core extraction (UC), that is a set of unsatisfiable formulae with no subset-minimality guarantee. Algorithms for single UC extraction have been recently surveyed [144] and compared on the single UC/MUC extraction task [132].

In this experiment, we consider all algorithms for single MUC extraction featured in the paper by Niu et al. [132] (namely, `aaltaf-muc.binary`,

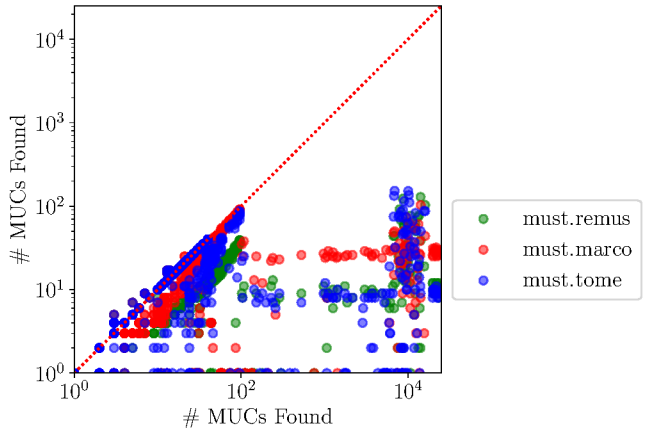


Figure 6.6: Number of MUCs enumerated within timeout for each instance.

`aaltaf-muc.naive`, `aaltaf-muc.binary-uc`, `aaltaf-muc.naive-uc` and the best-performing algorithm for single UC extraction featured in Roveri et al. [144] (namely, `aaltaf-uc`). We compare with our system `mus2muc` configured to stop at the first MUC extracted from each LTL_f formula.

The cactus plot in Figure 6.1a reports the performance of different systems in this task. Overall, we can observe that most of the formulae are trivial for all systems, resulting in sub-second runtimes. Figure 6.1b "zooms-in" to the hardest instances, where we 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 of Niu et al. [132]. Overall, `mus2muc` outperforms all systems in this task.

Enumeration of MUCs. Our second experiment consists in evaluating `mus2muc` effectiveness in *enumerating* MUCs of the formulae in the benchmark suite. Table 6.1 reports statistics about the number of found MUCs, probe depth and size of MUCs (i.e., number of conjuncts).

In general, different formula families exhibit heterogeneous behavior, ranging from *easy* (e.g., fully enumerated within seconds) to *hard* — yielding a number of MUCs in the order of thousands per instance that cannot be fully enumerated within the timeout. In particular, some of the *easy* families can be fully-enumerated with a probe depth that does not exceed one. Essentially, all inconsistencies can be detected at a propositional level, involving no temporal reasoning.

For the remaining formula families, we study *how fast* MUCs are extracted using `mus2muc`. The heatmaps in figures 6.2a–6.2d 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 formulae, our approach is able to output a considerable amount of MUCs in short time, albeit not able to fully enumerate them within the timeout. Conversely, the heatmaps in figures 6.3a–6.3d 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 (i.e., enumerated within timeout) within x seconds. We can see that for all these families, in the majority of instances a MUCs are computed in a steady fashion and MUCs become available within seconds of runtime. Instances in these families are characterized by a huge number of MUCs that cannot be realistically inspected. However, also in this scenario, our approach can provide a reasonable number of MUCs within few seconds.

Generation vs. Certification. In the `mus2muc` system, following Algorithm 2, each (unique) MUS extracted from the probe is checked for satisfiability by an LTL_f solver, to be either *certified* (that is, found unsatisfiable) or *disproved* (that is, there exists a satisficing trace whose length exceeds the current probe depth). In our implementation, MUS search and MUS certification run concurrently rather than in an interleaved fashion. Given the modularity of our approach, it is interesting to study which component affects runtimes the most. To this end, we consider formula families that are not fully enumerated within the timeout, but behave differently from the ones considered in the previous experiment.

In particular, when performing MUC extraction over an instance F, a certain amount of seconds due to MUS generation and MUS certification are accrued. Scatter plots in Figure 6.4 report each instance as a point (x, y), where x is the total CPU time spent running MUS generation procedures and y is the total CPU time spent running MUS certification[7]. Colors denote which family each datapoint belongs to.

We can observe in Figure 6.4b that MUS generation and MUC certification can both become bottlenecks in `mus2muc`, for unsatisfiable instances. Notably, some formula families such as *rozier.counter* feature unsatisfiable instances for

[7]Notice that, for a given instance, MUS generation runtimes and MUS certification runtimes do not necessarily sum up to the timeout since components run concurrently (i.e., CPU time could be greater than wall time). Furthermore, if a timeout signal is received *while* a MUC is being certified, `aaltaf` can't output any timestamp. Same goes for `wasp` during MUS generation. This explains not-fully-enumerated instances below the upper-right corner of the scatter plot.

which `wasp` is able to provide MUSes in less than a second, but whose certification time exceeds the allowed runtime. Similarly, in the $C100$ random conjunction family features instances for which the cumulative certification time is one order of magnitude smaller than MUS generation time. This sort of trade-off can be better analyzed by considering only fully enumerated instances in Figure 6.4a, where it is possible to observe heterogeneous behavior among families, ranging from families that are trivial from both standpoints (lower left corner); hard from both standpoints (upper right corner); easy MUS generation-wise, but hard MUC-certification wise (upper left corner). No fully-enumerated instances are easy certification-wise and hard generation-wise — as we have a mostly empty lower right corner in scatter plot.

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, a number of *general purpose*, *domain-agnostic* MUC extraction algorithms (which also support LTL as a domain) are available [15]. The survey by [144], does not compare with algorithms proposed in [15].

Figure 6.5 compares the number of fully-enumerated instances among different `must` algorithms and `mus2muc`. `mus2muc` is more effective, and can fully-enumerate approximately 500 more instances than any `must` variant. All `must` variants perform roughly the same. Figure 6.6 compares the number of found MUCs of each `must` variant with respect to `mus2muc`. A point (x, y) in Figure 6.6 denotes that, for a given instance in the benchmarks suite, `mus2muc` has computed x MUCs whereas one of the `must` algorithms has computed y MUCs. Each color distinguish a specific *must* algorithm.

We can see, from the cactus plot in Figure 6.5, that different algorithms of `must` are able to fully-enumerate (roughly) the same number of instances, indeed corresponding lines are mostly overlapped. Despite the different algorithms of `must` are able to fully-enumerate (roughly) the same number of instances (as we can see from the overlapping lines the cactus plot in Figure 6.5), we are able to observe some differences in the scatter plot in Figure 6.6. Overall, `mus2muc` is able to enumerate more MUCs than any of the `must` variants — in some extreme cases, enumerating several order of magnitude more MUCs (see the points that lie on the x-axis).

6.6 Discussion

Satisfiability of temporal specifications expressed in LTL_f play an important role in several artificial intelligence application domains [14, 28, 53, 54, 52]. Therefore, in case of unsatisfiable specifications, detecting reasons for unsatisfiability — e.g., 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 [132, 144] 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 [125], as it is the case for propositional logic [117, 119].

In this thesis, 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 [15].

The approach presented herein is modular with respect to ASP and LTL_f reasoners, which essentially constitute two sub-modules of the system, and with respect to 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 `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 available state-of-the-art for single MUC computation.

To the best of our knowledge, this represent the first attempt to address this task in the LTL_f setting.

As far as future works are concerned, we are interested in studying how the choice of probes affect 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.

Part III

Applications

Chapter 7

Direct ASP Encoding for Declare Constraints

In this chapter we propose ad-hoc ASP encoding to support Declare tasks, a template-based language which maps to a syntactical fragment of LTL_f. We also provide an experimental analysis to compare this novel encoding to other, indirect, ASP encodings that rely on translations to intermediate representations (resp. finite state machines and syntax trees).

7.1 Introduction

In the context of process mining, a *process* typically refers to a sequence of events or activities that are performed in order to accomplish a particular goal within a business or organizational setting [4]. Process Mining [1] is an interdisciplinary field offering techniques and tools to discover, monitor, and improve real processes by extracting knowledge from event logs readily available in today's information systems. One of the main tasks of Process Mining is *conformance checking*, assessing the correctness of a specific execution of a process, known as *trace*, against a *process model*. Such a process model is a formal mathematical representation enabling various analytical and reasoning tasks related to the underlying process. Process models adopt either an imperative or a declarative language form, with the former explicitly describing all possible process executions, and the latter using logic-based constraints to define what is not permitted within process executions. Imperative process models are suitable for well-structured routine processes but often fall short in scenarios involving complex coordina-

tion patterns. Declarative models, conversely, offer a flexible alternative in such scenarios.

Linear temporal logic over finite traces (LTL_f) [55] has emerged has a natural choice for declarative process modeling. However, within real-world Process Mining scenarios, LTL_f formulae that are unrestricted in structure are rarely used. Rather, a specific collection of predefined patterns, which have their origins in the domain of system verification [61], is commonly used. Limiting declarative modeling languages to a set of predefined patterns has two advantages: first, it simplifies modeling tasks [89], and second, it allows for the development of specialized solutions that may outperform generic LTL_f reasoners in terms of efficiency. Specifically, Declare [3] is the declarative process modeling language most commonly used in Process Mining applications and consists of a set of patterns, referred to as *templates*. The semantics of Declare templates is provided in terms of LTL_f, thus enabling logical reasoning and deduction processes within the Declare framework [57].

Recent research proposals suggest that Answer Set programming (ASP) [81] can be successfully used for various tasks within declarative process mining [92, 40]. Nevertheless, to the best of our knowledge, there has been no prior attempt to encode the Declare LTL_f patterns library using ASP in a *direct* way. Here, "direct" means an encoding that captures the semantics of Declare constraints without the need for any intermediary translation. This chapter fills this gap by introducing a direct encoding approach for the main Declare patterns. The proposed technique is benchmarked against existing ASP-based encodings across various logs commonly used in Process Mining [112]. The goal of the experimental analysis is twofold: first, to assess the performance of ASP-based methods in tasks related to conformance and query checking; and second, to evaluate the effectiveness of our proposed direct encoding strategy. To facilitate reproducibility, code and data used in our experiments are openly accessible at `https://github.com/ainnoot/padl-2024`.

7.2 Translation-based ASP encodings for Declare

This section introduces ASP encodings for conformance checking of Declare models and query checking of Declare constraints with respect to an input event log, based on the translation to automata and syntax trees. Both encodings share the same input fact schema to specify which Declare constraints belong to the model, or which constraint we are performing query checking against. These en-

codings are *indirect*, since they rely on a translation, but also *general* in the sense that they can be applied to the evaluation of arbitrary LTL_p formulae. This is achieved, in the case of the syntax tree encoding, by reifying the syntax tree of a formula and by explicitly modeling the semantics of each LTL_p temporal operator through a logic program, and in the case of the automaton encoding, by exploiting the well-known LTL_p-to-automaton translation [41, 55]. Thus, one can use these two encodings to represent Declare constraints by their LTL_p definitions. The automaton-based encoding is adapted from [40], the syntax tree-based encoding is adapted from [99] integrating changes to allow for the above-mentioned shared fact schema and evaluation over multiple traces. A similar encoding has also been used in [91] to learn LTL_f formulae from sets of example traces, using the ASP-based inductive logic programming system ILASP [102].

Earlier work in answer set planning [149] also exploited LTL constraints using a similar syntax tree-reification approach. We start by defining how event logs and Declare constraints are encoded into facts, then introduce conformance checking and query checking encodings with the two approaches.

7.2.1 Encoding process traces and Declare models

Encoding process traces For our purposes, an event log $\mathscr{L}$ is a multiset of process traces, thus a multiset of strings over an alphabet of propositional symbols $\mathscr{A}$ (representing activities). We assume that each trace $\pi \in \mathscr{L}$ is uniquely indexed by an integer, and we denote that the trace π has index i by $id(\pi) = i$. This is a common assumption in Process Mining, where i is referred to as the *trace identifier*. Traces are modeled through the predicate `trace/3`, where the atom $trace(i,t,a)$ encodes that $\pi_t = a, id(\pi) = i$ — that is, the t-th activity in the i-th trace π is a. Given a process trace π, we denote by $E(\pi)$ the set of facts that encodes it. Thus, an event log $\mathscr{L}$ is encoded as $E(\mathscr{L}) = \bigcup_{\pi \in \mathscr{L}} E(\pi)$.

Example 7.2.1 (Encoding a process trace)**.** *Consider an event log composed of the two process traces $\pi^0 = abc$ and $\pi^1 = xyz$, respectively with identifiers 0 and 1, over the propositional alphabet $\mathscr{A} = \{a,b,c,x,y,z\}$. This is encoded by the following set of facts:*

```
trace(0,0,a). trace(0,1,b). trace(0,2,c). trace(1,0,x). trace(1,1,y).
    trace(1,2,z).
```

In our encoding, activities are represented as constants that appear at least once in a trace in the event log, e.g. a, such that $trace(\_,\_,a)$ is a fact in the input event log.

Each Declare template, informally, can be understood as a "LTL_p formula with variables". Substituting these variables with activities yields a Declare constraint. How templates are instantiated into constraints, and how constraints are evaluated over traces, depends on the ASP encoding we use. However, all encodings share a common fact schema where constraints are expressed as templates with bound variable substitutions.

Encoding Declare constraints A Declare constraint is modeled by predicates `constraint/2` and `bind/3`. The former model which Declare template a given constraint is instantiated from and the latter which activity-variable bindings instantiate the constraint. An atom $\mathit{constraint}(\mathit{cid}, \mathit{template})$ encodes that the constraint uniquely identified by cid is an instance of the template $\mathit{template}$. The atom $\mathit{bind}(\mathit{cid}, \mathit{arg}, \mathit{value})$ encodes that the constraint uniquely identified by cid is obtained by binding the argument arg to the activity value. Given a Declare model $\mathscr{M} = \{c_1, \ldots, c_n\}$, where the subscript i uniquely indexes the constraint c_i, we denote by $E(\mathscr{M})$ the set of facts that encodes $\mathscr{M}$, that is $E(\mathscr{M}) = \bigcup_{c \in \mathscr{M}} E(c)$. Recall that in Declare $\pi \models \mathscr{M}$ if and only if $\pi \models c$ for all $c \in \mathscr{M}$, thus there is no notion of "order" among the constraints within $\mathscr{M}$ and it does not matter how indexes are assigned to constraints as long as they are unique.

Example 7.2.2 (Encoding a Declare model)**.** *Consider the model $\mathscr{M}$ composed of the two constraints $\mathit{Response}(a_1, a_2)$ and $\mathit{Precedence}(a_2, a_3)$. $\mathscr{M}$ is encoded by the following facts:*

```
constraint(0,"Response").
bind(0,arg_0,a_1).
    bind(0,arg_1,a_2).
```

```
constraint(1,"Precedence").
bind(1,arg_0,a_2).
    bind(1,arg_1,a_3).
```

7.2.2 Encoding Conformance Checking

All the Declare conformance checking encodings we propose consist of a stratified normal logic program [111, 9] P_{CF} such that, given a log $\mathscr{L}$ and a Declare model $\mathscr{M}$, we have that for all $\pi_i \in \mathscr{L}$, $\pi_i \models c_j \in \mathscr{M}$ if and only if the unique model of $P_{CF} \cup E(\mathscr{M}) \cup E(\mathscr{L})$ contains the atom $\mathit{sat}(i, j)$. The binary template Response will be our example to showcase the different encodings. Complete encodings for all the templates in Table 4.1 are available at `https://github.com/ainnoot/padl-2024`.

Automaton encoding The automaton encoding, reported in Figure 7.1, models Declare templates through their corresponding automaton obtained by translating the template's LTL_p definition [51]. The automaton's complete transition function is reified into a set of facts that defines the template in ASP. The predicates `initial/2, accepting/2` model the initial and accepting states of the automaton, while `template/4` stores the transition function of the template-specific automaton.

In particular, `arg_0` refers to the template *activation*, and `arg_1` refers to the template *target*. A constraint c instantiated from a template binds its `arg_0`, `arg_1` to specific activities. The constant `"*"` is used as a placeholder for any activity in $\mathscr{A} \setminus \{x, y\}$ – where x and y are the bindings of `arg_0` and `arg_1`. Activities not explicitly mentioned as within the atomic propositions in an LTL_p formula φ have the same influence to $\pi \models \varphi$. Consequently, all unbound activities can be denoted by the symbol `"*"` in the automaton transition table. As an example, consider the constraint

$$c = \mathsf{Response}(\mathsf{a},\mathsf{b}),$$

shown in Figure 7.3. Evaluating the trace `abwqw` is equivalent to evaluating the trace `abtts`, which would be equivalent to evaluating the trace `ab***`, since a and b are the only propositional formulae that appear in the definition of $\mathsf{Response}(\mathsf{a},\mathsf{b})$.

Within the encoding shown in Figure 7.1, the predicate $cur\_state/4$ stores the computation the automaton has performed to evaluate a trace, that is the "sequence of visited states". In particular, the atom $cur\_state(c, \pi, s, i)$ models that within the to evaluate constraint c over π, the corresponding automaton is in state s on the i-th state of π. Rules 4 and 5 encode the automaton transition function; in particular, Rule 4 encodes the transition function for the constraint's activation and target, while Rule 5 encodes the transition function for all the other characters. Rule 1 determines the automaton initial state, according to the constraint definition. Finally, Rule 2 and 3 define the notion of "accepted string" for the automaton—that is, the computation over π ending in an accepting state.

Syntax tree-based encoding. The syntax tree encoding, shown in Figure 7.2, reifies the syntax tree of a LTL_p formula into a set of facts, where each node represents a sub-formula. The semantics of temporal operators and propositional operators is defined in terms of ASP rules, as previously discussed in Chapter 5. Analogously to the automaton encoding, templates are defined in terms of reified syntax trees, which are used to evaluate each constraint according to the template

```
% (1) Automaton initial state
cur_state(C,TID,S,0) :-
  trace(TID,_,_),
  initial(Template,S),
  constraint(C,Template).

% (2) Last point of each trace
last(TID,T) :-
  trace(TID,T,_),
  not trace(TID,T+1,_).

% (3) A trace is accepted
sat(TID,C) :-
  cur_state(C,TID,S,T+1),
  last(TID,T),
  template(Template,C),
  constraint(C, Template),
  accepting(Template,S).
```

```
% (4) Reads activation/target
cur_state(C,TID,S2,T+1) :-
  cur_state(C,TID,S1,T),
  constraint(C,Template),
  template(Template,S1,Arg,S2),
  trace(TID,T,A),
  bind(C,Arg,A).



% (5) Reads "*"
cur_state(C,TID,S2,T+1) :-
  cur_state(C,TID,S1,T),
  constraint(C,Template),
  template(Template,S1,"*",S2),
  trace(TID,T,A),
  not bind(C,_,A).
```

Figure 7.1: ASP program to execute a finite state machine corresponding to a constraint, encoded as `template/4` facts, on input strings encoded by `trace/3` facts.

they are instantiated from. The following normal rules define the semantics of each temporal and propositional operator. With respect to Chapter 5, some minor, syntactical adaptations are required in order to being able to evaluate multiple traces and templates. We report the rules for operators $\{\mathsf{U}, \mathsf{X}, \neg, \wedge\}$ which are the basic operators of LTL_p. The full encoding, that also includes definitions of derived operators, is available online. In particular, the `true/4` predicate tracks which sub-formula of a constraints' definition is true at any given time. As an example, the atom $true(c, f, t, i)$ encodes that *at time t the constraint sub-formula f of constraint c is satisfied on the i-th trace*. The predicates `conjunction/3`, `negate/2`, `next/2`, `until/3` model the topology of the syntax tree of the corresponding formula. The first term refers to a node identifier, while the other terms (one for unary operators, two for the binary operators are the node identifiers of its child nodes. The `atom/2` predicate models that a given node (first term) is an atom, bound to a particular argument (second term) by the `bind/3` predicate which is used in encoding of Declare constraints. Figure 7.4 shows an example.

7.2.3 Encoding Query Checking

The query checking problem takes as input a Declare template $\mathscr{T}$, an event log $\mathscr{L}$ and consists in deciding which constraints c can be instantiated from $\mathscr{T}$ such that $\sigma(c, \mathscr{L}) \geq k$, where $\sigma(c, \mathscr{L})$ is the support and denotes the fraction of traces in $\mathscr{L}$ that are models of c. The problem has been formally defined by Chan [34] for temporal logic formulae, and it has also been re-proposed in a Process Mining setting [140] in the context of LTL_f. An ASP-based solution to the problem has been provided through the same automaton encoding we have been referring to throughout the chapter[40], and instead an exhaustive search-based, Declare-specific implementation is provided in the Declare4Py [60] library. From the ASP perspective, a conformance checking encoding can be easily adapted to perform query checking, by searching over possible variable-activities bindings that yield a constraint above the chosen support threshold. In order to encode the query checking problem, we slightly change our input model representation, as reported in the following example.

Example 7.2.3 (Query checking)**.** *Consider the query checking problem instance over the template Response, with both its activation and target ranging over $\mathscr{A}$. The `var_bind/3` predicate, analogously to `bind_3`, models that in a given template a parameter is bound to a variable. For the query checking problem, we are interested in tuples of activities that, when substituted to the constraints' vari-*

```
last(TID,T) :-
  trace(TID,T,_),
  not trace(TID,T+1,_).

true(C,F,T,TID) :-
  constraint(C,Template),
  template(Template,
    atom(F,Arg)
  ),
  bind(C,Arg,A),
  trace(TID,T,A).

true(C,F,T,TID) :-
  constraint(C,Template),
  template(Template,
    conjunction(F,G,H)
  ),
  trace(TID,T,_),
  true(C,G,T,TID),
  true(C,H,T,TID).

true(C,F,T,TID) :-
  constraint(C,Template),
  template(Template,negate(F,G)),
  not true(C,G,T,TID),
  trace(TID,T,_).
```

```
sat(C,TID) :-
  true(C,0,0,TID).

true(C,F,Ti,TID) :-
  constraint(C,Template),
  template(Template,next(F,G)),
  trace(TID,Ti,_),
  Tj=Ti+1,
  Ti<M,
  last(TID,M),
  true(C,G,Tj,TID).

true(C,F,Ti,TID) :-
  constraint(C,Template),
  template(Template,
      until(F,G,H)),
  trace(TID,Ti,_),
  trace(TID,Tj,_),
  Tj>=Ti, Tj<=M,
  last(TID,M),
  {true(C,G,T,TID):
      trace(TID,T,_), T>=Ti,
      T<Tj} = Tj-Ti,
  true(C,H,Tj,TID).
```

Figure 7.2: ASP program to evaluate each sub-formula of the LTL_p definition of a given template, encoded as `template/2` facts, on input strings encoded by a syntax tree representation through the `conjunction/3`, `negate/2`, `until/3`, `next/2` and `atom/2`.

```
template("Response",0,"*",0).
template("Response",0,arg_1,0).
template("Response",0,arg_0,1).
template("Response",1,arg_1,0).
template("Response",1,"*",1).
template("Response",1,arg_0,1).
accepting("Response",0).
initial("Response",0).
```

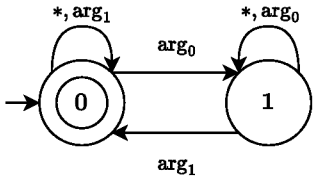


Figure 7.3: Left: Facts that encode the Response template; Right: A minimal finite state machine whose recognized language is equal to the set of models of Response, under LTL_p semantics.

```
template("Response",always(0,1)).
template("Response",implies(1,2,3)).
template("Response",atom(2,arg_0)).
template("Response",eventually(3,4)).
template("Response",atom(4,arg_1)).
```

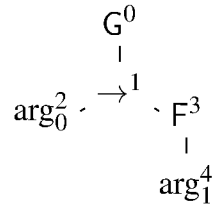


Figure 7.4: Left: Facts that encode the Response template; Right: Syntax tree of the Response template LTL_p definition.

ables, yield a constraint whose support is above the threshold over the input log. The `domain/2` *predicate can be used to give each variable its own subset of possible values, but in this case, for both variables, the domain of admissible substitutions spans over* $\mathscr{A}$*. The choice rule generates candidate substitutions that are pruned by the constraints if they are above the maximum number of violations. Given an input support threshold* $s \in (0,1]$*, the constant* `max_violations` *is set to the nearest integer above* $(1-s)\cdot|\mathscr{L}|$*.*

```
constraint(c,"Response"). var_bind(c,arg_0,var(a)).
    var_bind(c,arg_1,var(b)).
domain(var(a),A) :- trace(_,_,A).
domain(var(b),A) :- trace(_,_,A).
{ bind(C,Arg,Value): domain(Var,Value) } = 1 :- var_bind(C,Arg,Var).
:- #count{X: not sat(C, X), constraint(C,_), trace(X, _, _)} >
    max_violations.
```

Notice the ASP formulation can be easily generalized (by simply adding facts encoding a Declare constraint and slightly modifying the final constraint) to query check entire Declare models (i.e., multiple constraints, where a variable might occur as activation/target of distinct constraints), rather than a single constraint at a time, by adding the following facts:

```
constraint(d,"Precedence"). var_bind(d,arg_0,var(a)).
    var_bind(d,arg_1,var(c)).
domain(var(d),A) :- trace(_,_,A).
```

Here, the variable a is the activation of the Response constraint (as before), as well as the target of the Precedence constraint.

7.3 Direct ASP Encoding for Declare

The encodings described in previous section are general techniques that enable reasoning over arbitrary LTL_p formulae. Both encodings require, respectively, to keep track of each subformula evaluation on each time-point of a trace in the case of the syntax-tree encoding, and to keep track of DFA state during the trace traversal, in the case of the automaton encoding. However, Declare patterns do not involve *complex* temporal reasoning, with deep nesting of temporal operators. Hence Declare templates admit a succinct and direct encoding in ASP that does not keep track of such evaluation at each time-point of the trace. The encoding discussed in this section exploits this, providing an ad-hoc, direct translation of the semantics of Declare constraints into ASP rules.

The general approach we follow in defining the templates, is to model cases in which constraints fail through a `fail/2` predicate. In our encoding, a constraint c over a trace tid holds true if we are unable to produce the atom $fail(c, tid)$. Due to the activation-target semantics of Declare templates, sometimes it is required to assert that an activation condition is matched in the suffix of the trace by a target condition. In the encoding, this is modeled by the `witness/3` predicate. This mirrors the *activation* and *target* concepts in the definition of Declare constraints. Note that, this approach is not based on a systematic, algorithmic rewriting, but on a template-by-template ad-hoc analysis. In the rest of the section, we show the principles behind our ASP encoding for the Declare templates in Table 4.1.

Response template Recall from Table 4.1 that Response(a,b) is defined as the LTL_p formula $\mathsf{G}(a \rightarrow \mathsf{F}b)$, whose informal meaning is that *whenever a happens, b must happen somewhere in the future*. Thus, every time we observe an a at time t, in order for $Response(a,b)$ to be true, we have to observe b at a time instant $t' \geq t$. The first rule below encodes this situation. If we observe at least one a that is not matched by any b in the future, the constraint fails, as encoded in the second rule. The following equivalences exploit the duality of temporal operators G, and F:

$$\neg\mathsf{G}(a \rightarrow \mathsf{F}b) \equiv \mathsf{F}\neg(a \rightarrow \mathsf{F}b) \equiv \mathsf{F}\neg(\neg a \vee \mathsf{F}b) \equiv \mathsf{F}(a \wedge \neg\mathsf{F}b)$$

Thus, we encode *Response* failure conditions with the following logic program:

```
witness(C,T,TID) :-
  constraint(C, "Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID, T, X),
  trace(TID, T', Y), T' > T.
```

```
fail(C,TID) :-
  constraint(C, "Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID, T, X),
  not witness(C,T,TID).
```

The rule above on the left yields a $witness(c,t,tid)$ atom whenever trace tid at time t satisfies $a \wedge \mathsf{F}b$. In the rule on the right, the $fail(c,tid)$ atom models that there exists some time-point t such that $\pi_t = a$, but $\neg(a \wedge \mathsf{F}b)$ thus (since $a \in \pi_t$) that $\neg\mathsf{F}b$; that is, there exists a time-point t such that $a \wedge \neg\mathsf{F}b$.

Figure 7.5 exemplifies graphically the failure conditions for the Response template. Color hints match the affected literals in the Response template ASP encoding.

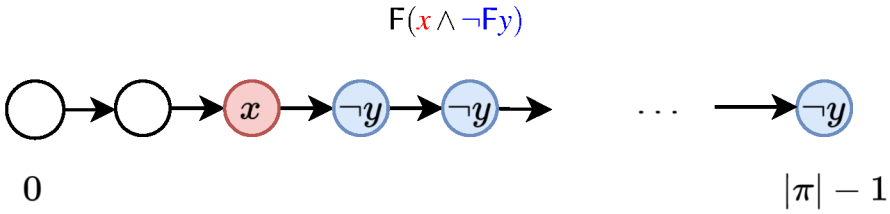


Figure 7.5: Graphical representation of the Response template failure condition.

Precedence template Recall from Table 4.1 that the constraint Precedence(x,y) is defined as the LTL_p formula $\neg y\mathsf{W}x = \mathsf{G}(\neg y) \vee \neg y\mathsf{U}x$, whose informal meaning is that *if y occurs in the trace, x must have happened before*. Notice that in order to witness the failure of this constraint, it is enough to reason about the trace prefix up to the first occurrence of y, since in the definition of the template the until operator is not under the scope of a temporal operator. The following equivalences hold:

$$\neg(\neg y\mathsf{W}x) \equiv \neg(\mathsf{G}(\neg y) \vee \neg y\mathsf{U}x) \equiv \neg\mathsf{G}(\neg y) \wedge \neg(\neg y\mathsf{U}x) \equiv \mathsf{F}y \wedge \neg(\neg y\mathsf{U}x) \equiv \mathsf{F}y \wedge y\mathsf{R}\neg x$$

We model this failure condition with the following logic rules:

```
fail(C,TID) :-
  constraint(C, "Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T',Y),
  T = #min{Q: trace(TID,Q,X)},
  trace(TID,T,X),
  T' < T.
```

```
fail(C,TID) :-
  constraint(C, "Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,_,Y),
  not trace(TID,_,X).
```

The rule on the left models the formula $(y\mathsf{R}\neg x)$, that is satisfied by traces where x does not occur up to the point where y first becomes true. The rule on the right models traces where x does not occur at all, but y does. We model this separately to be compliant with `clingo`'s behavior where `#min` would yield the special term `#sup` over an empty set of literals.

Figure 7.6 exemplifies graphically the failure conditions for the Precedence template. Color hints match the affected literals in the Precedence template ASP encoding.

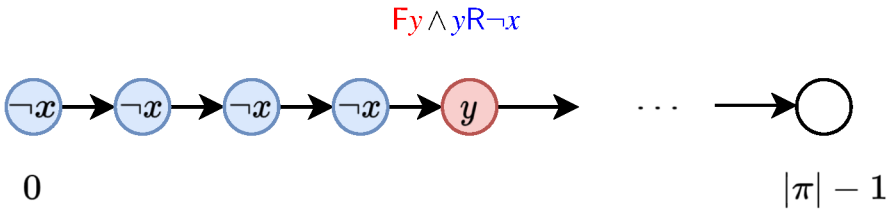


Figure 7.6: Graphical representation of the Precedence template failure condition.

Next, we consider the *AlternateResponse* and *AlternatePrecedence* templates. Differently from the previous cases, mapping out failure from the LTL_p formula is less intuitive, due to temporal operator nesting in the template definitions, and requires more care.

AlternateResponse template Recall from Table 4.1 that $\mathit{AlternateResponse}(a,b)$ is defined as the LTL_p formula $\mathsf{G}(a \rightarrow \mathsf{X}(\neg a \mathsf{U} b))$, whose informal meaning is that *whenever a happens, b must happen somewhere in the future and, up to that point, a must not happen.* Failure conditions follow from the this chain of equivalences:

$$\neg\mathsf{G}(a \rightarrow \mathsf{X}(\neg a \mathsf{U} b) \equiv \mathsf{F}(a \wedge \neg\mathsf{X}(\neg a \mathsf{U} b)) \equiv \mathsf{F}(a \wedge \mathsf{X}_\mathsf{w} \neg(\neg a \mathsf{U} b)) \equiv \mathsf{F}(a \wedge \mathsf{X}_\mathsf{w}(a \mathsf{R} \neg b))$$

Thus, the failure condition is to observe an a at time t, such that $\mathsf{X}_\mathsf{w}(a \mathsf{R} \neg b)$ holds, that is an occurrence of a at time $t' > t$ with $b \notin \pi_k$ for all $t < k < t'$. To model the constraint, we ensure at least one b appears in-between the a occurrences. We model this by the following rules:

```
witness(C,T,TID) :-
  constraint(C, "Alternate
      Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  T'' = #min{Q: trace(TID,Q,X),
      Q > T},
  trace(TID,T',Y), T'' > T', T'
      > T.
```

```
fail(C,TID) :-
  constraint(C, "Alternate
      Response"),
  bind(C, arg_0, X),
  trace(TID,T,X),
  not witness(C,T,TID).
```

A witness here is observing an a at time t such that the next occurrence of a at

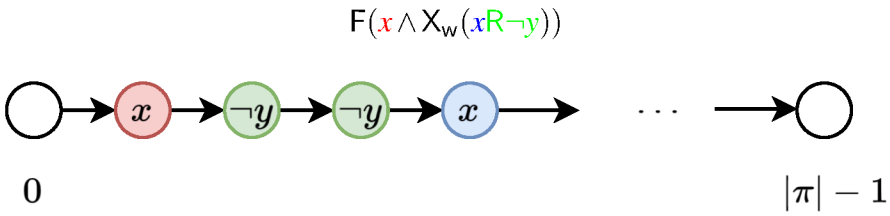


Figure 7.7: Graphical representation of the Alternate Response template failure condition.

time t', with at least one b in-between. To get this succinct encoding, we rely on `clingo` `#min` behavior: `#min` of an empty term set is the constant `#sup`, so the arithmetic literal `T'' > T'` will always be true when there's no occurrence of a following the one at time T - this deals with the "last" occurence of a in the trace. If two "adjacent" occurrences of a do not have a b in-between, that does not count as a witness (due to the constraint's semantics).

Figure 7.7 exemplifies graphically the failure conditions for the Alternate Response template. Color hints match the affected literals in the Alternate Response template ASP encoding.

AlternatePrecedence template Recall from Table 4.1 that the constraint

$$\mathit{AlternatePrecedence}(x,y)$$

is defined as the LTL_p formula

$$(\neg y \mathsf{W} x) \wedge \mathsf{G}(b \rightarrow \mathsf{X_w}(\neg y \mathsf{W} x)) = \mathit{Precedence}(a,b) \wedge \mathsf{G}(b \rightarrow \mathsf{X_w}\mathit{Precdence}(a,b))$$

whose informal meaning is that *every time y occurs in the trace, it has been preceded by x and no y happens in-between.* To map its failure conditions, we build this chain of equivalences, where $\alpha = \mathit{Precedence}(a,b)$:

$$\neg(\alpha \wedge \mathsf{G}(y \rightarrow \mathsf{X_w}\alpha)) \equiv \neg\alpha \vee \mathsf{F}(y \wedge \neg\mathsf{X_w}\alpha)$$

Above we described the encoding of $\mathit{Precedence}(x,y)$, thus, now we focus on the second term:

$$\mathsf{F}(y \wedge \neg\mathsf{X_w}\alpha) \equiv \mathsf{F}(y \wedge \neg \mathit{last} \wedge \neg\mathsf{X}\alpha) \equiv \mathsf{F}(y \wedge \neg \mathit{last} \wedge \mathsf{X_w}\neg\alpha) \equiv \mathsf{F}(y \wedge \mathsf{X}\neg\alpha)$$

where *last* is a propositional symbol that is assumed to be true only in the last $(|\pi| - 1)$ state of the trace, that is $last \equiv \mathsf{X_w}\bot$.

And, by substituting $\neg\alpha$ with the failure condition of $Precedence(x, y)$:

$$\mathsf{F}(y \wedge \mathsf{X}\neg\alpha) \equiv \mathsf{F}(y \wedge \mathsf{X}(\mathsf{F}y \wedge y\mathsf{R}\neg x))$$

Thus, $AlternatePrecedence(x, y)$ admits the same failure conditions as $Precedence(a, b)$, that are reported in the rules on the left below. Furthermore, from the second term we derive the failure condition which regards the occurrence of a b that is followed by a trace suffix where $Precedence(a, b)$ does not hold; that is, the b is followed by another b with no a in between. This is modeled by the rule on the right:

```
fail(C,TID) :-
  constraint(C, "Alternate
      Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T',Y),
  T = #min{Q: trace(TID,Q,X)},
  trace(TID,T,X),
  T' < T.

fail(C,TID) :-
  constraint(C, "Alternate
      Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,_,Y),
  not trace(TID,_,X).
```

```
fail(C,TID) :-
  constraint(C, "Alternate
      Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID, T0, Y),
  trace(TID, T2, Y),
  T2 > T0,
  #count{Q: trace(TID,Q,X), Q
      >= T0, Q <= T2} = 0.
```

Figure 7.8 exemplifies graphically the failure conditions for the Alternate Precedence template. Color hints match the affected literals in the Alternate Precedence template ASP encoding.

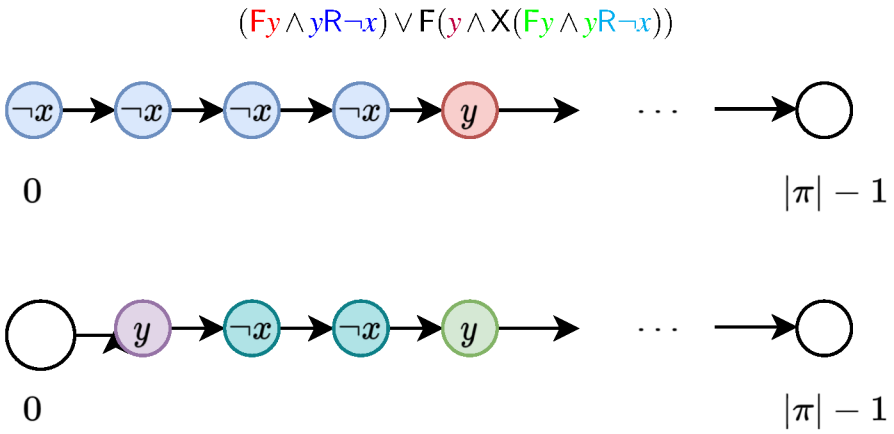


Figure 7.8: Graphical representation of the Alternate Precedence template failure condition.

ChainResponse template Recall that from Table 4.1 that the constraint *ChainResponse*(x, y) is defined as the LTL_p formula $\mathsf{G}(x \rightarrow \mathsf{X}y)$, whose informal meaning is that *whenever x occurs, it must be immediately followed by y*. The failure conditions follow from the following chain of equivalences:

$$\neg(\mathsf{G}(x \rightarrow \mathsf{X}y)) \equiv \mathsf{F}\neg(x \rightarrow \mathsf{X}y) \equiv \mathsf{F}\neg(\neg x \vee \mathsf{X}y) \equiv \mathsf{F}(x \wedge \neg\mathsf{X}y) \equiv \mathsf{F}(x \wedge \mathsf{X}_\mathsf{w}\neg y)$$

That is, an x occurs that is not followed by a y - either because x occurs as the last activity of the trace or that x is followed by a different activity. We can model this by the following rule:

```
fail(C,TID) :-
  constraint(C, "Chain Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  not trace(TID,T+1,Y).
```

Figure 7.9 exemplifies graphically the failure conditions for the Chain Response template. Color hints match the affected literals in the Chain Response template ASP encoding.

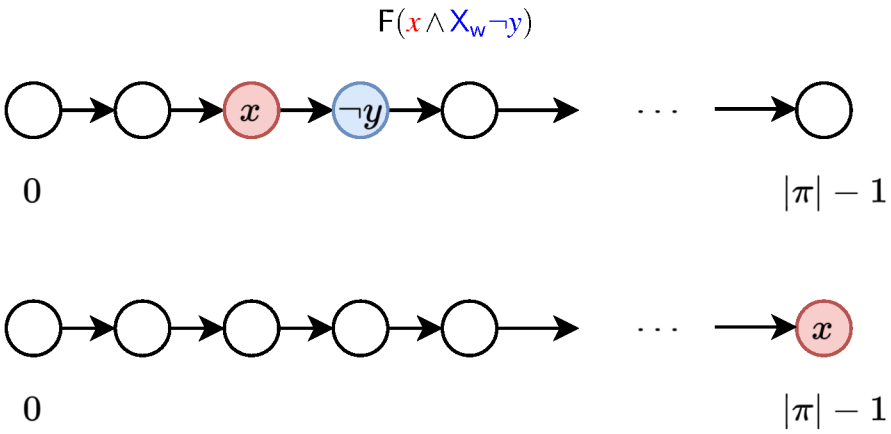


Figure 7.9: Graphical representation of the Chain Response template failure condition.

ChainPrecedence template Recall that from Table 4.1 that the constraint

$$\mathit{ChainPrecedence}(x, y)$$

is defined as the LTL_p formula $\mathsf{G}(\mathsf{X}y \rightarrow x) \wedge \neg b$, whose informal meaning is that *whenver y occurs, it must have been immediately preceded by x*. The failure conditions follow from the following chain of equivalences:

$$\neg(\mathsf{G}(\mathsf{X}y \rightarrow x) \wedge \neg y) \equiv (\mathsf{F}\neg(\mathsf{X}y \rightarrow x) \vee y \equiv \mathsf{F}(\mathsf{X}y \wedge \neg x) \vee y$$

In this case, *ChainPrecedence* fails if the predecessor of y is not x, or if y occurs in the first instant of the trace. This is modeled by the following rules:

```
fail(C, TID) :-
  constraint(C, "Chain Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T+1,Y),
  trace(TID,T,_),
  not trace(TID,T,X).

fail(C, TID) :-
  constraint(C, "Chain Precedence"),
  bind(C, arg_1, Y),
  trace(TID, 0, Y).
```

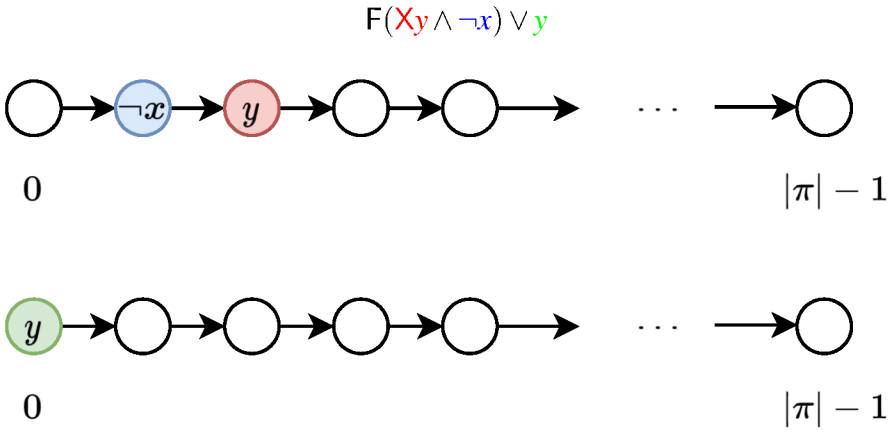


Figure 7.10: Graphical representation of the Chain Precedence template failure condition.

Figure 7.10 exemplifies graphically the failure conditions for the Chain Precedence template. Color hints match the affected literals in the Chain Precedence template ASP encoding.

Modeling the Succession hierarchy Templates in the *Succession* chain are defined as the conjunction of the respective *Precedence*, *Response* templates at the same "level" of the subsumption hierarchy (see Figure 4.1); thus, it is possible to encode them in the `fail/2`, `witness/3` schema, since $\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$. Hence, the failure conditions for *Succession*-based templates are the union of the failure conditions of its sub-formulae (all the cases were thoroughly described above in this section).

Modeling non-temporal templates Choice (i.e., *Choice*, *ExclusiveChoice*) and Existential templates (i.e., *RespondedExistence*, *Coexistence*) are defined only in terms of conjunction and disjunction of atomic formulae (e.g., activity occurrences in the trace). Thus, they are easy to implement using normal rules whose body contains `trace/3` literals to model that $a \in \pi_i, a \notin \pi_i$. As an example, consider the constraint $\mathit{RespondedExistence}(a,b)$ that states that *whenever a occurs in the trace, b must occur as well.* Its LTL_p definition is $\mathsf{F}(a) \rightarrow \mathsf{F}(b)$, its failure conditions follow from the following chain of equivalences:

Table 7.1: Log Statistics: $|\mathscr{A}|$ is the number of activities; Average $|\pi|$ is the average trace length; $|\mathscr{L}|$ is the number of traces; $|\mathscr{C}^{\text{IV}}|$ is the number of Declare constraints above 50% support.

Log name	$\lvert\mathscr{A}\rvert$	Average $\lvert\pi\rvert$	Max $\lvert\pi\rvert$	Min $\lvert\pi\rvert$	$\lvert\mathscr{L}\rvert$	$\lvert\mathscr{C}^{\text{IV}}\rvert$
Sepsis Cases (SC)	16	14.5	185	3	1050	76
Permit Log (PL)	51	12.3	90	3	7065	26
BPI Challenge 2012 (BC)	23	12.6	96	3	13087	10
Prepaid Travel Cost (PC)	29	8.7	21	1	2099	52
Request For Payment (RP)	19	5.4	20	1	6886	52
International Declarations (ID)	34	11.2	27	3	6449	152
Domestic Declarations (DD)	17	5.4	24	1	10500	52

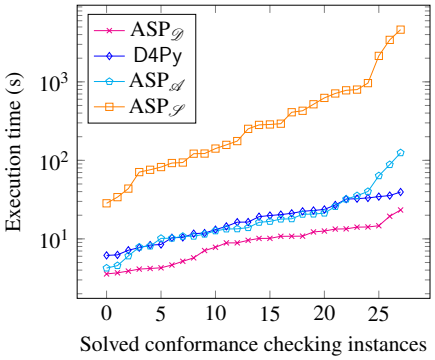


Figure 7.11: Conformance checking cactus.

Log	ASP$_{\mathscr{D}}$	D4Py	ASP$_{\mathscr{A}}$	ASP$_{\mathscr{S}}$
ID	**23.3**	39.5	124.9	4621.7
RP	**10.8**	16.3	25.8	409.8
PT	**5.2**	8.5	12.6	121.6
SC	**4.3**	11.6	13.4	141.2
PL	**10.8**	35.6	20.7	624.1
DD	**14.2**	22.4	40.1	963.2
BC	**14.1**	23.7	20.5	796.6

Table 7.2: Runtime in seconds for conformance checking on $\mathscr{C}^{\text{IV}}$.

$$\neg(\mathsf{F}(a) \rightarrow \mathsf{F}(b)) \equiv \neg(\neg\mathsf{F}(a) \vee \mathsf{F}(b) \equiv \mathsf{F}(a) \wedge \neg\mathsf{F}(b)$$

```
fail(C,TID) :- constraint(C, "Responded Existence"),
  bind(C, arg_0, X), bind(C, arg_1, Y), trace(TID,_,X), not
     trace(TID,_,Y).
```

All the encodings for the Declare constraints in Table 4.1 are available in the our git repository.[1]

[1] https://github.com/ainnoot/padl-2024.

7.4 Experiments

In this section, we report the results of our experiments comparing different methods to perform conformance checking and query checking of Declare models, using the ASP-based representations outlined in the previous sections and Declare4Py, a recent Python library for Declare-based process mining tasks which also implements - among other tasks, such as *log generation* and *process discovery* - conformance checking and query checking functionalities. While the approaches discussed here are declarative in nature, Declare4Py implements Declare semantics by imperative procedures, based on the algorithms proposed by Burattin, Maggi, and Sperduti [23], that scan the input traces. Declare4Py supports also other tasks, and most importantly, supports a data-aware variant of Declare, while here we deal with standard, control-flow only Declare. Methods will be referred to as $\text{ASP}_{\mathscr{D}}$, $\text{ASP}_{\mathscr{A}}$, $\text{ASP}_{\mathscr{S}}$ and D4Py - denoting respectively our direct encoding, the automata and syntax tree-based translation methods and Declare4Py. We start by describing datasets (logs and Declare models), and execution environment to conclude by discussing experimental results. Subsection 7.4.2 encompass further experimental analysis about memory consumption and behavior on longer traces for our ASP encoding.

Data. We validate our approach on real-life event logs from past BPI Challenges [112]. These event logs are well-known and actively used in Process Mining literature. For each event log $\mathscr{L}_i$, we use D4Py to mine the set of Declare constraints $\mathscr{C}_i$ whose support on $\mathscr{L}_i$ is above 50%. Then, we define four models, $\mathscr{C}_i^{\text{I}}, \mathscr{C}_i^{\text{II}}, \mathscr{C}_i^{\text{III}}, \mathscr{C}_i^{\text{IV}}$, containing respectively the first 25%, 50%, 75% and 100% of the constraints in a random shuffling of $\mathscr{C}_i$, such that $\mathscr{C}_i^{\text{I}} \subset \mathscr{C}_i^{\text{II}} \subset \mathscr{C}_i^{\text{III}} \subset \mathscr{C}_i^{\text{IV}}$. Table 7.1 summarizes some statistics about the logs and the Declare models we mined over the logs. All resource measurements take into account the fact that ASP encodings require an additional translation step from the XML-based format of event logs to a set of facts. The translation time is included in the measurement times and is comparable with the time taken by D4Py.

Execution Environment. The experiments in this section were executed on an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 512GB RAM machine, using CLINGO version 5.4.0, Python 3.10, D4Py 1.0 and `pyrunlim`[2] to measure resources usage. Experiments were run sequentially. All data and scripts

[2]https://github.com/alviano/python/tree/master/pyrunlim

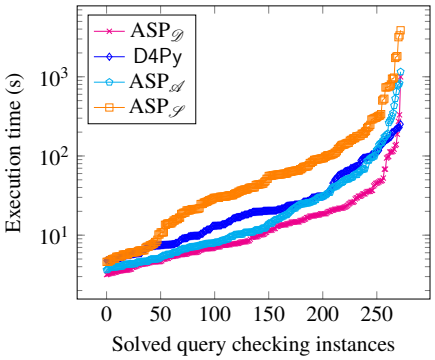


Figure 7.12: Query checking cactus plot.

Log	ASP$_{\mathscr{D}}$	D4Py	ASP$_{\mathscr{A}}$	ASP$_{\mathscr{S}}$
ID	**817.2**	1624.5	1654.0	3522.4
RP	884.2	565.8	**318.2**	1179.4
PT	**223.6**	451.1	236.1	427.9
SC	**163.8**	267.0	173.1	665.1
PL	**1614.0**	4227.7	3926.8	5397.5
DD	**407.7**	698.2	479.2	2436.2
BC	**2304.8**	2467.7	6636.0	27445.3

Table 7.3: Cumulative runtime in seconds for query checking tasks.

to reproduce our experiments are available at `https://github.com/ainnoot/padl-2024`.

7.4.1 Conformance checking & Query checking on real-world logs

Conformance Checking. We consider the conformance checking tasks $(\mathscr{L}_i, \mathscr{M})$, with $\mathscr{M} \in \{\mathscr{C}_i^{\mathrm{I}}, \mathscr{C}_i^{\mathrm{II}}, \mathscr{C}_i^{\mathrm{III}}, \mathscr{C}_i^{\mathrm{IV}}\}$, over the considered logs and its Declare models. Figure 7.11 reports the solving times for each method in a cactus plot. Recall that a point (x, y) in a cactus plot represents the fact that a given method solves the x-th instance, ordered by increasing execution times, in y seconds. Table 7.2 reports the same data aggregated by the event log dimension, best run-time in bold. Overall, our direct encoding approach is faster than the other ASP-based encodings as well as D4Py on considered tasks. ASP$_{\mathscr{A}}$ and D4Py perform similarly, whereas ASP$_{\mathscr{S}}$ is less efficient. It must also be noticed that D4Py, beyond computing whether a trace is compliant or not with a given constraint, also stores additional information such as the number of times a constraint is violated, or activate, while the ASP encodings do not. However, the automata and direct encoding can be straightforwardly extended in such sense; in particular, similar "book-keeping" in the direct encoding is performed by the `fail/3` and `witness/3` atoms.

Query Checking. We consider the query checking instances $(t, \mathscr{L}_i, s)$ where t is a Declare template, from the ones defined in Table 4.1, $s \in \{0.50, 0.75, 1.00\}$ is a support threshold, and $\mathscr{L}_i$ is a log. Figure 7.12 summarizes the results in a cactus plot, and Table 7.3 aggregates the same data on the log dimension, best runtime in bold. ASP$_{\mathscr{D}}$ is again the best method overall, outperforming other ASP-based methods with the exception of ASP$_{\mathscr{A}}$ on the RP log tasks. Again, ASP$_{\mathscr{A}}$ and D4Py perform similarly and ASP$_{\mathscr{S}}$ is the worst.

Discussion. In the comparative evaluation of the different methods for conformance and query checking tasks, our direct encoding approach ASP$_{\mathscr{D}}$ showed better performance compared to both D4Py and other ASP-based methods, as reported in Table 7.2 and Table 7.3 (and in Fig. 7.11 and Fig. 7.12). As shown in our experiments, ASP$_{\mathscr{D}}$ not only outperformed in terms of runtime the other methods but also offers a valuable alternative when considering overall efficiency. Notably, ASP$_{\mathscr{A}}$ and D4Py showed similar performances, with ASP$_{\mathscr{A}}$ slightly outperforming in certain instances, particularly in the RP log, but ASP$_{\mathscr{S}}$ exhibited less efficiency in nearly all the logs.

From the point of view of memory consumption (Table 7.4), D4Py proved to be the most efficient in query checking tasks. This can be attributed to its imperative implementation that allows for an *"iterate and discard"* approach to candidate assignments, avoiding the need for their explicit grounding required by ASP-based techniques. ASP$_{\mathscr{S}}$ is the least efficient method regarding memory usage, consistently across both types of tasks and all logs. We conjecture the significant increase in maximum memory usage is the primary factor contributing to the runtime performance degradation in both tasks. In fact, ASP$_{\mathscr{S}}$ proved to be the less efficient in both memory consumption and running time. Furthermore, we observe that in conformance checking ASP$_{\mathscr{D}}$ is more efficient w.r.t. memory consumption when compared to D4Py, and, when combined, these two methods collectively show better memory efficiency when compared to other ASP-based methods, translating in lower running times.

The obtained results clearly indicates the lower memory requirements of D4Py, demonstrating its applicability in resource-constrained environments. However, the compact and declarative nature of ASP provides an efficient means to implement Declare constraints, as demonstrated by the performance of ASP$_{\mathscr{D}}$, which is especially suitable in environments where memory is less of a constraint and execution speed is a key factor. Furthermore, the ASP-based approaches can be more readily extended, in a pure declarative way, to perform , for example,

Log	Conformance Checking				Query Checking			
	$\text{ASP}_{\mathscr{D}}$	D4Py	$\text{ASP}_{\mathscr{A}}$	$\text{ASP}_{\mathscr{S}}$	$\text{ASP}_{\mathscr{D}}$	D4Py	$\text{ASP}_{\mathscr{A}}$	$\text{ASP}_{\mathscr{S}}$
BC	**323.6**	566.3	546.0	8757.9	3157.0	**580.7**	1450.8	18866.5
DD	536.6	**386.0**	927.2	14473.8	728.1	**336.4**	513.0	2706.0
ID	837.1	**578.4**	2338.9	55435.2	1498.5	**583.5**	763.6	5029.8
PL	**312.8**	2062.2	579.5	11388.3	2434.2	2071.0	**1023.3**	7006.2
PC	**222.2**	281.9	341.2	5211.2	435.3	283.4	**282.8**	1209.3
RP	372.3	**347.6**	610.8	9706.0	574.8	**312.3**	396.9	1843.3
SC	**195.0**	245.6	336.4	5786.5	480.4	**244.6**	247.2	2146.7

Table 7.4: Max memory usage (MB) over all the conformance checking and query checking tasks, aggregated by log, for all the considered methods. Lowest value in boldface.

query-checking of multiple Declare constraints in a matter of few extra rules.

An intuitive reason for such a memory usage gap in the conformance checking tasks is the following. As noted in the encodings section, both the automata encoding and the syntax tree encoding rely on a 4-ary predicate ($true(TID,C,X,T)$, and $cur\_state(TID,C,X,T)$ respectively), that keeps track of subformulae evaluation on each instant of the trace and current state in the constraint DFA for each instant of the trace. In the case of the automaton encoding, X does not index subtrees but states of the automaton - all the other terms have the same meaning. During the grounding of the logic programs, this yields a number of symbolic atoms that scales linearly w.r.t. the total number of events in the log (that is, the sum of lengths of the traces in the log) and linearly in the number of nodes in the syntax tree/states in the automaton. The gap between the formula encoding and the automata encoding is explained by the fact that the LTL_f to symbolic automaton translation, albeit 2-EXP in the worst case, results usually in *more compact* automatons in the case of the Declare patterns. For example, considering the Response template: its formula definition is $\mathsf{G}(a \rightarrow \mathsf{X}b)$, with a size of 5 (that is, the number of nodes in its syntax tree), while its automaton as 2 states. Furthermore, since Declare assumes simplicity (e.g., has $|\pi_i| = 1$), the symbolic automaton can be further simplified for some constraints. This yields, overall, *less states w.r.t the number of nodes in the parse tree* - that reduces the number of ground symbols. Total number of events is the same for both encodings, since both encodings share the same input fact schema. On the other hand, as we sketched in Section 3, the direct encoding explicitly models *how constraints are violated*, thus for most templates, it produces less atoms, since we yield at most one `witness/3` atom

Metric	Method		
	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{S}}$	$\text{ASP}_{\mathscr{A}}$
Symbols	117176	4593770	1329322
Rules	143106	5953435	1345738
Execution time (s)	0.59	123.55	10.75
Max Memory (MB)	60.94	5756.16	300.81

Table 7.5: Metrics comparison for conformance checking of $\mathscr{C}^{\text{IV}}$ over the Sepsis Cases Event Log, with ASP encodings $\text{ASP}_{\mathscr{D}}$, $\text{ASP}_{\mathscr{S}}$, and $\text{ASP}_{\mathscr{A}}$. Execution times do not take into account XES input parsing and output parsing (as in Table 7.2 , but are performed directly on facts representation of the input. Thus, reported times slightly differ from Table 7.2, although being executed on the same log.

for each occurrence of the constraint activation, and at most one `fail/2` atom for each constraint and each trace. Table 7.5 reports the number of generated symbols on the three different encodings to conformance check $\mathscr{C}^{\text{IV}}$ on the Sepsis Cases event log, where (see Table 7.2) conformance checking shows a noticeable wide gap in runtime between the encodings, although remaining manageable runtime-wise for all the three ASP encodings. This confirms our intuitive explanation. In the case of the query checking task, being a classic guess & check ASP encoding, the performance depends on many factors, and it is difficult to pinpoint - size of the search space (quadratic in $|\mathscr{A}|$), order of branching while searching the model, number of constraints grounded, the nature itself of traces in the log. In this context, trade-offs between runtime and memory consumption are expected.

7.4.2 Behavior on longer traces

Lastly, we analyze how the automata-based and direct ASP encodings scale w.r.t the length of the traces. We generate synthetic event logs of 1000 traces of fixed length (up to 1000 events) for each constraint, and we perform conformance checking with respect to a single constraint, chosen among the *Response* and *Precedence* hierarchies. Table 7.6 reports the result of our experiment. We compare only automata and direct encodings, since the previous analysis make it evident the syntax tree encoding is subpar due to memory consumption. The direct encoding yields consistent advantage time-wise w.r.t the automata encoding, and about half of peak memory consumption. Recall this is for a *single* constraint, and

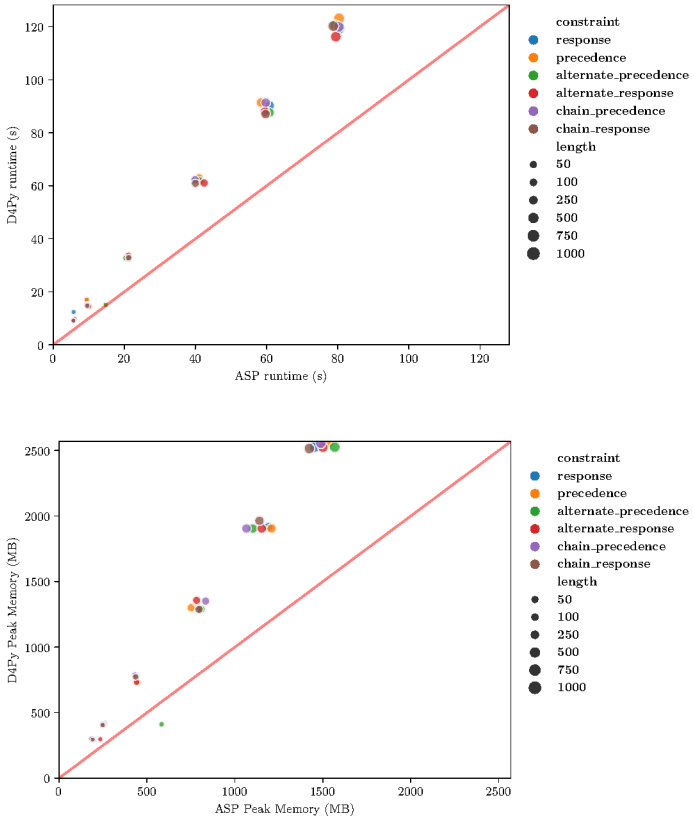


Figure 7.13: Comparison of runtime (upper) and peak memory usage (lower) for the ASP direct encoding and D4Py, on synthetic logs used in Table 7.6.

usually Declare models are composed of multiple Declare constraints, so this gap is expected to widen even more in conformance checking real Declare models. This is consistent with our previous experiments, see Table 7.2.

Moreover, we include a comparison between the direct encoding and D4Py over the same synthetic logs. For each synthetic log in Table 7.6, a point (x, y) in the scatter plot of Figure 7.13 means that the conformance checking task is solved in x seconds using the direct encoding and y seconds using D4Py[3]. Again, these results are consistent with observed behavior over real-world event logs.

[3]Note that time measurements of Figure 7.13 and Table 7.6 are not directly comparable, since in Table 7.6 input logs were stored as ASP facts, while in Figure 7.13 input logs were XES files.

	Response		Precedence	
$\|\pi\|$	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{A}}$	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{A}}$
50	(0.898 s, 175.922 MB)	(1.603 s, 463.297 MB)	(1.055 s, 69.926 MB)	(1.186 s, 316.082 MB)
100	(1.877 s, 103.254 MB)	(2.429 s, 326.863 MB)	(1.660 s, 110.941 MB)	(2.458 s, 318.984 MB)
250	(4.249 s, 216.777 MB)	(7.589 s, 573.191 MB)	(4.263 s, 316.070 MB)	(6.489 s, 660.438 MB)
500	(9.111 s, 401.844 MB)	(15.604 s, 871.145 MB)	(8.701 s, 385.801 MB)	(16.096 s, 845.754 MB)
750	(13.121 s, 505.500 MB)	(29.809 s, 1427.172 MB)	(12.873 s, 497.191 MB)	(28.641 s, 1409.781 MB)
1000	(18.693 s, 784.520 MB)	(40.228 s, 1725.414 MB)	(17.416 s, 764.488 MB)	(39.589 s, 1717.375 MB)
	Alt. Response		**Alt. Precedence**	
$\|\pi\|$	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{A}}$	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{A}}$
50	(1.397 s, 215.266 MB)	(1.679 s, 461.453 MB)	(1.016 s, 635.871 MB)	(1.751 s, 741.715 MB)
100	(2.101 s, 164.242 MB)	(2.476 s, 461.355 MB)	(6.232 s, 635.867 MB)	(3.045 s, 741.715 MB)
250	(4.636 s, 207.594 MB)	(7.735 s, 562.719 MB)	(4.653 s, 207.570 MB)	(8.123 s, 855.000 MB)
500	(9.419 s, 404.379 MB)	(16.354 s, 869.719 MB)	(9.121 s, 401.465 MB)	(17.967 s, 995.680 MB)
750	(14.057 s, 503.562 MB)	(30.529 s, 1429.836 MB)	(14.646 s, 737.258 MB)	(31.910 s, 1577.168 MB)
1000	(19.415 s, 783.363 MB)	(43.013 s, 1726.945 MB)	(20.054 s, 823.027 MB)	(49.133 s, 1714.512 MB)
	Chain Response		**Chain Precedence**	
$\|\pi\|$	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{A}}$	$\text{ASP}_{\mathscr{D}}$	$\text{ASP}_{\mathscr{A}}$
50	(1.116 s, 69.902 MB)	(1.245 s, 349.805 MB)	(0.922 s, 166.375 MB)	(1.245 s, 452.973 MB)
100	(1.818 s, 349.098 MB)	(2.563 s, 587.199 MB)	(1.798 s, 166.371 MB)	(2.610 s, 452.969 MB)
250	(4.972 s, 349.805 MB)	(6.891 s, 655.785 MB)	(4.678 s, 206.789 MB)	(7.862 s, 559.523 MB)
500	(9.527 s, 378.543 MB)	(16.778 s, 859.781 MB)	(9.882 s, 405.285 MB)	(16.104 s, 850.094 MB)
750	(13.120 s, 720.094 MB)	(27.913 s, 1539.625 MB)	(13.165 s, 744.195 MB)	(28.160 s, 1605.547 MB)
1000	(17.960 s, 772.133 MB)	(40.541 s, 1731.586 MB)	(17.837 s, 794.250 MB)	(41.960 s, 1739.156 MB)

Table 7.6: Comparing metrics between different ASP encodings, with increasing trace lenghts. An entry is an ordered pair (t, m) where t is the runtime (seconds), m the memory peak (MBs).

7.5 Discussion

Declare is a declarative process modeling language, which describes processes by sets of temporal constraints. Declare specifications can be expressed as LTL_p formulae, and traditionally have been evaluated by executing the equivalent automata [57], regular expressions [44], or procedural approaches [106]. Translation-based approaches (on automata, or syntax trees) are at the foundation of existing ASP-based solutions [40, 99]. We propose a novel direct encoding of Declare in ASP that is not based on translations. Moreover, for the first time, we put on common ground (regarding input fact schema) and compare available ASP solutions for conformance checking and query checking. Our experimental evaluation over well-known event logs provides the first aggregate picture of the performance of the methods considered. The results show that our direct encoding, albeit limited to Declare, outperforms other ASP-based methods in terms of execution time and peak memory consumption and compares favourably with

dedicated libraries. Thus, ASP provides a compact, declarative, and efficient way to implement Declare constraints in the considered tasks. Interesting future avenues of research are to investigate whether this approach can be extended to *data-aware* [106, 120, 127] variants of Declare, that take into account data attributes associated to events in a trace, as the more general case of data-aware extensions of LTL_f [86, 76, 75] and other Process Mining tasks.

Chapter 8

Passive Learning

In this chapter we propose two ASP-based approach to the LTL_f passive learning problem, that is inferring LTL_f properties from positive and negative examples. The first approach is based on abduction, while the second approach exploits the Inductive Logic Programming system ILASP. We conclude the chapter with an use case on learning network protocols temporal invariants from labeled execution examples.

8.1 Introduction

Passive learning of LTL_f formulae [130] refers to the problem of inferring an LTL_f formula from a set of system execution traces (some of which might be labeled as negative examples). The problem has been extensively studied in the literature [10, 83, 130, 139] motivated by the need of learning human-interpretable models that allow explaining the observed behavior of complex systems. Such automated approaches for inferring LTL_f specifications from data are becoming increasingly important, as manual development of specification is increasingly harder due to the complexity, dynamic changes and evolution of current systems. Surprisingly, the problem has been proven to be challenging for model-based techniques. In fact, despite its combinatorial nature, the current state-of-the-art [83] is exhaustive-search-based systems that do not apply any pruning to reduce the search space, beyond efficiently evaluating formulae over traces. Although these approaches may be acceptable in certain domains, their reliance on exhaustive search poses limitations in domains with numerous atomic propositions or lengthy target formulae. Specifically, the search space of LTL_f formulae grows exponen-

tially both in the size of the formula being searched and in the number of atomic propositions, thereby preventing their applicability. Thus, investigating the use of techniques capable of exploiting multiple examples to prune the search space, such as ILP, is an important step towards solutions to the passive learning problem that is applicable to real-world problems.

In this chapter we propose an approach, based on inductive logic programming (ILP), for learning LTL_f formulae from positive and negative execution traces. Due to the combinatorial complexity of the problem, we make use of ILASP [103], a learning from answer sets system shown to generalize many of the existing ILP methods [104], and to be able to learn specifications expressed in answer set programming [103], a computational environment tailored to solve combinatorial optimization problems. ILP techniques have been applied to the task of learning of DECLARE [135] models [37], a declarative process modeling language whose patterns are defined by LTL_f formulae. However, to the best of our knowledge, this is the first ILP-based method for learning LTL_f formulae from finite execution traces without imposing syntactic restrictions or limiting the search to a set of patterns. Other ILP-based works tackled how to include LTL_f specifications among features used in learning algorithms [143], as well as applications to learning *temporal logic programs* [95].

In this chapter, we propose a novel representation of the problem of passive learning of LTL_f formulae in answer set programming [21]. We show how this representation can be adapted to express this problem as a learning from answer set task and as an SAT-based inference task. We then conduct an experimental evaluation of our ILP-based approach over a set of event logs on cellular networks' attacks (already used in the literature to benchmark systems), and compare its performance with respect to the existing SAT-based methods. Our evaluation shows that our approach improves upon previous SAT- and SMT-based techniques, as implemented by the SySLite [10] system. The contributions herein presented are the first stepping stone towards the development of ILP-based systems for LTL_f passive learning that can be used in real-world settings. To this end, our future work will address the scalability of our proposed method to demonstrate its advantage versus exhaustive search methods, and to solve passive learning problems where execution traces refer also to structured properties of the data.

8.1.1 Passive Learning problem

The problem of passive learning of LTL_f formulae, introduced in [130], refers to the challenge of automatically inferring LTL_f formulae from observed traces of

system behavior, usually partitioned into sets of positive and negative examples, such that the positive traces are models of the formula and the negative traces are not models of the formula. This can be formally defined as follows:

Definition 4 (PL_{LTL_f} Passive Learning Task)**.** *Let $\mathscr{P}$ be a set of propositional symbols. A* PL_{LTL_f} passive learning task *is a tuple* $\text{PL}_{LTL_f} = (\mathscr{P}, \mathscr{E}^+, \mathscr{E}^-)$ *where $\mathscr{E}^+$ is a set of traces over $\mathscr{P}$ called* positive traces*, and $\mathscr{E}^-$ is a set of traces over $\mathscr{P}$ called* negative traces *such that $\mathscr{E}^+ \cap \mathscr{E}^- = \emptyset$. A* solution *of a* PL_{LTL_f} *task is an* LTL_f *formula φ, written in $\mathscr{P}$, such that (i) $\pi \models \varphi$, for every $\pi \in \mathscr{E}^+$; and (ii) $\pi \not\models \varphi$, for every $\pi \in \mathscr{E}^-$.*

Note that, a PL_{LTLf} passive learning task always accepts a *trivial* solution given by the formula $\phi = \bigvee_{\pi \in \mathscr{E}^+} \bigwedge_{i \in [0,\ldots,|\pi|-1]} \mathsf{X}^i(\bigwedge_{p \in \pi_i} p \wedge \bigwedge_{p \notin \pi_i} \wedge \neg p) \wedge \neg \mathsf{X}^{|\pi|} true$, where X^i denotes the nested application of the X operator i times. We therefore focus on *optimal solutions* of a PL_{LTLf} passive learning task, as defined below.

Definition 5 (Optimal solution of a PL_{LTL_f} Passive Learning Task)**.** *Let* $\text{PL}_{LTL_f} = (\mathscr{P}, \mathscr{E}^+, \mathscr{E}^-)$ *be a* PL_{LTL_f} *passive learning task. An LTL_f formula φ, written in $\mathscr{P}$, is an* optimal solution *of* PL_{LTL_f} *if and only if φ is a solution of* PL_{LTL_f} *and there is no LTL_f formula φ' written in $\mathscr{P}$ that is a solution of* PL_{LTL_f} *and* $|\varphi'| < |\varphi|$.

Solving a PL_{LTL_f} passive learning task means searching for an optimal (i.e. minimal-size) solution with respect to a fixed set of propositional symbols $\mathscr{P}$.

8.2 Formalizing LTL_f semantics in ASP

In this section, we present an encoding for evaluating LTL_f formulae over traces, by embedding LTL_f semantics into a normal logic program. We present our encoding into different subsections, addressing how to represent traces, formulae, and temporal logic operators' evaluation rules in logic programs.

Encoding traces. We assume traces to be uniquely indexed by integers, and in particular we will assume that a trace π^i is referred to by the integer i. We encode a trace π^i over $\mathscr{P}$ as a set of facts matching the predicates `trace/3` and `trace/2`. The atom $trace(i,t,a)$ models that $a \in \pi^i_t$. In order to be able to model empty states, we introduce the atoms $trace(i,t)$ for $0 \leq t < |\pi^i|$. Further information about the trace can be encoded by auxiliary predicates which refer to the trace identifier in the first term. In our case, the only additional information to encode is

whether each trace is a positive, $\pi^i \in \mathscr{E}^+$, or negative, $\pi^i \in \mathscr{E}^-$, example, and this is done through the atoms $pos(i), neg(i)$, respectively. We denote by $P(\pi)$ the set of facts that encode the trace π. With a slight abuse of notation, we will also denote by $P(\mathscr{E})$ the set of facts that encode the set of traces $\mathscr{E}$, that is $P(\mathscr{E}) = \bigcup_{\pi^i \in \mathscr{E}} P(\pi^i)$.

Example 8.2.1. *Consider the trace* $\pi^0 = \{a\} \cdot \{a,b\} \cdot \{\}$*, and assume* $\pi^0 \in \mathscr{E}^+$*. This is encoded by the following facts:*

```
trace(0,0,a). trace(0,1,a). trace(0,1,b).
trace(0,0). trace(0,1). trace(0,2). pos(0).
```

Encoding formulae. We encode a formula by reifying its syntax tree. The approach is similar to what we have described in Chapter 5, however this application requires a slightly different encoding since *we are not reifying a fixed formula*, but rather searching over possible formulae. In practice, we adopt a strategy which is similar to the SAT-based approach of [130], by means of the predicates `edge/2`, `order/3`, `label/2` and `node/1`. The predicates `node/1`, `edge/2` model trees in a natural way, where we use natural numbers to identify nodes. The atoms $node(x), edge(y,x)$ model that x is a node of the tree, and that y is its parent. The predicate `label/2` models logic operators (or propositions) associated with each node in the tree. An atom $label(x, j)$ encodes that the node x is labeled with $j \in \mathscr{O} \cup \mathscr{P}$, where $\mathscr{O}$ is the set of available temporal and propositional logic operators. For the rest of the chapter, we assume without loss of generality that $\mathscr{O} = \{\neg, \vee, \wedge, \mathsf{X}, \mathsf{U}, \rightarrow, \mathsf{F}, \mathsf{G}\}$. The atom $order(i, lhs, rhs)$ distinguishes between left and right of node i, which is needed for the evaluation of the non-commutative operators $\{\mathsf{U}, \rightarrow\}$. We denote by $P(\varphi)$ the set of facts which encode a formula φ. Without loss of generality, we will assume that the node identified by 1 is the root of a formula's tree.

Example 8.2.2. *Consider the formula* $(\mathsf{X}a)\ \mathsf{U}\ b$*. This is encoded by the following facts:*

```
node(1..4). label(1, until). label(2, next). label(3, a).
label(4,b). edge(1,2). edge(1,4). edge(2,3). order(1,2,4).
```

Encoding semantics. We encode the semantics of each supported operator by simulating the recursive application of the `xnf`$(\cdot)$ transformation by means of normal recursive rules. In particular, each subformula is identified by the node identifier of its root in the syntax tree. The atom $holds(i,t,x)$ models that the $\pi^i, t \models \varphi_x$ where φ_x is the subformula of φ rooted in the node identified by integer

x. The atom $last(i,t)$ models that $|\pi^i| = t$, that is π^i_t is the last state of π^i. The definition of these rules, which we denote by P_{LTL_f}, follows the $\texttt{xnf}(\cdot)$ definitions in Section 3 and is similar to the encoding of Chapter 5, however over different predicates.

```
holds(TID, T, X)
:- label(X, A), trace(TID, T, A).
holds(TID, T, X)
:- label(X, next), edge(X, Y), holds(TID, T+1, X), not last(TID, T).
holds(TID, T, X)
:- label(X, until), order(X,LHS,RHS), holds(TID, T, RHS).
holds(TID, T, X)
:- label(X, until), order(X,LHS,RHS), holds(TID, T, LHS), holds(TID,
    T+1, X).
holds(TID, T, X)
:- label(X, and), order(X,A,B), holds(TID, T, A), holds(TID, T, B).
holds(TID, T, X)
:- label(X, or), edge(X, A), holds(TID, T, A).
holds(TID, T, X)
:- label(X, neg), edge(X, Y), not holds(TID, T, Y), trace(TID, T).
```

8.3 LTL_f passive learning in plain ASP

A first way to model the passive learning problem is to frame it as an abduction problem in ASP [63], where the set of abducibles corresponds to facts matching the predicates `node/1`, `edge/2`, `label/2`, which reify into facts the syntax tree of an LTL_f formula. The goal of the abduction task is to find an LTL_f formula φ for which all $e \in \mathscr{E}^+$ we have that $e \models \varphi$ and for all $e \in \mathscr{E}^-$ we have that $e \not\models \varphi$. The following rules encode, denoted P_{tree} and P_{label} respectively, the abduction of an LTL_f formula of size n:

```
node(1..n).
pair(X,Y) :- node(X), node(Y), X < Y.
1 { edge(Y,X): pair(Y,X) } 1 :- node(X), X > 1.
reach(1). reach(X) :- edge(Y,X), reach(Y).
:- node(X), not reach(X).
id(1,(0,0)).
id(V,(U,V*V+U)) :- edge(U,V).
:- id(I,RI), id(I+1,RJ), RI >= RJ.
```

Listing 8.1: The logic program P_{tree}

Each answer set of P_{tree} encodes a tree of size n. Due to the combinatorial nature of the problem, we introduce basic symmetry-breaking constraints in order to avoid generating isomorphic trees. In particular, we assume that each node has an identifier that is greater than its parent's identifier. Furthermore, the last constraints force the node identifiers to respect the order of a BFS traversal of the tree starting from the root node, which is adapted from [71] to our setting which does not require labeled edges.

```
unary_label(neg; next; eventually; always).
binary_label(and; until; or).
leaf(X) :- node(X), not edge(X,_).
unary(X) :- node(X), not leaf(X), not binary(X).
binary(X) :- edge(X,Y), edge(X,Y'), Y < Y'.
1 {label(X,L): unary_label(L) } 1 :- unary(X).
1 {label(X,L): binary_label(L) } 1 :- binary(X).
proposition(A) :- trace(_,_,A).
1 {label(X,L): proposition(L) } 1 :- leaf(X).
```

Listing 8.2: The logic program P_{label}

Each answer set, projected on the predicates `node/1`, `edge/2`, `label/2`, `order/3`, corresponds to the encoding of an $\mathrm{LTL_f}$ formula. In order to encode the goals of the abduction, we have to constrain the generated formulae to accept positive examples and reject negative examples. The following rules, P_{goal}, can encode the goal of the abduction:

```
sat(TID) :- holds(TID,1,1).
unsat(TID) :- trace(TID,_), not sat(TID).
:- sat(TID), neg(TID).
:- unsat(TID), pos(TID).
```

Listing 8.3: The logic program P_{goal}

Thus, every answer set, projected onto the predicates `label/2`, `edge/2`, `node/1` and `order/3` as shown in Example 8.2.2, of the logic program $P_{tree} \cup P_{label} \cup P_{goal} \cup P(\mathscr{E}^+) \cup P_{\mathrm{LTL_f}} \cup P(\mathscr{E}^-)$, can be mapped to an $\mathrm{LTL_f}$ formula of size n accepting all positive examples and rejecting all negative examples, that is a solution to the passive learning problem instance $(\mathscr{P}, \mathscr{E}^+, \mathscr{E}^-)$. In order to find an optimal, size-minimal, solution the above approach can be easily implemented in an incremental fashion using CLINGO incremental solving APIs, in order to search for solutions of increasing formula length n. This allows generating trees incrementally, rather than up-front for a fixed size n. Thus, the first solution to be found will be a minimal one. We omit this for brevity since it is very similar to well-known examples in literature [94].

8.4 LTL_f passive learning using ILASP

In this section, we frame an instance $(\mathscr{P}, \mathscr{E}^+, \mathscr{E}^-)$ of the passive learning problem as a context-dependent learning from answer sets task, by providing suitable mode biases, background knowledge and encoding of the examples $\mathscr{E}^+ \cup \mathscr{E}^-$.

Mode bias. We use the following mode bias, which we report in the input language of ILASP [105]. This specifies available operators (via the type `op`), the maximum size of the formula (via the type `node_id`) to search for as well as the available atomic propositions (via the type `atom`) (here omitted, since it depends on the traces in $\mathscr{E}^+ \cup \mathscr{E}^-$). We assume a constant declaration of type `atom` for each $p \in \mathscr{P}$. Analogously to the plain ASP encoding, the predicates `edge/2`, `label/2` encode the labeled syntax tree of a LTL_f formula. The constant `n` below refers to the maximum size of the target formula to be inferred. Notice the mode bias consists only of `#modeh` directives: thus the inductive solution will be a set of facts matching the predicates in Example 8.2.2, which can be interpreted as the encoding of a LTL_f formula.

```
#constant(node_id, 1..n).
#constant(op, next).
#constant(op, until).
#constant(op, eventually).
#constant(op, always).
#constant(op, and).
#constant(op, neg).
#constant(op, or).
#constant(op, implies).
#modeh( edge(const(node_id), const(node_id)) ).
#modeh( label(const(node_id), const(op)) ).
#modeh( label(const(node_id), const(atom)) ).
```

Background knowledge. We assume the logic program P_{LTL_f} to be the background knowledge of our LAS task. We assume an atom $\mathit{proposition}(a)$ for each $a \in \mathscr{P}$. Basically, since the evaluation of the inductive solution over examples is automatically handled by the ILASP system, it is not needed to handle the evaluation of a formula over multiple traces in the encoding. Thus, we drop the first term of the predicates `holds/3`, `last/2`, `sat/1`, `unsat/1`.

Encoding constraints over search space. A candidate inductive solution $H \subseteq S_M$ models a syntax tree encoding a LTL_f formula according to the previously defined fact schema, using predicates `node/1`, `edge/2`, `label/2`. The following ASP program represents the context of a special positive example e^*, which imposes

syntactical constraints over the syntax tree of the candidate inductive solution. This addresses the same constraints as the ones in P_{tree} and P_{label}.

```
% Nodes are terms that appear in edge/2 or first term of label/2
node(X) :- label(X,_).
node(X) :- edge(_,X).
node(X) :- edge(X,_).
% Don't skip nodes
node(X) :- node(X+1), X >= 1.
% The syntax tree of the formula must be connected
reach(1). reach(T) :- edge(R,T), reach(R).
:- node(X), not reach(X).
:- node(X), not edge(_,X), X > 1.
% Bounded fan-out for logic operators
:- node(X), 3 #count { Z: edge(X,Z) }.
% Exactly one label per node
:- node(X), not label(X,_).
:- label(X,A), label(X,B), A < B.
% Syntax tree admits a BFS-indexing
id(1,(0,0)).
id(V,(U,V*V+U)) :- edge(U,V).
:- id(I,RI), id(I+1, RJ), RI >= RJ.
:- id(I+1,RI), id(I,RJ), RI <= RJ.
% Labels must match node's arity
arity(X,0) :- node(X), not edge(X,_).
arity(X,2) :- node(X), edge(X,Y), edge(X,Y1), Y < Y1.
arity(X,1) :- node(X), not arity(X,0), not arity(X,2).
:- arity(X,N), label(X,Y), not symbol(Y,N).
symbol(A,0) :- proposition(A).
symbol(next,1). symbol(until,2). symbol(eventually,1). symbol(always,1).
symbol(neg,1). symbol(and,2). symbol(or,2). symbol(implies,2).
```

Encoding examples. We encode $\pi \in \mathscr{E}^+$ as a positive example which includes the constant `sat` in the inclusion set and an empty exclusion set, while $\pi \in \mathscr{E}^-$ includes the constant `unsat` in the inclusion set and an empty exclusion set. In both cases, the context of the example consists of $P(\pi)$ for $\pi \in \mathscr{E}^+ \cup \mathscr{E}^-$. Similarly, we drop the term `TID` from the predicates `trace/3`, `trace/2` as for the background knowledge program.

8.5 Evaluation

This section reports an experimental evaluation that aims at assessing both ASP-based approaches presented in the previous section, and to compare existing solutions based on SAT and SMT. In the experimental evaluation we used the event logs pertaining to the passive learning of *attack signatures* on cellular networks, and partitioned each event log into positive and negative traces. *Signatures* are formulae of kind $\mathsf{G}\varphi$, which characterize the positive traces of each log on each time instant. A comprehensive description of each log is available in a technical report [10]. We compare our ILASP-based solution with our plain ASP solution in order to assess whether ILP can help in this setting, as well as other SAT-based approaches previously implemented in literature, referring to their implementation in the SySLite system. Experimental data and full encodings are available on GitHub (`https://github.com/ilp2023-27/data`).

Execution environment. All experiments were executed on an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 512GB RAM machine, using CLINGO version 5.4.0,

Table 8.1: Execution time in seconds for compared methods over the event logs. Best execution time is in bold. T.O denotes timeout (execution time >3600s).

Event Log	SySLite$^{\text{sygus}}$	ILASP 2i	Abduction	SySLite$^{\text{sat}}$	SySLite$^{\text{guided_sat}}$
AKA	144.2	**62.542**	612.31	T.O.	T.O.
AF	**1.98**	3.32	4.705	360.7	T.O.
BT	2.17	**1.437**	13.239	659.23	133.19
CWP	22.01	**7.739**	148.891	T.O.	T.O.
DSP	T.O.	T.O	T.O	T.O.	T.O.
EMM	**4.64**	5.78	7.207	1155.18	T.O.
FI	51.59	**19.865**	510.189	T.O.	T.O.
GLBA	26.86	**13.606**	183.542	T.O.	T.O.
HIPPA-1	**25.96**	31.994	194.694	T.O.	T.O.
HIPPA-2	2.41	**1.693**	15.962	707.43	153.41
IM	**4.04**	4.483	5.837	977.77	T.O.
IMSI-1	3.72	**3.584**	4.877	774.34	T.O.
IMSI-2	6.89	**4.782**	8.092	1144.58	T.O.
MR	995.85	**55.622**	T.O	T.O.	T.O.
NE	**2.43**	4.301	4.706	480.18	T.O.
NA	2.54	**2.545**	2.8	1877.79	T.O.
IMSI-3	16.55	**11.652**	98.643	T.O.	T.O.
RLF	560.44	**23.266**	T.O	T.O.	T.O.

Python 3.10, ILASP 4.2.0 and the version of SySLite available in the authors' repository. All experiments were run in parallel using GNU Parallel [150]. We report execution time in seconds, with a timeout of 3600 seconds execution time on each event log. Currently, the most appropriate ILASP version for the task is ILASP 2i, which is the one we use to run the experiments.

Data. The dataset is composed by 18 logs: AKA Bypass (AKA), uthentication Failure (AF), Bank Transaction (BT), Chinese Wall Policy (CWP), Dynamic Separation Policy (DSP), EMM Information (EMM), Financial Institute (FI), GLBA, HIPPA 16450A2 (HIPPA-1), HIPPA 16450A3 (HIPPA-2), Identity Malformed (IM), IMSI Catching (IMSI-1), IMSI Cracking (IMSI-2), Measurement Report (MR), NULL Encryption (NE), Numb Attack (NA), Paging with IMSI (IMSI-3), RLF Report (RLF). We considered each log as an instance for the passive learning problem. Since SySLite algorithms target pure-past formulae, we reverse each trace in the log in order to use our encodings and samples2ltl tool. In this way, all approaches are able to learn the same formulae up to dual relabeling of temporal operators involved in the formulae. In particular, we indicate by SySLite^L, $L \in \{\text{sygus}, \text{sat}, \text{sat\_guided}\}$ the different algorithms available in SySLite, implementing SMT- and SAT-based algorithms. In particular, the `sygus` algorithm is SMT-based, exploiting bit-vector theories for efficient computations. `ILASP 2i` column refers to our ILASP encoding, and `Abduction` column refers to our (incremental) plain ASP encoding that uses abduction. Since the SySLite tool targets pure-past formulae rooted on the *historically* operator (the pure-past dual of G), we (i) invert the traces before defining our LAS and ASP encoding; (ii) add to our encoding the constraint that target formulae must be rooted in G. Since ILASP currently does not support incremental solving (with respect to the definition of the hypothesis space), but rather solves a complexity-wise harder optimization task, we assume the maximum size of the target formula is known beforehand. All systems support the same set of temporal logic operators $\{\mathsf{X}, \mathsf{U}, \mathsf{F}, \mathsf{G}, \wedge, \vee, \neg\}$. We run the different algorithms available in SySLite with a timeout of one hour, along with ILASP and our abductive encoding, on a suite of event logs. The solution based on ILASP compares favorably, as shown in Table 8.1, with the algorithms implemented in SySLite, and even outperforms it on some event logs. Our plain ASP solution is noticeably slower than the `sygus` SMT-based algorithm and ILASP alike. This suggests that ILP based on ILASP might be a viable approach to scale beyond current model-based techniques without over-relying on pure enumerative approaches.

8.6 Discussion

In this chapter, we presented an ILP approach based on the ILASP system for the passive learning of LTL_f formulae. Our approach embeds LTL_f semantics into a normal logic program, similar to previous works based on SAT, which is provided as the background knowledge. We outperform SAT-based techniques as implemented in SySLite and compare favorably against its best-performing SMT-based syntax-guided enumeration algorithm. We also implement an abduction-based algorithm based on ASP, proving our performance gains are due to ILASP's inductive loop rather than the use of plain ASP with respect to SAT or SMT encoding. As future work we plan to improve the scalability of our proposed method and extend it to take into account data attached to events that occur during the system's execution. A comparison with the approach of Ghiorzi et al. [83] is also in our plans to possibly demonstrate there is an advantage versus exhaustive search methods. Another interesting extension we are interested in, which would extend the applicability of passive learning in real-world settings, is to apply our techniques to *noisy domains* [73, 129] (where traces or their labels might contain errors) by exploiting ILASP's support for *example's penalties* and ASP optimization techniques. It would also be interesting to check whether a compilation-based ASP system [121] can be beneficial to improve the performance of the abductive approach, where we conjecture the number of symbols generated by evaluating candidate solutions over the event log is one of the causes of performance degradation.

Chapter 9

Related works

In this chapter, we discuss related works of each of our contribution. We dedicate a specific subsection to each contribution of the thesis.

9.1 Bounded Satisfiability

Linear Temporal Logic (LTL) [138] is a type of modal logic developed in the 1970s as a formal method for verifying the correctness of computer programs and reactive systems [138]. Linear Temporal Logic on finite traces (LTL_f) [55] is a variant of standard Linear Temporal Logic (LTL) that has found applications in various fields of Artificial Intelligence where reasoning about finite system execution is more natural [31, 69, 50, 43, 151]. Although the computational complexity of all the main problems, including satisfiability checking, remains unchanged when shifting to finite trace semantics [68], dealing with finite traces can lead to more efficient systems in practice.

Early methods to LTL_f satisfiability involved translating LTL_f formulas into LTL formulas to leverage existing LTL satisfiability checkers [55]. Soon, ad-hoc systems specifically designed for finite trace semantics were developed [68, 78, 107], demonstrating faster performance compared to repurposed LTL tools. Ad-hoc systems' approaches primarily fall into two categories: tableaux-based techniques and SAT-based methods. Tableaux-based techniques construct a tableau for the given LTL_f formula, systematically exploring possible extensions of partial models until either a model is found or all possibilities are exhausted [77, 78, 108]. These methods are complete, meaning they can prove both satisfiability and unsatisfiability, but they often struggle with the practical efficiency needed

for large or complex formulas. On the other hand, SAT-based approaches [68, 107] typically involve translating the LTL_f formula into a propositional formula, which is then solved using a SAT solver. These methods are usually incomplete but faster, making them suitable for bounded model checking scenarios where the goal is to find counterexamples up to a certain length. In more detail, the first ad-hoc tool for LTL_f satisfiability checking was Aalta-finite, which uses a tableaux-based technique [108]. This method was later enhanced by a conflict-driven algorithm known as CDLSC [107]. CDLSC reduces the satisfiability checking to a path-search problem within the LTL_f transition system which is built incrementally during search. A pure SAT-based approach is LTL2SAT [68], which directly translates the temporal formula into a propositional formula parameterized on the length of the model, to be fed in a SAT solver. LTL2SAT also implements early termination strategies for classes of formulas where satisfiability can be achieved in polynomial time. The BLACK [78] system merges tableaux and SAT-based approaches by implicitly building the tableau as a SAT formula, thus combining the strengths of both methods.

The encoding in ASP used in this work has similarities and overlaps with encodings proposed for other relevant AI tasks, such as injecting temporal knowledge in planning [149], business process verification [87], passive learning of LTL_f formulae [91], reasoning about inconsistencies [47] in LTL_f, and applications to declarative process mining [98, 99].

9.2 Minimal Unsatisfiable Cores Extraction

The task of computing MUCs has been considered, under different names, for several representation languages including propositional logic [110], constraint satisfaction problems [123], databases [122], description logics [145], and ASP [8] among many others. Peñaloza [133] provides a general overview of the task and known approaches to solve it.

Although the task was briefly studied for LTL [13, 146], it was only recently considered for the specific case of LTL_f [132, 144]. 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 [147] and later on using ASP solvers [134]. Our approach falls into the latter class.

Our reduction to ASP is inspired on bounded satisfiability checking, previously used for SAT-based satisfiability checking [68, 107], alongside an incremental approach that verifies the (non-)existence of models up to a certain length.

Similar notions have been explored in the context of *declarative process monitoring* [115]. Differently from conformance checking, which assumes traces to be complete, process monitoring attempts to detect—at the earliest possible point in time— when a trace violates a process model.

Lastly, a paraconsistent semantic for inconsistent LTL_f specifications have been proposed [47], that relies on the number of minimal unsatisfiable cores to quantitatively assess degree of unsatisfiability in specifications [46]. In particular, Kuhlmann and Corea [97] propose also an ASP-based approach to enumerate minimal unsatisfiable cores of LTL_f formulae *assuming a known upper bound on trace length*, while our approach iteratively refines such an upper bound.

9.3 Passive Learning

The seminal work [130] defines two algorithms for the passive learning problem of LTL_f formulae. One of them introduces SAT solvers as practical tools for LTL_f passive learning, encoding formulae's syntax trees and their evaluation over traces as a satisfiability problem, while the other exploits a decision tree to propositionally combine smaller LTL_f formulae, addressing scalability but dropping the "optimality" of the solution (in terms of formula size). Another approach [139], in order to improve scalability, targets the *directed* fragment of LTL_f, which however is not as expressive as LTL_f as it is unable to express the *until* temporal operator. Other SAT-based works target equivalent formalisms (such as *alternate automata* [29]), that can then be translated into LTL_f formulae. The SySLite [10] system targets pure-past LTL_f formulae of the form $\mathsf{H}\varphi$, where H is the past version of the operator G. It implements different SAT-based algorithms (the ones in [141] as well as SMT-based *syntax-guided synthesis* [141] enumeration which exploits bit-vector theories for fast evaluation. Recently, an approach based solely on a highly optimized exhaustive search has been proposed [83] that enumerates formulae of increasing size and performs pruning on syntactic and semantic criteria on a single trace at a time. A direct comparison with [83] could not be implemented since the tool does not expose an API to force learning of specific formulae, e.g. starting with G. Thus implementing a comprehensive empirical comparison, in the specific case of learning signatures, would require a modification of the tool of [83]. From the theoretical standpoint, computational complexity-wise, the

authors of [67] identify multiple fragments of LTL for which the passive learning problem is already NP-complete, and sample complexity-wise (that is, how many examples are required to guarantee a given formula is learned) it is known [29] passive learning of arbitrary LTL_f formulae can be done with an exponential number of examples under some conditions.

In Chapter 8, we presented an ILP approach based on the ILASP system for the passive learning of LTLf formulae. Our approach embeds LTL_f semantics into a normal logic program, similar to previous works based on SAT, which is provided as the background knowledge. We outperform SAT-based techniques as implemented in SySLite and compare favorably against its best-performing SMT-based syntax-guided enumeration algorithm. We also implement an abduction-based algorithm based on ASP, proving our performance gains are due to ILASP's inductive loop rather than the use of plain ASP with respect to SAT or SMT encoding. As future work we plan to improve the scalability of our proposed method and extend it to take into account data attached to events that occur during the system's execution. A comparison with the approach of Ghiorzi et al. [83] is also in our plans to possibly demonstrate there is an advantage versus exhaustive search methods. Another interesting extension we are interested in, which would extend the applicability of passive learning in real-world settings, is to apply our techniques to *noisy domains* [73, 129] (where traces or their labels might contain errors) by exploiting ILASP's support for *example's penalties* and ASP optimization techniques. It would also be interesting to check whether a compilation-based ASP system [121] can be beneficial to improve the performance of the abductive approach, where we conjecture the number of symbols generated by evaluating candidate solutions over the event log is one of the causes of performance degradation.

9.4 Declare-based process mining

Answer Set Programming (ASP) [21, 81, 131] has shown promise in planning to inject domain-dependent knowledge rules, resembling predefined patterns, encoded via an action theory language into ASP [149]. Researchers additionally explored injecting temporal knowledge, expressed as LTL formulae, in answer set planning [149]. An extension of the ASP language with temporal operators was proposed [25], and other applications of logic programming to solve process mining tasks have been developed in earlier works [38, 100]. The works in which ASP has been used to tackle various computational tasks in Declare-based Pro-

cess Mining [92] are closer to the contents of this chapter. Chiariello, Maggi, and Patrizi [40] propose a solution based on the well-known LTL_f-to-automata translation [55]. Kuhlmann, Corea, and Grant [99] suggests a method for encoding the semantics of temporal operators into a logic program, enabling the encoding of arbitrary LTL_f formulae, by a reification of their syntax tree. Some other recent studies [35, 36] tackle a slightly different objective by using ASP to directly encode Declare constraints but with the focus of distinguishing between normal and deviant process traces.

9.5 Answer Set Programming and Temporal Reasoning

At the intersection of temporal reasoning and Answer Set Programming lies also *temporal answer set programming* [24, 25, 26], an extension of ASP with temporal operators akin to LTL but in the logical framework of *temporal equilibrium logic* [5]. Similarly, the DatalogMTL [154] language extends Datalog with metric temporal operators. Recently a stable model semantics extension has been recently proposed by Walega et al. [153] for DatalogMTL.

However, while these works attempt to extend Answer Set Programming with temporal modalities, the subject of this thesis was to efficiently address LTL_f reasoning tasks in standard ASP.

Chapter 10

Conclusion and Future Work

This thesis has described some novel applications of Answer Set Programming to LTL_f reasoning tasks and declarative process mining. We briefly discuss and recap each of our own contributions, also with respect to our plans for future works.

10.1 Conclusion

This thesis proposes the application of Answer Set Programming to various reasoning tasks in the comntext of LTL_f. The core contributions of this thesis describe two ASP-based systems, `ltlf2asp` and `mus2muc`, that respectively address LTL_f bounded satisfiability and minimal unsatisfiable core enumeration. These come together with applications to the field of declarative process mining, related to passive learning and Declare-evaluation tasks.

Concerning bounded satisfiability, the `ltlf2asp` system is based on an ASP encoding (inspired by Fionda and Greco [68]) that reifies the syntax tree of an input formula, searching for a trace such that its bottom-up evaluation of the formula yields a satisficing root node. We compare both approaches, as well as provide an experimental evaluation with respect to other state-of-the-art systems.

Addressing the bounded satisfiability task in ASP certainly has some advantages. First of all, it yields an *uniform* encoding — our input is a set of facts that encode the formula as a directed acyclic graph, the *propositionalization* is implicit in the ASP grounding step. This is in stark contrast with SAT-based approaches, where clauses have to be explicitly encoded. This clearly improves readability. Surprisingly, this also improves performance, as our system scales much better with respect to other available system, especially on satisfiable formulae requir-

ing longer traces.

In the domain of minimal unsatisfiable cores enumeration, the `mus2muc` system leverages ASP minimal unsatisfiable subprograms to compute minimal unsatisfiable cores. It is mostly related to works in the SAT community to *domain agnostic* MUC enumeration [15]. While in these works are based on SAT solvers, in our approach the ASP solver *explicitly* encodes the LTL_f semantics, and has extra information on unsatisfiable (up to a point) traces. Candidate MUCs are later certified by standard LTL_f satisfiability solvers.

This introduces a novel technique for LTL_f MUC extraction, as far as we know the first one (i) which uses ASP; and (ii) which targets LTL_f enumeration. Although designed to tackle the enumeration task, `mus2muc` is also competitive with previous systems on the single MUC extraction task.

In relation to LTL_f passive learning, we proposed an inductive logic programming approach to passive learning of LTL_f formulae. In particular, we exploit the inductive logic programming system ILASP, based on the learning from answer sets approach [102].

Our approach scales better than previous SAT-based system, while being more declarative, readable and easily modifiable.

In the context of declarative process mining, we proposed an ASP-encoding that directly models the main Declare constraints semantics. Our approach supports the Declare conformance checking and query checking tasks. The raison d'étre for the work was to compare different ASP encodings for Declare conformance checking, namely syntax tree-based approaches [99] and automata-based approaches [40]. Our direct approach results to be better, both time- and memory-wise with respect to other ASP encodings in literature. To attest relevance of Answer Set Programming in this domain, we point out that most recent and advanced tooling for Declare actually already uses ASP to provide a flexibile, data-aware *log generator* [60].

Final Remark. Overall, this shows that ASP is a suitable formalism to handle LTL_f-related tasks, and a viable alternative to SAT solvers as a *backend* for LTL_f reasoners. Most importantly, this allows us to do so declaratively, yet without compromising on performance.

10.2 Future works

The contributions of this thesis can be expanded on multiple ends. Here, we discuss possible future avenues that could stem from our ASP-based approach on bounded satisfiability and minimal unsatisfiable cores enumeration.

Bounded Satisfiability. As for as future work is concerned, this work can be extended in different directions, which we plan to pursue. We start with the "low hanging fruits".

One of the benefits of this approach being ASP-based is that several extensions, that would require careful encoding in SAT and major refactorings, become trivial.

First, the approach can be extended to handle past operators. It is known in literature that, although past operators do not make LTL_f more expressive, there exist formulae that are *exponentially more succinct* including past operators (and viceversa).

Another interesting option for extension is concerned with support for *weighted semantics* for LTL_f [58]. Informally, each propositional symbol is attached a *value*; the objective given a formula φ is not only to find a model π for φ, but also to *minimize* a cost associated to π, in terms of the propositional symbols that appear in π. The framework assumes monotone cost functions; thus, optimal models are to be searched within the shortest models of φ. Optimizing with respect to sum of weights is easy to implement in `ltlf2asp`, as a few weight rules are sufficient. More complex cost functions would require custom propagators, which is however a commonly used (although quite advanced) feature of modern ASP systems.

A second possibility is to extend our results towards a complete approach. Our experiments show that available complete systems for LTL_f satisfisfiability greatly suffer for specifications where the shortest available model is quite long. This motivates ad-hoc systems for semi-decision. The natural extension is to appropriately *combine* efficient, "model-based" techniques for semi-decision with "complete" techniques that can prove unsatisfiabiliy. A natural way to investigate this is whether it is feasible to provide ASP-based implementations for tableaux-like techniques, as explored in [78, 108].

Minimal unsatisfiable cores. A possible avenue of future research is the investigation of parallel implementations, with multiple MUS generators and MUC

checkers. Also, we one might expand our experimental analysis to other probes, mainly tableaux-based [78] and Declare-based [39]. Another possibilty is to provide ad-hoc implementations for repairing LTL_f declarative specifications, exploiting duality between minimal unsatisfiable cores and minimal corrections sets [118].

Bibliography

[1] Wil M. P. van der Aalst. “Process Mining: A 360 Degree Overview”. In: *PM Handbook*. Vol. 448. LNIP. Springer, 2022, pp. 3–34.

[2] Wil M. P. van der Aalst. “The Application of Petri Nets to Workflow Management”. In: *Journal of circuits, systems, and computers* 8.1 (1998), pp. 21–66.

[3] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. “Declarative workflows: Balancing between flexibility and support”. In: *Computer Science-Research and Development* 23.2 (2009), pp. 99–113.

[4] Wil MP Van der Aalst and Anton JMM Weijters. “Process mining: a research agenda”. In: *Computers in industry* 53.3 (2004), pp. 231–244.

[5] Felicidad Aguado, Pedro Cabalar, Martín Diéguez, Gilberto Pérez, and Concepción Vidal. “Temporal Equilibrium Logic: A Survey”. In: *J. Appl. Non Class. Logics* 23.1-2 (2013), pp. 2–24.

[6] Mario Alviano, Giovanni Amendola, Carmine Dodaro, Nicola Leone, Marco Maratea, and Francesco Ricca. “Evaluation of Disjunctive Programs in WASP”. In: *LPNMR*. Vol. 11481. Lecture Notes in Computer Science. Springer, 2019, pp. 241–255.

[7] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. “WASP: A Native ASP Solver Based on Constraint Learning”. In: *LPNMR*. Vol. 8148. Lecture Notes in Computer Science. Springer, 2013, pp. 54–66.

[8] Mario Alviano, Carmine Dodaro, Salvatore Fiorentino, Alessandro Previti, and Francesco Ricca. “ASP and subset minimality: Enumeration, cautious reasoning and MUSes”. In: *Artif. Intell.* 320 (2023), p. 103931.

[9] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. “Towards a Theory of Declarative Knowledge”. In: *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988, pp. 89–148.

[10] M. Fareed Arif, Daniel Larraz, Mitziu Echeverria, Andrew Reynolds, Omar Chowdhury, and Cesare Tinelli. “SYSLITE: Syntax-Guided Synthesis of PLTL Formulas from Finite Traces”. In: *FMCAD*. 2020, pp. 93–103.

[11] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, et al. “The ForSpec temporal logic: A new temporal property-specification language”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2002, pp. 296–311.

[12] Gilles Audemard, Frédéric Koriche, and Pierre Marquis. “On tractable XAI queries based on compiled representations”. In: *17th International Conference on Principles of Knowledge Representation and Reasoning (KR’20)*. 2020.

[13] Franz Baader and Rafael Peñaloza. “Automata-Based Axiom Pinpointing”. In: *J. Autom. Reason.* 45.2 (2010), pp. 91–129.

[14] Fahiem Bacchus and Froduald Kabanza. “Planning for Temporally Extended Goals”. In: *Ann. Math. Artif. Intell.* 22.1-2 (1998), pp. 5–27.

[15] Jaroslav Bendík and Ivana Cerna. “Evaluation of Domain Agnostic Approaches for Enumeration of Minimal Unsatisfiable Subsets”. In: *LPAR*. Vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 131–142.

[16] Alessandro Berti, Sebastiaan J. van Zelst, and Daniel Schuster. “PM4Py: A process mining library for Python”. In: *Softw. Impacts* 17 (2023), p. 100556.

[17] Leopoldo E. Bertossi. “Database Repairs and Consistent Query Answering: Origins and Further Developments”. In: *PODS*. ACM, 2019, pp. 48–58.

[18] Leopoldo E. Bertossi, Anthony Hunter, and Torsten Schaub. “Introduction to Inconsistency Tolerance”. In: *Inconsistency Tolerance*. Vol. 3300. Lecture Notes in Computer Science. Springer, 2005, pp. 1–14.

[19] Armin Biere. “Bounded Model Checking”. In: *Handbook of Satisfiability*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 739–764.

[20] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. “Bounded model checking”. In: *Adv. Comput.* 58 (2003), pp. 117–148.

[21] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. “Answer set programming at a glance”. In: *Commun. ACM* 54.12 (2011), pp. 92–103.

[22] Gerhard Brewka, Matthias Thimm, and Markus Ulbricht. “Strong inconsistency”. In: *Artif. Intell.* 267 (2019), pp. 78–117.

[23] Andrea Burattin, Fabrizio Maria Maggi, and Alessandro Sperduti. “Conformance checking based on multi-perspective declarative process models”. In: *Expert Syst. Appl.* 65 (2016), pp. 194–211.

[24] Pedro Cabalar, Martín Diéguez, Torsten Schaub, and Anna Schuhmann. “Towards Metric Temporal Answer Set Programming”. In: *Theory Pract. Log. Program.* 20.5 (2020), pp. 783–798.

[25] Pedro Cabalar, Roland Kaminski, Philip Morkisch, and Torsten Schaub. “telingo = ASP + Time”. In: *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Proceedings*. Vol. 11481. LNCS. 2019, pp. 256–269.

[26] Pedro Cabalar, Roland Kaminski, Torsten Schaub, and Anna Schuhmann. “Temporal Answer Set Programming on Finite Traces”. In: *Theory Pract. Log. Program.* 18.3-4 (2018), pp. 406–420.

[27] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. “ASP-Core-2 Input Language Format”. In: *Theory Pract. Log. Program.* 20.2 (2020), pp. 294–309.

[28] Diego Calvanese, Giuseppe De Giacomo, and Moshe Y. Vardi. “Reasoning about Actions and Planning in LTL Action Theories”. In: *KR*. 2002, pp. 593–602.

[29] Alberto Camacho and Sheila A. McIlraith. “Learning Interpretable Models Expressed in Linear Temporal Logic”. In: *ICAPS*. AAAI Press, 2019, pp. 621–630.

[30] Alessio Cecconi, Claudio Di Ciccio, Giuseppe De Giacomo, and Jan Mendling. “Interestingness of Traces in Declarative Process Mining: The Janus LTLp _f Approach”. In: *BPM*. Vol. 11080. Lecture Notes in Computer Science. Springer, 2018, pp. 121–138.

[31] Alessio Cecconi, Giuseppe De Giacomo, Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. “Measuring the interestingness of temporal logic behavioral specifications in process mining”. In: *Inf. Syst.* 107 (2022), p. 101920.

[32] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in Computer Science. Springer, 1990.

[33] Stefano Ceri, Georg Gottlob, and Letizia Tanca. “What you Always Wanted to Know About Datalog (And Never Dared to Ask)”. In: *IEEE Trans. Knowl. Data Eng.* 1.1 (1989), pp. 146–166.

[34] William Chan. “Temporal-logic Queries”. In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 450–463. ISBN: 978-3-540-45047-4.

[35] Federico Chesani, Chiara Di Francescomarino, Chiara Ghidini, Giulia Grundler, Daniela Loreti, Fabrizio Maria Maggi, Paola Mello, Marco Montali, and Sergio Tessaris. “Optimising Business Process Discovery Using Answer Set Programming”. In: *LPNMR*. Vol. 13416. Lecture Notes in Computer Science. Springer, 2022, pp. 498–504.

[36] Federico Chesani, Chiara Di Francescomarino, Chiara Ghidini, Daniela Loreti, Fabrizio Maria Maggi, Paola Mello, Marco Montali, and Sergio Tessaris. “Process Discovery on Deviant Traces and Other Stranger Things”. In: *IEEE Trans. Knowl. Data Eng.* 35.11 (2023), pp. 11784–11800.

[37] Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. “Exploiting Inductive Logic Programming Techniques for Declarative Process Mining”. In: *Trans. Petri Nets Other Model. Concurr.* 2 (2009), pp. 278–295.

[38] Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. “Exploiting Inductive Logic Programming Techniques for Declarative Process Mining”. In: *Trans. Petri Nets Other Model. Concurr.* 2 (2009), pp. 278–295.

[39] Francesco Chiariello, Valeria Fionda, Antonio Ielo, and Francesco Ricca. “A Direct ASP Encoding for Declare”. In: *PADL*. Vol. 14512. Lecture Notes in Computer Science. Springer, 2024, pp. 116–133.

[40] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. “ASP-Based Declarative Process Mining”. In: *AAAI*. AAAI Press, 2022, pp. 5539–5547.

[41] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. “From LTL on Process Traces to Finite-state Automata”. In: *BPM*. Vol. 3469. CEUR WP. CEUR-WS.org, 2023, pp. 127–131.

[42] Michele Chinosi and Alberto Trombetta. “BPMN: An introduction to the standard”. In: *Comput. Stand. Interfaces* 34.1 (2012), pp. 124–134.

[43] Claudio Di Ciccio and Marco Montali. “Declarative Process Specifications: Reasoning, Discovery, Monitoring”. In: *Process Mining Handbook*. Vol. 448. Lecture Notes in Business Information Processing. Springer, 2022, pp. 108–152.

[44] Claudio Di Ciccio, Mitchel H. M. Schouten, Massimiliano de Leoni, and Jan Mendling. “Declarative Process Discovery with MINERful in ProM”. In: *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015)*. Ed. by Florian Daniel and Stefan Zugal. Vol. 1418. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 60–64.

[45] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. “Bounded Model Checking Using Satisfiability Solving”. In: *Formal Methods Syst. Des.* 19.1 (2001), pp. 7–34.

[46] Carl Corea, John Grant, and Matthias Thimm. “Measuring Inconsistency in Declarative Process Specifications”. In: *BPM*. Vol. 13420. Lecture Notes in Computer Science. Springer, 2022, pp. 289–306.

[47] Carl Corea, Isabelle Kuhlmann, Matthias Thimm, and John Grant. “Paraconsistent reasoning for inconsistency measurement in declarative process specifications”. In: *Inf. Syst.* 122 (2024), p. 102347.

[48] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. “Inductive logic programming at 30”. In: *Mach. Learn.* 111.1 (2022), pp. 147–172.

[49] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. “Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness”. In: *AAAI*. AAAI Press, 2014, pp. 1027–1033.

[50] Giuseppe De Giacomo, Marlon Dumas, Fabrizio Maria Maggi, and Marco Montali. “Declarative Process Modeling in BPMN”. In: *CAiSE*. Vol. 9097. Lecture Notes in Computer Science. Springer, 2015, pp. 84–100.

[51] Giuseppe De Giacomo and Marco Favorito. “Compositional Approach to Translate LTLf/LDLf into Deterministic Finite Automata”. In: *ICAPS*. AAAI Press, 2021, pp. 122–130.

[52] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. “Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications”. In: *ICAPS*. AAAI Press, 2019, pp. 128–136.

[53] Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Sebastian Sardiña. “Computing Trace Alignment against Declarative Process Models through Planning”. In: *ICAPS*. 2016, pp. 367–375.

[54] Giuseppe De Giacomo and Moshe Y. Vardi. “Automata-Theoretic Approach to Planning for Temporally Extended Goals”. In: *ECP*. Vol. 1809. LNCS. 1999, pp. 226–238.

[55] Giuseppe De Giacomo and Moshe Y. Vardi. “Linear Temporal Logic and Linear Dynamic Logic on Finite Traces”. In: *IJCAI*. IJCAI/AAAI, 2013, pp. 854–860.

[56] Stéphane Demri, Valentin Goranko, and Martin Lange. *Temporal Logics in Computer Science: Finite-State Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016.

[57] Claudio Di Ciccio and Marco Montali. “Declarative Process Specifications: Reasoning, Discovery, Monitoring”. In: *PM Handbook*. Vol. 448. LNIP. Springer, 2022, pp. 108–152.

[58] Carmine Dodaro, Valeria Fionda, and Gianluigi Greco. “LTL on Weighted Finite Traces: Formal Foundations and Algorithms”. In: *IJCAI*. ijcai.org, 2022, pp. 2606–2612.

[59] Carmine Dodaro and Francesco Ricca. “The External Interface for Extending WASP”. In: *Theory Pract. Log. Program.* 20.2 (2020), pp. 225–248.

[60] Ivan Donadello, Francesco Riva, Fabrizio Maria Maggi, and Aladdin Shikhizada. “Declare4Py: A Python Library for Declarative Process Mining”. In: *BPM (PhD/Demos)*. Vol. 3216. CEUR WP. CEUR-WS.org, 2022, pp. 117–121.

[61] Matthew B Dwyer, George S Avrunin, and James C Corbett. “Patterns in property specifications for finite-state verification”. In: *Proceedings of the 21st international conference on Software engineering*. 1999, pp. 411–420.

[62] Thomas Eiter and Georg Gottlob. “On the Computational Cost of Disjunctive Logic Programming: Propositional Case”. In: *Ann. Math. Artif. Intell.* 15.3-4 (1995), pp. 289–323.

[63] Thomas Eiter, Georg Gottlob, and Nicola Leone. “Abduction from Logic Programs: Semantics and Complexity”. In: *Theor. Comput. Sci.* 189.1-2 (1997), pp. 129–177.

[64] Thomas Eiter, Nicola Leone, and Domenico Saccà. “Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases”. In: *Theor. Comput. Sci.* 206.1-2 (1998), pp. 181–218.

[65] Esra Erdem, Michael Gelfond, and Nicola Leone. “Applications of Answer Set Programming”. In: *AI Mag.* 37.3 (2016), pp. 53–68.

[66] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. “Industrial Applications of Answer Set Programming”. In: *Künstliche Intell.* 32.2-3 (2018), pp. 165–176.

[67] Nathanaël Fijalkow and Guillaume Lagarde. “The Complexity of Learning Linear Temporal Formulas from Examples”. In: *International Conference on Grammatical Inference*. PMLR. 2021, pp. 237–250.

[68] Valeria Fionda and Gianluigi Greco. “LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner”. In: *J. Artif. Intell. Res.* 63 (2018), pp. 557–623.

[69] Valeria Fionda and Antonella Guzzo. “Control-Flow Modeling with Declare: Behavioral Properties, Computational Complexity, and Tools”. In: *IEEE TKDE* 32.5 (2020), pp. 898–911.

[70] Valeria Fionda, Antonio Ielo, and Francesco Ricca. “LTLf2ASP: LTLf Bounded Satisfiability in ASP”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Carmine Dodaro, Gopal Gupta, and Maria Vanina Martinez. Cham: Springer Nature Switzerland, 2025, pp. 373–386. ISBN: 978-3-031-74209-5.

[71] Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo. “Induction and Exploitation of Subgoal Automata for Reinforcement Learning”. In: *J. Artif. Intell. Res.* 70 (2021), pp. 1031–1116.

[72] Mark Gabel and Zhendong Su. “Symbolic mining of temporal specifications”. In: *ICSE*. 2008, pp. 51–60.

[73] Jean-Raphaël Gaglione, Daniel Neider, Rajarshi Roy, Ufuk Topcu, and Zhe Xu. “MaxSAT-based temporal logic inference from noisy data”. In: *Innov. Syst. Softw. Eng.* 18.3 (2022), pp. 427–442.

[74] Alfredo Garro, Luigi Palopoli, and Francesco Ricca. “Exploiting agents in e-learning and skills management context”. In: *AI Commun.* 19.2 (2006), pp. 137–154.

[75] Luca Geatti, Alessandro Gianola, and Nicola Gigante. “Linear Temporal Logic Modulo Theories over Finite Traces”. In: *IJCAI*. ijcai.org, 2022, pp. 2641–2647.

[76] Luca Geatti, Alessandro Gianola, Nicola Gigante, and Sarah Winkler. “Decidable Fragments of $\mathrm{LTL_f}$ Modulo Theories”. In: *ECAI*. Vol. 372. Frontiers in Artificial Intelligence and Applications. IOS Press, 2023, pp. 811–818.

[77] Luca Geatti, Nicola Gigante, and Angelo Montanari. “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, Proceedings*. Vol. 11714. Lecture Notes in Computer Science. Springer, 2019, pp. 3–20.

[78] Luca Geatti, Nicola Gigante, Angelo Montanari, and Gabriele Venturato. “SAT Meets Tableaux for Linear Temporal Logic Satisfiability”. In: *J. Autom. Reason.* 68.2 (2024), p. 6.

[79] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[80] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. “Multi-shot ASP solving with clingo”. In: *Theory Pract. Log. Program.* 19.1 (2019), pp. 27–82.

[81] Michael Gelfond and Vladimir Lifschitz. “Classical Negation in Logic Programs and Disjunctive Databases”. In: *New Gener. Comput.* 9.3/4 (1991), pp. 365–386.

[82] Michael Gelfond and Vladimir Lifschitz. “The Stable Model Semantics for Logic Programming”. In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium*. Ed. by Robert A. Kowalski and Kenneth A. Bowen. MIT Press, 1988, pp. 1070–1080.

[83] Enrico Ghiorzi, Michele Colledanchise, Gianluca Piquet, Stefano Bernagozzi, Armando Tacchella, and Lorenzo Natale. “Learning Linear Temporal Properties for Autonomous Robotic Systems”. In: *IEEE Robotics Autom. Lett.* 8.5 (2023), pp. 2930–2937.

[84] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. “Restraining Bolts for Reinforcement Learning Agents”. In: *AAAI*. AAAI Press, 2020, pp. 13659–13662.

[85] Giuseppe De Giacomo and Sasha Rubin. “Automata-Theoretic Foundations of FOND Planning for LTLf and LDLf Goals”. In: *IJCAI*. ijcai.org, 2018, pp. 4729–4735.

[86] Alessandro Gianola, Marco Montali, and Sarah Winkler. “Linear-Time Verification of Data-Aware Processes Modulo Theories via Covers and Automata”. In: *AAAI*. AAAI Press, 2024, pp. 10525–10534.

[87] Laura Giordano, Alberto Martelli, Matteo Spiotta, and Daniele Theseider Dupré. “Business process verification with constraint temporal answer set programming”. In: *Theory Pract. Log. Program.* 13.4-5 (2013), pp. 641–655.

[88] Giovanni Grasso, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. “Some DLV Applications for Knowledge Management”. In: *LPNMR*. Vol. 5753. Lecture Notes in Computer Science. Springer, 2009, pp. 591–597.

[89] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. “Little Tricky Logic: Misconceptions in the Understanding of LTL”. In: *Art Sci. Eng. Program.* 7.2 (2023).

[90] Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, eds. *Handbook of Knowledge Representation*. Vol. 3. Foundations of Artificial Intelligence. Elsevier, 2008. ISBN: 978-0-444-52211-5.

[91] Antonio Ielo, Mark Law, Valeria Fionda, Francesco Ricca, Giuseppe De Giacomo, and Alessandra Russo. "Towards ILP-Based LTL f Passive Learning". In: *Inductive Logic Programming - 32nd International Conference, ILP 2023, Proceedings*. Ed. by Elena Bellodi, Francesca Alessandra Lisi, and Riccardo Zese. Vol. 14363. Lecture Notes in Computer Science. Springer, 2023, pp. 30–45.

[92] Antonio Ielo, Francesco Ricca, and Luigi Pontieri. "Declarative Mining of Business Processes via ASP". In: *PMAI@IJCAI*. Vol. 3310. CEUR WP. CEUR-WS.org, 2022, pp. 105–108.

[93] ILASP. *ILASP Manual*. Accessed: 2024-10-07. 2023. URL: `https://doc.ilasp.com/`.

[94] Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. "How to Build Your Own ASP-based System?!" In: *Theory Pract. Log. Program.* 23.1 (2023), pp. 299–361.

[95] Robert Kolter. "Inductive temporal logic programming". PhD thesis. University of Kaiserslautern, 2009.

[96] Ron Koymans. "Specifying Real-Time Properties with Metric Temporal Logic". In: *Real Time Syst.* 2.4 (1990), pp. 255–299.

[97] Isabelle Kuhlmann and Carl Corea. "Inconsistency Measurement in LTL $_{\mathrm{f}}$ Based on Minimal Inconsistent Sets and Minimal Correction Sets". In: *SUM*. Vol. 15350. Lecture Notes in Computer Science. Springer, 2024, pp. 217–232.

[98] Isabelle Kuhlmann, Carl Corea, and John Grant. "An ASP-Based Framework for Solving Problems Related to Declarative Process Specifications". In: *NMR*. Vol. 3464. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 129–132.

[99] Isabelle Kuhlmann, Carl Corea, and John Grant. "Non-Automata Based Conformance Checking of Declarative Process Specifications Based on ASP". In: *Business Process Management Workshops*. Vol. 492. Lecture Notes in Business Information Processing. Springer, 2023, pp. 396–408.

[100] Evelina Lamma, Paola Mello, Fabrizio Riguzzi, and Sergio Storari. “Applying Inductive Logic Programming to Process Mining”. In: *Inductive Logic Programming, 17th International Conference, ILP 2007, Corvallis, OR, USA, June 19-21, 2007, Revised Selected Papers*. Ed. by Hendrik Blockeel, Jan Ramon, Jude W. Shavlik, and Prasad Tadepalli. Vol. 4894. Lecture Notes in Computer Science. Springer, 2007, pp. 132–146.

[101] Leslie Lamport. “The Temporal Logic of Actions”. In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923.

[102] Mark Law. “Conflict-Driven Inductive Logic Programming”. In: *Theory Pract. Log. Program.* 23.2 (2023), pp. 387–414.

[103] Mark Law, Alessandra Russo, and Krysia Broda. “Logic-Based Learning of Answer Set Programs”. In: *Reasoning Web.* 2019, pp. 196–231.

[104] Mark Law, Alessandra Russo, and Krysia Broda. “The complexity and generality of learning answer set programs”. In: *Artif. Intell.* 259 (2018), pp. 110–146.

[105] Mark Law, Alessandra Russo, and Krysia Broda. *The ILASP system for learning Answer Set Programs*. `www.ilasp.com`. 2015.

[106] Massimiliano de Leoni and Wil M. P. van der Aalst. “Data-aware process mining: discovering decisions in processes using alignments”. In: *SAC*. ACM, 2013, pp. 1454–1461.

[107] Jianwen Li, Kristin Y. Rozier, Geguang Pu, Yueling Zhang, and Moshe Y. Vardi. “SAT-Based Explicit LTLf Satisfiability Checking”. In: *AAAI*. AAAI Press, 2019, pp. 2946–2953.

[108] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. “LTLf Satisfiability Checking”. In: *ECAI*. Vol. 263. Frontiers in Artificial Intelligence and Applications. IOS Press, 2014, pp. 513–518.

[109] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. “Fast, flexible MUS enumeration”. In: *Constraints An Int. J.* 21.2 (2016), pp. 223–250.

[110] Mark H. Liffiton and Karem A. Sakallah. “Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints”. In: *J. Autom. Reason.* 40.1 (2008), pp. 1–33.

[111] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[112] Iezalde F. Lopes and Diogo R. Ferreira. “A Survey of Process Mining Competitions: The BPI Challenges 2011-2018”. In: *Business Process Management Workshops*. Vol. 362. Lecture Notes in Business Information Processing. Springer, 2019, pp. 263–274.

[113] Inês Lynce and João Marques-Silva. “On Computing Minimum Unsatisfiable Cores”. In: *SAT*. 2004.

[114] Fabrizio Maria Maggi, Marco Montali, and Rafael Peñaloza. “Temporal Logics Over Finite Traces with Uncertainty”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*. AAAI Press, 2020, pp. 10218–10225.

[115] Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. “Runtime Verification of LTL-Based Declarative Process Models”. In: *RV*. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 131–146.

[116] Marco Manna, Francesco Ricca, and Giorgio Terracina. “Consistent query answering via ASP from different perspectives: Theory and practice”. In: *Theory Pract. Log. Program.* 13.2 (2013), pp. 227–252.

[117] João Marques-Silva. “Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper)”. In: *2010 40th IEEE International Symposium on Multiple-Valued Logic*. IEEE Computer Society, 2010, pp. 9–14.

[118] João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. “On Computing Minimal Correction Subsets”. In: *IJCAI*. IJCAI/AAAI, 2013, pp. 615–622.

[119] João Marques-Silva, Mikolás Janota, and Carlos Mencía. “Minimal sets on propositional formulae. Problems and reductions”. In: *Artif. Intell.* 252 (2017), pp. 22–50.

[120] Riccardo De Masellis, Fabrizio Maria Maggi, and Marco Montali. “Monitoring data-aware business constraints with finite state automata”. In: *ICSSP*. ACM, 2014, pp. 134–143.

[121] Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. “Compilation of aggregates in ASP systems”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 5. 2022, pp. 5834–5841.

[122] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. “Causality and Explanations in Databases”. In: *Proc. VLDB Endow.* 7.13 (2014), pp. 1715–1716.

[123] Carlos Mencía and João 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.

[124] Carlos Mencía and João Marques-Silva. “Reasoning About Strong Inconsistency in ASP”. In: *SAT*. Vol. 12178. Lecture Notes in Computer Science. Springer, 2020, pp. 332–342.

[125] Tim Miller. “Explanation in artificial intelligence: Insights from the social sciences”. In: *Artif. Intell.* 267 (2019), pp. 1–38.

[126] Marco Montali. *Specification and Verification of Declarative Open Interaction Models - A Logic-Based Approach*. Vol. 56. Lecture Notes in Business Information Processing. Springer, 2010.

[127] Marco Montali, Federico Chesani, Paola Mello, and Fabrizio Maria Maggi. “Towards data-aware constraints in declare”. In: *SAC*. ACM, 2013, pp. 1391–1396.

[128] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. “Iterative and core-guided MaxSAT solving: A survey and assessment”. In: *Constraints An Int. J.* 18.4 (2013), pp. 478–534.

[129] Artur Mrowca, Martin Nocker, Sebastian Steinhorst, and Stephan Günnemann. “Learning Temporal Specifications from Imperfect Traces Using Bayesian Inference”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, pp. 1–6.

[130] Daniel Neider and Ivan Gavran. “Learning Linear Temporal Properties”. In: *FMCAD*. 2018, pp. 1–10.

[131] Ilkka Niemelä. “Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm”. In: *Ann. Math. Artif. Intell.* 25.3-4 (1999), pp. 241–273.

[132] Tong Niu, Shengping Xiao, Xiaoyu Zhang, Jianwen Li, Yanhong Huang, and Jianqi Shi. “Computing minimal unsatisfiable core for LTL over finite traces”. In: *Journal of Logic and Computation* (Aug. 2023). ISSN: 0955-792X.

[133] Rafael Peñaloza. “Axiom Pinpointing”. In: *Applications and Practices in Ontology Design, Extraction, and Reasoning*. Ed. by Giuseppe Cota, Marilena Daquino, and Gian Luca Pozzato. Vol. 49. Studies on the Semantic Web. IOS Press, 2020, pp. 162–177.

[134] Rafael Peñaloza and Francesco Ricca. “Pinpointing Axioms in Ontologies via ASP”. In: *Proceedings of LPNMR 2022*. Ed. by Georg Gottlob, Daniela Inclezan, and Marco Maratea. Vol. 13416. Lecture Notes in Computer Science. Springer, 2022, pp. 315–321.

[135] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. “DECLARE: Full Support for Loosely-Structured Processes”. In: *Proceedings of EDOC 2007*. IEEE Computer Society, 2007, pp. 287–300.

[136] Nir Piterman and Amir Pnueli. “Temporal Logic and Fair Discrete Systems”. In: *Handbook of Model Checking*. Springer, 2018, pp. 27–73.

[137] Amir Pnueli. “Linear and Branching Structures in the Semantics and Logics of Reactive Systems”. In: *Automata, Languages and Programming, 12th Colloquium, Nafplion, Greece, July 15-19, 1985, Proceedings*. Ed. by Wilfried Brauer. Vol. 194. Lecture Notes in Computer Science. Springer, 1985, pp. 15–32.

[138] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[139] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. “Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic”. In: *TACAS*. Vol. 13243. Lecture Notes in Computer Science. 2022, pp. 263–280.

[140] Margus Räim, Claudio Di Ciccio, Fabrizio Maria Maggi, Massimo Mecella, and Jan Mendling. “Log-Based Understanding of Business Processes through Temporal Logic Query Checking”. In: *OTM Conferences*. Vol. 8841. Lecture Notes in Computer Science. Springer, 2014, pp. 75–92.

[141] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. “cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis”. In: *CAV*. 2019, pp. 74–83.

[142] Mark Reynolds. “A New Rule for LTL Tableaux”. In: *GandALF*. Vol. 226. EPTCS. 2016, pp. 287–301.

[143] Tony Ribeiro, Maxime Folschette, Morgan Magnin, Kotaro Okazaki, Lo Kuo-Yen, and Katsumi Inoue. “Diagnosis of Event Sequences with LFIT”. In: *The 31st International Conference on Inductive Logic Programming (ILP)*. 2022.

[144] Marco Roveri, Claudio Di Ciccio, Chiara Di Francescomarino, and Chiara Ghidini. “Computing Unsatisfiable Cores for LTLf Specifications”. In: *J. Artif. Intell. Res.* 80 (2024), pp. 517–558.

[145] Stefan Schlobach and Ronald Cornet. “Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies”. In: *Proceedings of IJCAI’03*. Ed. by Georg Gottlob and Toby Walsh. Morgan Kaufmann, 2003, pp. 355–362.

[146] Viktor Schuppan and Luthfi Darmawan. “Evaluating LTL Satisfiability Solvers”. In: *ATVA*. Vol. 6996. Lecture Notes in Computer Science. Springer, 2011, pp. 397–413.

[147] Roberto Sebastiani and Michele Vescovi. “Axiom Pinpointing in Lightweight Description Logics via Horn-SAT Encoding and Conflict Analysis”. In: *Procedding of the 22nd International Conference on Automated Deduction*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 84–99.

[148] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. “Debugging Overconstrained Declarative Models Using Unsatisfiable Cores”. In: *ASE*. IEEE Computer Society, 2003, pp. 94–105.

[149] Tran Cao Son, Chitta Baral, Tran Hoai Nam, and Sheila A. McIlraith. “Domain-dependent knowledge in answer set planning”. In: *ACM Trans. Comput. Log.* 7.4 (2006), pp. 613–657.

[150] Ole Tange. “GNU Parallel: The Command-Line Power Tool”. In: *login Usenix Mag.* 36.1 (2011).

[151] Yih-Kuen Tsay and Moshe Y. Vardi. “From Linear Temporal Logics to Büchi Automata: The Early and Simple Principle”. In: *Model Checking, Synthesis, and Learning*. Vol. 13030. LNCS. 2021, pp. 8–40.

[152] Lluís Vila. “A Survey on Temporal Reasoning in Artificial Intelligence”. In: *AI Commun.* 7.1 (1994), pp. 4–28.

[153] Przemyslaw Andrzej Walega, David J. Tena Cucala, Bernardo Cuenca Grau, and Egor V. Kostylev. “The Stable Model Semantics of Datalog with Metric Temporal Operators”. In: *Theory Pract. Log. Program.* 24.1 (2024), pp. 22–56.

[154] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. “DatalogMTL: Computational Complexity and Expressive Power”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 1886–1892. DOI: 10.24963/IJCAI.2019/261. URL: https://doi.org/10.24963/ijcai.2019/261.

[155] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, Third Edition*. Springer, 2019.