SAPIENZA
UNIVERSITÀ DI ROMA

LTL and Past LTL on Finite Traces for Planning and Declarative Process Mining

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea Magistrale in Engineering in Computer Science

Candidate
Francesco Fuggitti
ID number 1735212

Thesis Advisor
Prof. Giuseppe De Giacomo

Academic Year 2017/2018

Thesis not yet defended

LTL and Past LTL on Finite Traces for Planning and Declarative Process Mining
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: fuggitti.1735212@studenti.uniroma1.it

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem . . . 1
- 1.3 Objectives . . . 1
- 1.4 Results . . . 1
- 1.5 Structure . . . 1

2 PLTL and LTL$_f$ **3**
- 2.1 Linear Temporal Logic (LTL) . . . 3
  - 2.1.1 Syntax . . . 4
  - 2.1.2 Semantics . . . 4
  - 2.1.3 Results . . . 5
- 2.2 Linear Temporal Logic on Finite Traces (LTL$_f$) . . . 5
  - 2.2.1 Syntax . . . 5
  - 2.2.2 Semantics . . . 6
  - 2.2.3 Results . . . 7
- 2.3 Past Linear Temporal Logic (PLTL) . . . 7
  - 2.3.1 Syntax . . . 7
  - 2.3.2 Semantics . . . 8
  - 2.3.3 Results . . . 9
  - 2.3.4 Expressiveness . . . 9
- 2.4 LTL$_f$ and PLTL Translation to Automata . . . 9
  - 2.4.1 ∂ function for LTL$_f$ . . . 10
  - 2.4.2 ∂ function for PLTL . . . 10
- 2.5 LTL$_f$/PLTL to FOL Encoding and MONA . . . 10
  - 2.5.1 LTL$_f$-to-FOL Encoding . . . 10
  - 2.5.2 PLTL-to-FOL Encoding . . . 12
  - 2.5.3 MONA and FOL-to-MONA Encoding . . . 13
- 2.6 Summary . . . 17

3 LTL$_f$2DFA **19**
- 3.1 Introduction . . . 19
- 3.2 Package Structure . . . 20

3.2.1 Lexer.py . . . 20
3.2.2 Parser.py . . . 22
3.2.3 Translator.py . . . 25
3.2.4 DotHandler.py . . . 33
3.3 Interpreting LTL$_f$2DFA output . . . 35
3.4 Comparison with FLLOAT . . . 37
3.5 Discussion . . . 38

4 Planning for Extended Temporal Goals 39
4.1 Idea and Motivations . . . 39
4.2 Preliminaries . . . 40
4.2.1 PDDL . . . 40
4.2.2 Fully Observable Non Deterministic Planning . . . 40
4.3 Encoding of Temporal Goals in PDDL . . . 40
4.4 Implementation . . . 40
4.4.1 Package Structure . . . 40
4.4.2 PDDL . . . 40
4.4.3 Automa . . . 40
4.4.4 Main Module . . . 40
4.5 Summary . . . 40

5 Janus 41
5.1 Declarative Process Mining . . . 41
5.2 The Janus Approach . . . 44
5.2.1 Algorithm . . . 48
5.3 Implementation . . . 50
5.3.1 Package Structure . . . 51
5.3.2 I/O . . . 51
5.3.3 Automata . . . 53
5.3.4 Formulas . . . 60
5.3.5 Main Module . . . 61
5.3.6 Results . . . 61
5.4 Summary . . . 61

6 Conclusions and Future Work 63
6.1 Overview . . . 63
6.2 Main Contributions . . . 63
6.3 Future Works . . . 63
6.4 Final Remarks . . . 63

Bibliography 65

Chapter 1

Introduction

Here the intro of the intro

1.1 Context

here the context of the thesis

1.2 Problem

what is the problem solved

1.3 Objectives

what are the objective of the thesis

1.4 Results

what are the results achieved

1.5 Structure

what is the structure of the thesis

Chapter 2

PLTL and LTL_f

This chapter will deal with the theoretical framework on which all topics present in the thesis are based. Initially, we will introduce the widely known Linear-Time Temporal Logic (LTL) and the Past Linear Time Temporal Logic (PLTL), focusing on their syntax and semantic. Secondly, we will talk about the concept of *Finite Trace* in these formal languages and how it changes them. Specifically, we will describe the Linear Time Temporal Logic over Finite Traces (LTL_f). Then, we will illustrate the theory behind the transformation of an LTL_f or PLTL formula to a Deterministic Finite State Automaton (DFA). Finally, we will describe the translation of an LTL_f or PLTL formula to the classic First-Order Logic formalism (FOL) and the translation of a FOL formula into a program that the MONA, a tool that translates formulas into a DFA, can manage. Some examples will be provided, but we will suppose the reader to be confident with classical logic and automata theory.

2.1 Linear Temporal Logic (LTL)

Temporal Logic formalisms are a set of formal languages designed for representing temporal information and reasoning about time within a logical framework (Goranko and Galton, 2015). Indeed, these logics are used when propositions have their truth value dependent on time.

In this scenario, we find the *Linear Temporal Logic* (LTL) which is a a very well known modal temporal logic with modalities referring to time. It was originally proposed in (Pnueli, 1977) as a specification language for concurrent programs. Consequently, LTL has been extensively used in Artificial Intelligence and Computer Science. For instance, it has been employed in planning, reasoning about actions, declarative process mining and verification of software/hardware systems.

2.1.1 Syntax

Given a set of propositional symbols $\mathcal{P}$, a valid LTL formula φ is defined as follows:

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where $a \in \mathcal{P}$. The unary operator $\bigcirc$(*next-time*) and the binary operator $\mathcal{U}$ (*until*) are temporal operators and we use $\top$ and $\bot$ to denote *true* and *false* respectively. Moreover, all classical logic operators $\vee, \Rightarrow, \Leftrightarrow, true$ and $false$ can be used. Intuitively, $\bigcirc\varphi$ says that φ is true at the *next* instant, $\varphi_1 \mathcal{U} \varphi_2$ says that at some future instant, φ_2 will hold and *until* that point φ_1 holds. We also define common abbreviations for some specific temporal formulas: *eventually* as $\Diamond\varphi \doteq true\,\mathcal{U}\,\varphi$, *always* as $\Box\varphi \doteq \neg\Diamond\neg\varphi$, *weak-next* as $\bullet\varphi \doteq \neg\bigcirc\neg\varphi$ and *release* as $\varphi_1 \mathcal{R} \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$.

LTL allows to express a lot of interesting properties defined over time. In the Example 2.1 we show some of them.

Example 2.1. Interesting LTL patterns:

- *Safety*: $\Box\varphi$, which means *it is always true that property in* φ *will happen* or φ *will hold forever*. For instance, $\Box\neg(reactorTemp > 1000)$ (the temperature of the reactor must never exceed 1000).

- *Liveness*: $\Diamond\varphi$, which means *sooner or later* φ *will hold* or *something good will eventually happen*. For instance, $\Diamond rich$ (eventually I will become rich).

- *Response*: $\Box\Diamond\varphi$ which means *for every point in time, there is a point later where* φ *holds*.

- *Persistence*: $\Diamond\Box\varphi$, which means *there exists a point in the future such that from then on* φ *always holds*.

- *Strong fairness*: $\Box\Diamond\varphi_1 \Rightarrow \Box\Diamond\varphi_2$, *if something is attempted/requested infinitely often, then it will be successful/allocated infinitely often*. For instance, $\Box\Diamond ready \Rightarrow \Box\Diamond run$ (if a process is in ready state infinitely often, then it will be selected by the scheduler infinitely often).

2.1.2 Semantics

The semantics of the main operators of LTL over *infinite traces* are expressed as an ω-word over the alphabet $2^{\mathcal{P}}$. We give the following definitions:

Definition 2.1. Given an infinite trace π, we inductively define when an LTL formula φ is *true* at an instant i, in symbols $\pi, i \models \varphi$, as follows:

$$\pi, i \models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$

$\pi, i \models \circ\varphi$ iff $\pi, i+1 \models \varphi$

$\pi, i \models \varphi_1 \mathcal{U} \varphi_2$ iff $\exists j.(j \geq i) \wedge \pi, j \models \varphi_2 \wedge \forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1$

Definition 2.2. An LTL formula φ is *true* in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is *satisfiable* if it is true in some π and is *valid* if it is true in every π. A formula φ_1 *logically implies* another formula φ_2, in symbols $\varphi_1 \models \varphi_2$ iff $\forall \pi, \pi \models \varphi_1 \Rightarrow \pi \models \varphi_2$.

Notice that satisfiability, validity and logical implication are all mutually reducible one to each other.

Example 2.2. Validity and logical implication as satisfiability

- φ is valid iff $\neg\varphi$ is unsatisfiable.
- $\varphi_1 \models \varphi_2$ iff $\varphi_1 \wedge \neg\varphi_2$ is unsatisfiable.

2.1.3 Results

About LTL complexity, we can state the following fundamental theorem:

Theorem 2.1. *(Sistla and Clarke, 1985) Satisfiability, validity, and logical implication for* LTL *formulas are* PSPACE*-complete.*

2.2 Linear Temporal Logic on Finite Traces (LTL$_f$)

Linear Temporal Logic on Finite Traces (LTL$_f$) is the variant of LTL described in Section 2.1 interpreted over *finite traces* (De Giacomo and Vardi, 2013). Although it seems a little difference, in some cases, the interpretation of a formula over finite traces completely changes its meaning with respect to the one over inifinite traces.

2.2.1 Syntax

The syntax of LTL$_f$ is exactly the same of LTL. Indeed, LTL$_f$ formulas are built from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\circ$(*next-time*) and the binary operator $\mathcal{U}$ (*until*). Formulas can be defined as follows:

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \circ\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where $a \in \mathcal{P}$. All usual logical operators such as $\vee, \Rightarrow, \Leftrightarrow, true$ and $false$ are also used. Similarly to LTL, we can define the following common abbreviations for temporal operators:

$$\Diamond\varphi \doteq true\,\mathcal{U}\,\varphi \tag{2.1}$$

$$\Box\varphi \doteq \neg\Diamond\neg\varphi \tag{2.2}$$

$$\bullet\varphi \doteq \neg\bigcirc\neg\varphi \tag{2.3}$$

$$\varphi_1 \mathcal{R}\, \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2) \tag{2.4}$$

$$\mathit{Last} \doteq \bullet\mathit{false} \tag{2.5}$$

$$\mathit{End} \doteq \Box\mathit{false} \tag{2.6}$$

Compared with LTL, in LTL$_f$ there have been defined also 2.5 and 2.6 which denotes the last instance of the trace and that the trace is ended, respectively. As we have seen in Example 2.1 with LTL, now we will see in Example 2.3 how properties expressed in LTL$_f$ have changed their meaning with the interpretation over finite traces.

Example 2.3. Interesting LTL$_f$ patterns:

- *Safety*: $\Box\varphi$, which now means always *till the end of the trace* φ holds.
- *Liveness*: $\Diamond\varphi$, which now means eventually *before the end of the trace* φ holds.
- *Responce*: $\Box\Diamond\varphi$, which means for any point in the trace there exist a point later in the trace where φ holds. This property, interpreted over finite traces, can be seen also as $\Diamond(\mathit{Last} \wedge \varphi)$ because $\Box\Diamond\varphi$ implies that the *last point in the trace satisfies* φ.
- *Persistance*: $\Diamond\Box\varphi$ means that there is a point in the trace such that from then on until the end of the trace φ holds. Also here the meaning can me seen as $\Diamond(\mathit{Last} \wedge \varphi)$ since $\Diamond\Box\varphi$ implies that at the last point of the trace $\Box\varphi$, and so φ, holds.

In other words, no direct nesting of *eventually* and *always* connectives is meaningful in LTL$_f$. However, indirect nesting of *eventually* and *always* connectives can still produce meaningful and interesting properties. One example could be $\Box(\psi \Rightarrow \Diamond\varphi)$, which stands for *always, before the end of the trace, if ψ holds then φ will eventually hold.*

2.2.2 Semantics

The semantics of LTL$_f$ is given as LT$_f$-interpretations, namely interpretations over a *finite traces* denoting a finite sequence of consecutive instants of time. Formally, LT$_f$-interpretations are expressed as finite words π over the alphabet $2^{\mathcal{P}}$, i.e. as alphabet we have all the possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We use the following notation. We denote the *length* of a trace π as $\mathit{length}(\pi)$. We denote the *positions*, i.e. instants, on the trace as $\pi(i)$ with $0 \leq i \leq \mathit{last}$ where $\mathit{last} = \mathit{length}(\pi) - 1$ is the last element of the trace. We denote by $\pi(i,j)$, the *segment* (i.e., the subword) of π, the trace $\pi' = \langle \pi(i), \pi(i+1), \ldots, \pi(j) \rangle$, with $0 \leq i \leq j \leq \mathit{last}$. We now give the following definitions:

Definition 2.3. Given an LT_f-interpretation π, we define when an LTL_f formula φ is *true* at position i (for $0 \leq i \leq last$), in symbols $\pi, i \models \varphi$, inductively as follows:

$$\pi, i \models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$

$$\pi, i \models \bigcirc\varphi \text{ iff } i < last \wedge \pi, i+1 \models \varphi \tag{2.7}$$

$$\pi, i \models \varphi_1 \,\mathcal{U}\, \varphi_2 \text{ iff } \exists j.(i \leq j \leq last) \wedge \pi, j \models \varphi_2 \wedge \forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1 \tag{2.8}$$

The Definition 2.3 is exactly the same Definiton 2.1 seen for LTL except for 2.7 and 2.8 in which the only difference lies on the intervals bounded by the last element of the trace.

Definition 2.4. An LTL_f formula is *true* in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is *satisfiable* if it is true in some LT_f-interpretation, and is *valid* if it is true in every LT_f-interpretation. A formula φ_1 *logically implies* another formula φ_2, in symbols $\varphi_1 \models \varphi_2$ iff for every LT_f-interpretation π we have that $\pi \models \varphi_1$ implies $\pi \models \varphi_2$.

2.2.3 Results

About LTL_f complexity, we can state the following theorem:

Theorem 2.2. *(De Giacomo and Vardi, 2013) Satisfiability, validity and logical implication for* LTL_f *formulas are* PSPACE*-complete.*

About LTL_f expressiveness, we have that:

Theorem 2.3. *(De Giacomo and Vardi, 2013; Gabbay et al., 1997)* LTL_f *has exactly the same expressive power of* FOL *over finite ordered sequences.*

2.3 Past Linear Temporal Logic (PLTL)

So far we have seen LTL and LTL_f languages, over infinite and finite traces respectively, that look into the future events. On the contrary, now we describe the so called *Past Linear Temporal Logic* (PLTL) which is the counterpart of the LTL and LTL_f because it uses temporal modalities for referring to past events, instead of future ones.

2.3.1 Syntax

The syntax of PLTL is exactly the same of the one seen in Section 2.1.1 for LTL and in Section 2.2.1 for LTL_f except for past temporal operators that are the inverse of the future ones. As stated before, PLTL formulas are built on top from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\ominus$(*previous-time*) and the binaty operator $\mathcal{S}$ (*since*). Formulas can be defined as follows:

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \ominus\varphi \mid \varphi_1 \mathcal{S} \varphi_2$$

where $a \in \mathcal{P}$. All usual logical operators such as $\vee, \Rightarrow, \Leftrightarrow, true$ and $false$ can be derived. Similarly to LTL and LTL$_f$, we define the following common abbreviations for temporal operator:

$$\diamondminus\varphi \doteq true\,\mathcal{S}\,\varphi \tag{2.9}$$

$$\boxminus\varphi \doteq \neg\diamondminus\neg\varphi \tag{2.10}$$

In particular, $\diamondminus\varphi$ in 2.9 is called *once* while $\boxminus\varphi$ in 2.10 is known as *historically*. Furthermore, both temporal operators *previous-time*, *since* and the two common abbreviations *once*, *historically* just defined above could be seen also as the inverse operators of future operators in LTL/LTL$_f$:

$$\ominus\varphi \equiv \bigcirc^{-1}\varphi$$

$$\varphi_1 \mathcal{S} \varphi_2 \equiv \varphi_1 \mathcal{U}^{-1} \varphi_2$$

$$\diamondminus\varphi \equiv \Diamond^{-1}\varphi$$

$$\boxminus\varphi \equiv \Box^{-1}\varphi$$

2.3.2 Semantics

As we did previously with LTL and then with LTL$_f$, here we define a semantics to PLTL. The first important thing to notice is that a PLTL formula could be only interpreted over *finite* traces. This is due to the fact that, no matter how long the trace is, there must be a starting point in the past. Formally, a trace π is a word over the alphabet $2^{\mathcal{P}}$ and as alphabet we have all possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We can now give the following definitions:

Definition 2.5. Given a trace π, we inductively define when a PLTL formula φ is *true* at time i, in symbols $\pi, i \models \varphi$, as follows:

$$\pi, i \models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i)$$
$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$
$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$
$$\pi, i \models \ominus\varphi \text{ iff } i > 0 \wedge \pi, i-1 \models \varphi$$
$$\pi, i \models \varphi_1 \mathcal{S} \varphi_2 \text{ iff } \exists j.(j \leq i) \wedge \pi, j \models \varphi_2 \wedge \forall k.(j < k \leq i) \Rightarrow \pi, k \models \varphi_1$$

The Definition 2.5 is quite similar to Definitions 2.1 and 2.3. The only difference lies on the position in time of instances, indeed, in this case, we go backward.

2.3.3 Results

About PLTL complexity, we can state the following theorem:

Theorem 2.4. *Satisfiability, validity and logical implication for* PLTL *formulas are* PSPACE*-complete.*

2.3.4 Expressiveness

About expressiveness of PLTL, we can state the following theorem:

Theorem 2.5. PLTL *has exactly the same expressive power of* LTL_f*.*

Although from Theorem 2.5 PLTL and LTL have the same expressive power, it is worth to say that the LTL_f formalism augmented with past temporal operators present in PLTL can be exponentially more succinct that LTL_f (with only future operators) (Markey, 2003). Indeed, having at the same time past and future temporal operators is really useful because, in general, expressions given in natural language use references to events occurred in the past. We give an example in the following.

Example 2.4. Succinctness of LTL_f with Past:

$$\Box(grant \Rightarrow \Diamond\!\!\!\!-\, request) \tag{2.11}$$

$$\neg((\neg request)\,\mathcal{U}(grant \wedge \neg request)) \tag{2.12}$$

Both formulas mean *every grant is preceeded by a request*. The former (2.11) is in LTL_f with past modalities whereas the latter (2.12) is pure LTL_f. It is pretty evident that the 2.11 is less intricate than the one in 2.12.

Finally, this property of LTL_f augmented with past temporal operators is interesting, however it is out of the scope of this thesis.

2.4 LTL_f and PLTL Translation to Automata

Given an LTL_f/PLTL formula φ, we can build a deterministic finite state automaton (DFA) (Rabin and Scott, 1959) $\mathcal{A}_\varphi$ that accepts the same finite traces that makes φ true. To achieve this, we proceed in two steps: first, we translate LTL_f and PLTL formulas into an (NFA) (De Giacomo and Vardi, 2015) following a simple direct algorithm; secondly, the obtained NFA can be converted into a DFA following the standard *determinization* procedure.

Now, we recall definitions of NFA and DFA:

Definition 2.6. An NFA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where:

- Σ is the input alphabet;
- Q is the finite set of states;

- $q_0 \in Q$ is the initial state;
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation;
- $F \subseteq Q$ is the set of final states;

Definition 2.7. A DFA is a NFA where δ is a function $\delta : Q \times \Sigma \to Q$

To denote the set of all traces over Σ accepted by $\mathcal{A}$ we will use $\mathcal{L}(A)$ henceforth.

In the next subsections, we will provide some definitions and we will illustrate the algorithm for the translation also giving an example.

2.4.1 ∂ function for LTL$_f$

2.4.2 ∂ function for PLTL

2.5 LTL$_f$/PLTL to FOL Encoding and MONA

In this section, we will illustrate how to translate an LTL$_f$ and a PLTL formula into *first-order logic* (FOL) over finite linear ordered sequences[1] (De Giacomo and Vardi, 2013; Zhu et al., 2018). Then, we will present the MONA tool with its syntax and we will explain the translation procedure from a FOL encoding to the MONA enconding.

2.5.1 LTL$_f$-to-FOL Encoding

In the following we deal with a first-order language augmented with monadic predicates *succ*, $<$ and $=$ plus two constants 0 and *last*. Afterwards, we focus our attention to *finite linear ordered* FOL *interpretations* under the form of $\mathcal{I} = (\Delta^I, \cdot^\mathcal{I})$, where the domain is $\Delta^I = \{0, \ldots, n\}$ with $n \in \mathbb{N}$, and the interpretation function $\cdot^\mathcal{I}$ interprets binary predicates and constants as follows:

$$\begin{aligned} succ^\mathcal{I} &= \{(i, i+1) \mid i \in \{0, \ldots, n-1\}\} \\ <^\mathcal{I} &= \{(i, j) \mid i, j \in \{0, \ldots, n\} \wedge i < j\} \\ =^\mathcal{I} &= \{(i, i) \mid i \in \{0, \ldots, n\}\} \\ 0^\mathcal{I} &= 0 \\ last^\mathcal{I} &= n \end{aligned} \tag{2.13}$$

Actually, all these operators can be derived from $<$ as follows:

$$\begin{aligned} succ(x, y) &\doteq x < y \wedge \neg\exists z.x < z < y \\ x = y &\doteq \forall z.x < z \equiv y < z \\ 0 &\doteq x \mid \neg\exists y.succ(y, x) \end{aligned}$$

[1]More precisely *monadic first-order logic on finite linearly ordered domains*, sometimes denoted as FO[$<$].

$$last \doteq x \mid \neg\exists y.succ(x, y)$$

Although there could be possible differences in notation, the relation between LT$_f$-intepretations and finite linear ordered FOL interpretations is isomorphic. Indeed, given an LT$_f$-interpretation π we can define the corresponding FOL interpretation $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$ as follows: $\Delta^I = \{0, \ldots, last\}$, with $last = length(\pi) - 1$, with the predefined predicates and constants interpretation and, for each $a \in \mathcal{P}$ its interpretation is $a^{\mathcal{I}} = \{i \mid a \in \pi(i)\}$. On the contrary, given a finite linear ordered FOL interpretation $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, with $\Delta^I = \{0, \ldots, n\}$, we determine the corresponding LT$_f$-interpretation π as follows: $length(\pi) = n + 1$, for each instant $0 \leq i \leq last$ (with $last = n$), we obtain $\pi(i) = \{a \mid i \in a^{\mathcal{I}}\}$.

At this moment, we can define the translation function $fol(\varphi, x)$ in the following way.

Definition 2.8. Given an LTL$_f$ formula φ and a variable x, the translation function $fol(\varphi, x)$, inductively defined on the LTL$_f$ formula's structure, returns the corresponding FOL formula open in x:

$$fol(a, x) = a(x)$$

$$fol(\neg\varphi, x) = \neg fol(\varphi, x)$$

$$fol(\varphi_1 \wedge \varphi_2, x) = fol(\varphi_1, x) \wedge fol(\varphi_2, x)$$

$$fol(\varphi_1 \vee \varphi_2, x) = fol(\varphi_1, x) \vee fol(\varphi_2, x)$$

$$fol(\circ\varphi, x) = \exists y.succ(x, y) \wedge fol(\varphi, y)$$

$$fol(\bullet\varphi, x) = x = last \vee \exists y.succ(x, y) \wedge fol(\varphi, y)$$

$$fol(\varphi_1 \,\mathcal{U}\, \varphi_2, x) = \exists y.x \leq y \leq last \wedge fol(\varphi_2, y) \wedge \forall z.x \leq z < y \Rightarrow fol(\varphi_1, z)$$

$$fol(\varphi_1 \,\mathcal{R}\, \varphi_2, x) = \exists y.x \leq y \leq last \wedge fol(\varphi_1, y) \wedge \forall z.x \leq z < y \Rightarrow fol(\varphi_2, z) \vee$$

$$\forall z.x \leq z < last \Rightarrow fol(\varphi_2, z)$$

The following Theorem ensures that a finite trace ρ satisfies an LTL$_f$ formula φ iff the corresponding finite linear ordered FOL interpretation $\mathcal{I}$ of ρ models $fol(\varphi, 0)$.

Theorem 2.6. *(De Giacomo and Vardi, 2013) Given an LT$_f$-interpretation π and a corresponding finite linear ordered FOL interpretation $\mathcal{I}$, we have:*

$$\pi, i \models \varphi \text{ iff } I, [x/i] models fol(\varphi, x)$$

where $[x/i]$ stands for a variable assignments that assigns the value i to the free variable x of $fol(\varphi, x)$.

In general, recalling the Definition 2.4, a formula φ is *true* in a trace π ($\pi \models \varphi$) if $\pi, 0 \models \varphi$. Hence, we should evaluate our translation function $fol(\varphi, x)$ in 0 (i.e. computing $fol(\varphi, 0)$). Finally, since also the converse reduction of Theorem 2.6 holds, we can state the following Theorem:

Theorem 2.7. *(Gabbay et al., 1980)* LTL$_f$ *has exactly the same expressive power of* FOL.

2.5.2 PLTL-to-FOL Encoding

As we have previously seen for LTL$_f$, in the current section we describe the translation function for a PLTL formula. Here, we also have a first-order language augmented with monadic predicates *prev*, $<$ and $=$ plus two constants 0 and *last*. Then, we have our *finite linear ordered* FOL *interpretations* under the form of $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, where the domain is $\Delta^I = \{0, \ldots, n\}$ with $n \in \mathbb{N}$, and the interpretation function $\cdot^{\mathcal{I}}$ interprets the same binary predicates defined as 2.13 except that here we change *succ* with *prev* defined as follows:

$$prev^{\mathcal{I}} = \{(i, i-1) \mid i \in \{1, \ldots, n\}\} \tag{2.14}$$

We can derive these operators from $<$ as well:

$$\begin{aligned} prev(x, y) &\doteq y < x \wedge \neg\exists z. y < z < x \\ x = y &\doteq \forall z. x < z \equiv y < z \\ 0 &\doteq x \mid \neg\exists y. prev(x, y) \\ last &\doteq x \mid \neg\exists y. prev(y, x) \end{aligned}$$

In the exactly same way done before, we can give the definition of the translation function $fol_p(\varphi, x)$:

Definition 2.9. Given a PLTL formula φ and a variable x, the translation function $fol_p(\varphi, x)$, inductively defined on the PLTL formula's structure, returns the corresponding FOL formula open in x:

$$\begin{aligned} &fol_p(a, x) = a(x) \\ &fol_p(\neg\varphi, x) = \neg fol_p(\varphi, x) \\ &fol_p(\varphi_1 \wedge \varphi_2, x) = fol_p(\varphi_1, x) \wedge fol_p(\varphi_2, x) \\ &fol_p(\varphi_1 \vee \varphi_2, x) = fol_p(\varphi_1, x) \vee fol_p(\varphi_2, x) \\ &fol_p(\ominus\varphi, x) = \exists y. prev(x, y) \wedge y \geq 0 \wedge fol_p(\varphi, y) \\ &fol_p(\varphi_1 \,\mathcal{S}\, \varphi_2, x) = \exists y. 0 \leq y \leq x \wedge fol_p(\varphi_2, y) \wedge \forall z. y < z \leq x \Rightarrow fol_p(\varphi_1, z) \end{aligned}$$

Consider a finite trace ρ, the corresponding FOL interpretation $\mathcal{I}$ is defined as in Section 2.5.1. The following Theorem ensures that a finite trace ρ satisfies an PLTL formula φ iff the corresponding finite linear ordered FOL interpretation $\mathcal{I}$ of ρ models $fol_p(\varphi, last)$.

Theorem 2.8. *(Kamp, 1968) Given a PLTL formula φ, a finite trace ρ, and the corresponding interpretation $\mathcal{I}$ of ρ, we have that*

$$\rho \models \varphi \text{ iff } I \models fol_p(\varphi, last)$$

where $last = length(\rho) - 1$.

2.5.3 MONA and FOL-to-MONA Encoding

In the following, firstly we introduce the MONA tool highlighting its main features, how it works and what is its role in this thesis. Secondly, we concentrate on the MONA syntax and we describe the algorithm to translate a FOL formula into a MONA program.

MONA

MONA (Elgaard et al., 1998) is a sophisticated tool written in C/C++ for the construction of symbolic DFA from logical specifications. This tool has been implemented starting from 1997 from the BRICS (a research center in computer science located at the Aarhus University) with the aim of efficiently implementing decision procedures for the *Weak Second-order Theory of One or Two successors* (WS1S/WS2S). These two theories are also called monadic (from here the name of the tool) second-order logics and are decidable[2] since allowed second-order variables are interpreted as a finite set of numbers. Moreover, the WS1S theory is a fragment of arithmetic augmented with second-order quantification over finite sets of natural numbers. Indeed, first-order terms represents just natural numbers. Furthermore, WS1S has not the addition operator because that would make the theory undecidable, however there is the unary predicate +1 that stands for the successor function. On the other hand, WS2S is a generalization of WS1S to tree structures. Hence, MONA efficiently translates WS1S and WS2S formulas respectively into minimum DFAs and GTAs (Guided Tree Automata (Biehl et al., 1996)), representing them by shared, multi-terminal BDDs (Binary Decision Diagrams (Henriksen et al., 1995)). Having considered the polyedric features of MONA, we will only use the translation to DFAs.

MONA has a lot of possible applications that have been published during the years. Additionally, thanks to its APIs, it could be used both as a standalone tool and as an integrated tool for other programs. Some examples of MONA usage are the following:

- Hardware verification
- Controller systems
- Program and Protocol verification
- Software Engineering

[2] A logic is decidable if there exists an algorithm such that for any given formula it determines its truth value.

At this point, we can explain how MONA works, at least for the part related to the DFA construction from a FOL formula. However, before doing that, we would like to clarify what the exact role of MONA is within this thesis. As stated before and as we will see in Chapter 3, MONA has been employed as a tool that translates a monadic FOL formula on finite linearly ordered domains, encoded as a M2L-Str[3], into a minimum DFA.

Now, we can briefly describe how MONA works.

FOL-to-MONA Encoding

The MONA syntax is quite similar to the WS1S syntax, but it has its own method to define variables and it has been enhanced with some special details, also known as syntactic sugar, making the overall language more readable and allowing to express things more clearly and more concisely.

MONA is executed on a file, with *.mona* extension, in which we can find some declarations and WS1S/WS2S formulas. We will refer to such file as the *.mona* program, henceforth. After the execution of the tool with a *.mona* program, we get a DFA. Additionally, MONA carries out an analysis of the program by recognizing the set of satisfying interpretations for the program. Let us consider the following example (Klarlund and Møller, 2001):

Example 2.5. A `simple.mona` program:

```
var2 P,Q;
P\Q = {0,4} union {1,2};
```

First, we have declared `P` and `Q` as second-order variables. After that, we have defined a formula telling that the set difference between `P` and `Q` is the union of set `{0,4}` and `{1,2}`. Obviously, this formula is not always true, nonetheless there is an interpretation that satisfies it. For instance, the assignments `{0,1,2,4}` to `P` and `{5}` to `Q`. This interpretation can also be represented as a bit string for each variable, where positions in the string correspond to natural numbers, 1 means that the number is in the set (remember that a second-order variable is a set) whereas 0 means that is not. In this case, we would have `P`$\rightarrow$ 111010 and `Q`$\rightarrow$ 000001. Thus, it is possible to define a *language* associated to these bit strings and, since it is *regular*, it is also possible to build a DFA. Moreover, MONA assumes that all defined formulas in the program are in conjunct and each statement should be terminated by a semicolon. There are also additional elements consisting the MONA syntax depicted in Figure 2.1. As we can see from that Figure, there are also quantifiers and all usual logical connectives (i.e. those used in FOL). In addition, since we would like to write FOL on *finite linearly ordered domains*, we should enable the M2L-Str mode specifying `m2l-str;` at the beginning of the MONA program. Actually, `m2l-str;` is a shortcut for:

[3] M2L-Str is a slight variation of WS1S where formulas are interpreted over *finite string* models, rather than *infinite string* models

Numbers (1st order terms)	
0	`0`
$+$	`+`
$-$	`-`

Sets (2nd order terms)	
$\emptyset$	`empty`
$\cup$	`union`
$\cap$	`inter`
$\setminus$	`\`
$\{\dots\}$	`{...}`

Formulas (0th order terms)	
0th order arguments	
$\neg$	`~`
$\wedge$	`&`
$\vee$	`|`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\exists$	`ex0 ex1 ex2`
$\forall$	`all0 all1 all2`

Formulas (0th order terms)	
1st order arguments	
$<$	`<`
$>$	`>`
$\leq$	`<=`
$\geq$	`>=`
$=$	`=`
$\neq$	`~=`
2nd order arguments	
$\subseteq$	`sub`
$=$	`=`
$\neq$	`~=`
1st/2nd order arguments	
$\in$	`in`
$\notin$	`notin`

Figure 2.1. The essential MONA syntax.

```
1 ws1s;
2 var2 $ where ~ex1 p where true: p notin $ & p+1 in $;
3 allpos $;
4 defaultwhere1(p) = p in $;
5 defaultwhere2(P) = P sub $;
```

At the first line, it is declared the intent to use exclusively WS1S. Then, at line 2, there is the declaration of a second-order variable `$` ensuring it to always have the value $\{0, \dots, n-1\}$ for some n. Likewise, it is needed the declaration at line 3 to bound the domain of interest. Lastly, at lines 4 and 5, the program restrict all first- and second-order variables to `$`.

At this point, since we have illustrated all the necessary stuff for the translation, we are able to give the FOL-to-MONA encoding with some examples.

To begin with, all usual logic operators can be encoded following the table in Figure 2.1. Secondly, to encode the *succ* and *prev* monadic predicates respectively defined in Equations 2.13 and 2.14 we use the successor and predecessor built-in operators as follows:

$$succ(x, y) \doteq \texttt{y=x+1}$$
$$prev(x, y) \doteq \texttt{y=x-1}$$

Additionally, the two constants 0 and *last* already defined in 2.13 are encoded as `0` and `max($)`, respectively. Thirdly, to express existential and universal quantifiers we use the corresponding syntax as follows:

$$\exists p. \doteq \texttt{ex1 p:}$$
$$\forall p. \doteq \texttt{all1 p:}$$

Then, we can express first-order predicates symbols with set containment. For instance, if we have $A(x)$, before we must declare it as `var2 A;` and, then, encode it as `x in A`, whereas its negation $(\neg A(x))$ would be `x notin A`. Finally, *true* and *false* remain the same. In the following, we give some examples.

Example 2.6. FOL-to-MONA encoding examples:

- Suppose we have the LTL$_f$ formula $\Diamond G$, its translation to FOL according to Definition 2.8 is:

$$\exists y.0 \leq y \leq \mathit{last} \wedge G(y) \tag{2.15}$$

(we have not included the last part $\forall z.0 \leq z < y \Rightarrow \mathit{true}$ since it is trivially *true*). The MONA program corresponding to the formula in 2.15 is the following:

```
m2l-str;
var2 G;
ex1 y: 0<=y & y<=max($) & y in G;
```

- Suppose we have the LTL$_f$ formula $\Box G$, its translation to FOL according to Definition 2.8 is:

$$\neg(\exists y.0 \leq y \leq \mathit{last} \wedge \neg G(y)) \tag{2.16}$$

The MONA program corresponding to the formula in 2.16 is the following:

```
m2l-str;
var2 G;
~(ex1 y: 0<=y & y<=max($) & y notin G);
```

- Suppose we have the PLTL formula $A\,\mathcal{S}\,B)$, its translation to FOL according to Definition 2.9 is:

$$\exists y.0 \leq y \leq \mathit{last} \wedge B(y) \wedge \forall z.y < z \leq \mathit{last} \Rightarrow A(z) \tag{2.17}$$

The MONA program corresponding to the formula in 2.17 is the following:

```
m2l-str;
var2 A,B;
(ex1 y: 0<=y & y<=max($) & y in B & (all1 z: y<z & z<=max($) => z in A));
```

2.6 Summary

In this chapter, we have illustrated the theoretical framework, consisted of LTL, LTL$_f$ and PLTL formalisms, underlying the thesis. These formal languages have been described focusing the attention on their syntax, semantics and interesting properties. Besides, we have talked about the theory behind the translation procedure of LTL$_f$ and PLTL formulas to DFAs. Finally, we have presented the MONA tool explaining in details the encoding process starting from an LTL$_f$/PLTL formula to a MONA program passing through a FOL translation.

Chapter 3

LTL$_f$2DFA

In this chapter we will present LTL$_f$2DFA, a software package written in Python.

3.1 Introduction

LTL$_f$2DFA is a Python tool that processes a given LTL$_f$/PLTL formula and generates the corresponding minimized DFA using MONA (Elgaard et al., 1998). In addition, it offers the possibility to compute the DFA with or without the DECLARE assumption (De Giacomo et al., 2014). The main features provided by the library are:

- parsing an LTL$_f$/PLTL formula;
- translation of an LTL$_f$/PLTL formula to MONA program;
- conversion of an LTL$_f$/PLTL formula to DFA automaton.

LTL$_f$2DFA can be used with Python>=3.6 and has the following dependencies:

- PLY, a pure-Python implementation of the popular compiler construction tools Lex and Yacc. It has been employed for parsing the input LTL$_f$ formula;
- MONA, a C++ tool that translates formulas to DFA. It has been used for the generation of the DFA;
- Dotpy, a Python library able to parse and modify `.dot` files. It has been utilized for post-processing the MONA output.

The package is available to download on PyPI and you can install it by typing in the terminal:

```
pip install ltlf2dfa
```

All the code is available online on GitHub[1], it is open source and it is released under the MIT License. Moreover, LTL$_f$2DFA can also be tried online at ltlf2dfa.diag.uniroma1.it.

[1] https://github.com/Francesco17/LTLf2DFA

3.2 Package Structure

The structure of the LTL$_f$2DFA package is quite simple. It consists of a main folder called `ltlf2dfa/` which hosts the most important library's modules:

- `Lexer.py`, where the Lexer class is defined;
- `Parser.py`, where the Parser class is defined;
- `Translator.py`, where the main APIs for the translation are defined;
- `DotHandler.py`, where we the MONA output is post-processed.

In the following paragraphs we will explore each module in detail.

3.2.1 Lexer.py

In the `Lexer.py` module we can find the declaration of the `MyLexer` class which is in charge of handling the input string and tokenizing it. Indeed, it implements a tokenizer that splits the input string into declared individual tokens. To our extent, we have defined the class as in Listing 3.1

Listing 3.1. `Lexer.py` module

```
import ply.lex as lex

class MyLexer(object):

    reserved = {
        'true':    'TRUE',
        'false':   'FALSE',
        'X':       'NEXT',
        'W':       'WEAKNEXT',
        'R':       'RELEASE',
        'U':       'UNTIL',
        'F':       'EVENTUALLY',
        'G':       'GLOBALLY',
        'Y':       'PASTNEXT', #PREVIOUS
        'S':       'PASTUNTIL', #SINCE
        'O':       'PASTEVENTUALLY', #ONCE
        'H':       'PASTGLOBALLY' #HISTORICALLY
    }
    # List of token names. This is always required
    tokens = (
        'TERM',
        'NOT',
        'AND',
```

```
        'OR',
        'IMPLIES',
        'DIMPLIES',
        'LPAR',
        'RPAR'
    ) + tuple(reserved.values())

    # Regular expression rules for simple tokens
    t_TRUE = r'true'
    t_FALSE = r'false'
    t_AND = r'\&'
    t_OR = r'\|'
    t_IMPLIES = r'\->'
    t_DIMPLIES = r'\<->'
    t_NOT = r'\~'
    t_LPAR = r'\('
    t_RPAR = r'\)'
    # FUTURE OPERATORS
    t_NEXT = r'X'
    t_WEAKNEXT = r'W'
    t_RELEASE = r'R'
    t_UNTIL = r'U'
    t_EVENTUALLY = r'F'
    t_GLOBALLY = r'G'
    # PAST OPERATOR
    t_PASTNEXT = r'Y'
    t_PASTUNTIL = r'S'
    t_PASTEVENTUALLY = r'O'
    t_PASTGLOBALLY = r'H'

    t_ignore = r' '+'\n'

    def t_TERM(self, t):
        r'(?<![a-z])(?!true|false)[_a-z0-9]+'
        t.type = MyLexer.reserved.get(t.value, 'TERM')
        return t # Check for reserved words

    def t_error(self, t):
        print("Illegal character '%s' in the input formula" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    def build(self,**kwargs):
```

```
67        self.lexer = lex.lex(module=self, **kwargs)
```

Firstly, we have defined the reserved words within a dictionary so to match each reserved word with its identifier. Secondly, we have defined the tokens list with all possible tokens that can be produced by the lexer. This tokens list is always required for the implementation of a lexer. Then, each token has to be specified by writing a regular expression rule. If the token is simple it can be specified using only a string. Otherwise, for non trivial tokens we have to write the regular expression in a class method as for our token `TERM` in line 56. In that case, defining the token rule as a method is also useful when we would like to perform other actions. After that, we have a method to handle unrecognized tokens and, finally, we have written the function that builds the lexer.

3.2.2 Parser.py

In the `Parser.py` module we can find the declaration of `MyParser` class which implements the parsing component of `PLY`. The `MyParser` class operates after the Lexer has split the input string into known tokens. The main feature of the parser is to interpret and build the appropriate data structure for the given input. To this extent, the most important aspect of a parser is the definition of the *syntax*, usually specified in terms of a BNF[2] grammar, that should be unambiguous. Furthermore, `Yacc`, the parsing component of `PLY`, implements a parsing technique known as LR-parsing or shift-reduce parsing. In particular, this parsing technique works on a bottom up fashion that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side and so on until there is no more rule to apply. The parser implementation is shown in Listing 3.2

Listing 3.2. `Parser.py` module

```python
import ply.yacc as yacc
from ltlf2dfa.Lexer import MyLexer

class MyParser(object):

    def __init__(self):
        self.lexer = MyLexer()
        self.lexer.build()
        self.tokens = self.lexer.tokens
        self.parser = yacc.yacc(module=self)
        self.precedence = (

            ('nonassoc', 'LPAR', 'RPAR'),
            ('left', 'AND', 'OR', 'IMPLIES', 'DIMPLIES', 'UNTIL', \
              'RELEASE', 'PASTUNTIL'),
```

[2]The Backus–Naur form is a notation technique for context-free grammars.

```
            ('right', 'NEXT', 'WEAKNEXT', 'EVENTUALLY', \
            'GLOBALLY', 'PASTNEXT', 'PASTEVENTUALLY', 'PASTGLOBALLY'),
            ('right', 'NOT')
        )

    def __call__(self, s, **kwargs):
        return self.parser.parse(s, lexer=self.lexer.lexer)

    def p_formula(self, p):
        '''
        formula : formula AND formula
                | formula OR formula
                | formula IMPLIES formula
                | formula DIMPLIES formula
                | formula UNTIL formula
                | formula RELEASE formula
                | formula PASTUNTIL formula
                | NEXT formula
                | WEAKNEXT formula
                | EVENTUALLY formula
                | GLOBALLY formula
                | PASTNEXT formula
                | PASTEVENTUALLY formula
                | PASTGLOBALLY formula
                | NOT formula
                | TRUE
                | FALSE
                | TERM
        '''

        if len(p) == 2: p[0] = p[1]
        elif len(p) == 3:
            if p[1] == 'F': # F(a) == true UNITL A
                p[0] = ('U','true', p[2])
            elif p[1] == 'G': # G(a) == not(eventually (not A))
                p[0] = ('~',('U', 'true', ('~',p[2])))
            elif p[1] == 'O': # O(a) = true SINCE A
                p[0] = ('S','true', p[2])
            elif p[1] == 'H': # H(a) == not(pasteventually(not A))
                p[0] = ('~',('S', 'true', ('~',p[2])))
            else:
                p[0] = (p[1], p[2])
        elif len(p) == 4:
```

```
            if p[2] == '->':
                p[0] = ('|', ('~', p[1]), p[3])
            elif p[2] == '<->':
                p[0] = ('&', ('|', ('~', p[1]), p[3]), ('|', ('~', p[3]),\
                p[1]))
            else:
                p[0] = (p[2],p[1],p[3])
        else: raise ValueError


    def p_expr_group(self, p):
        '''
        formula : LPAR formula RPAR
        '''
        p[0] = p[2]

    def p_error(self, p):
        raise ValueError("Syntax error in input! %s" %str(p))
```

As we can see, as soon as the parser is instantiated it builds the lexer, gets the tokens and defines their precedence if needed. Then, we have defined methods of the `MyParser` class that are in charge of constructing the syntax tree structure from tokens found by the lexer in the input string. In our case, we have chosen to use as data structure a tuple of tuples as it is the one of the simplest data structure in Python. In general, a tuple of tuples represents a tree where each node represents an item present in the formula.

For instance, the LTL$_f$ formula $\varphi = G(a \Rightarrow Xb)$ is represented as $('\sim', ('U',' true', ('\sim' , ('|', ('\sim',' a'), ('X',' b')))))$ and it corresponds to a tree as the one depicted in Figure 3.1. Finally, as in the `MyLexer` class, we have to handle errors defining a specific method.

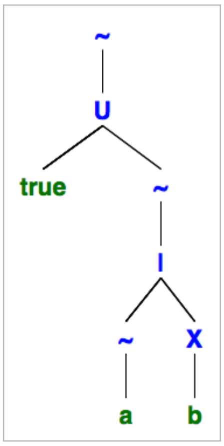


Figure 3.1. The syntax tree generated for the formula "$G(a \sim Xb)$". Symbols are in green while operators are in blue.

LTL$_f$2DFA can be used just for the parsing phase of an LTL$_f$/PLTL formula as shown in Listing 3.3.

Listing 3.3. How to use only the parsing phase of LTL$_f$2DFA.

```
from ltlf2dfa.Parser import MyParser

formula = "G(a->Xb)"
parser = MyParser()
parsed_formula = parser(formula)

print(parsed_formula) # syntax tree as tuple of tuples
```

3.2.3 Translator.py

The `Translator.py` module contains the majority of APIs that the LTL$_f$2DFA package exposes. Indeed, this module consists of a `Translator` class which concerns the core feature of the package: the translation of an LTL$_f$/PLTL formula into the corresponding minimum DFA. Since the package takes advantage of the MONA tool for the formula conversion, the `Translator` class has to translate first the given formula into the syntax recognized by MONA, then create the input program for MONA and, finally, invoke MONA to get back the resulting DFA in the Graphviz[3] format. The main methods of the `Translator` class are:

- `translate()`, which starting from the formula syntax tree generated (Figure 3.1) in the parsing phase translates it into a string using the syntax of MONA;
- `createMonafile(flag)`, which, as the name suggests, creates the program *.mona* that will be given as input to MONA. The flag parameter is going to be `True` of `False` whether we need to compute also DECLARE assumptions or not;
- `invoke_mona()`, which invokes MONA in order to obtain the DFA.

Now we will go into details of the methods stated above showing their implementation.

The `translate` method

The `translate` method is a crucial step towards reaching a good result and performance. Formally, the translation procedure from an LTL$_f$/PLTL formula to the MONA syntax is done passing through FOL as shown in 3.1.

$$\mathrm{LTL}_f/\mathrm{PLTL} \rightarrow \mathrm{FOL} \rightarrow \mathrm{MONA} \tag{3.1}$$

The former translation from LTL$_f$/PLTL to FOL is done accordingly to (De Giacomo and Vardi, 2013), while the latter follows from (Klarlund and Møller, 2001), as seen in

[3]Graphviz is open source graph visualization software. For further details see https://www.graphviz.org

Section 2.5. In Listing 3.4 we can see the translation's implementation. Three dots ... represent omitted code.

Listing 3.4. The `translate` method.

```
import ...

class Translator:
    ...

    def translate(self):
        self.translated_formula = translate_bis(self.parsed_formula, \
        self.formulaType, var='v_0')+";\n"

   ...

def translate_bis(formula_tree, _type, var):
    if type(formula_tree) == tuple:
        if formula_tree[0] == '&':
            if var == 'v_0':
                if _type == 2:
                    a = translate_bis(formula_tree[1], _type, 'max($)')
                    b = translate_bis(formula_tree[2], _type, 'max($)')
                else:
                    a = translate_bis(formula_tree[1], _type, '0')
                    b = translate_bis(formula_tree[2], _type, '0')
            else:
                a = translate_bis(formula_tree[1], _type, var)
                b = translate_bis(formula_tree[2], _type, var)
            if a == 'false' or b == 'false':
                return 'false'
            elif a == 'true':
                if b == 'true': return 'true'
                else: return b
            elif b == 'true': return a
            else: return '('+a+' & '+b+')'
        elif formula_tree[0] == '|':
            if var == 'v_0':
                if _type == 2:
                    a = translate_bis(formula_tree[1], _type, 'max($)')
                    b = translate_bis(formula_tree[2], _type, 'max($)')
                else:
                    a = translate_bis(formula_tree[1], _type, '0')
                    b = translate_bis(formula_tree[2], _type, '0')
            else:
```

```
            a = translate_bis(formula_tree[1], _type, var)
            b = translate_bis(formula_tree[2], _type, var)
        if a == 'true' or b == 'true':
            return 'true'
        elif a == 'false':
            if b == 'true': return 'true'
            elif b == 'false': return 'false'
            else: return b
        elif b == 'false': return a
        else: return '('+a+'␣|␣'+b+')'
    elif formula_tree[0] == '~':
        if var == 'v_0':
            if _type == 2:
                a = translate_bis(formula_tree[1], _type, 'max($)')
            else:
                a = translate_bis(formula_tree[1], _type, '0')
        else: a = translate_bis(formula_tree[1], _type, var)
        if a == 'true': return 'false'
        elif a == 'false': return 'true'
        else: return '~('+ a +')'
    elif formula_tree[0] == 'X':
        new_var = _next(var)
        a = translate_bis(formula_tree[1], _type, new_var)
        if var == 'v_0':
            return '('+ 'ex1␣'+new_var+':␣'+ new_var +'␣=␣1␣'+ '&␣'+ \
            a +')'
        else:
            return '('+ 'ex1␣'+new_var+':␣'+ new_var +'␣=␣'+ var + \
            '␣+␣1␣'+ '&␣'+ a +')'
    elif formula_tree[0] == 'U':
        new_var = _next(var)
        new_new_var = _next(new_var)
        a = translate_bis(formula_tree[2], _type, new_var)
        b = translate_bis(formula_tree[1], _type, new_new_var)

        if var == 'v_0':
            if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
            new_var+'␣&␣'+ new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
            elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
            new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+ \
            new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+ \
            new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
            elif a == 'false': return 'false'
```

```
            else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+new_var+ \
            '␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣&␣all1␣'+ \
            new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+ \
            new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
        else:
            if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+ \
            '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
            elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+ \
            '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+ \
            new_new_var+':␣'+var+'␣<=␣'+new_new_var+'␣&␣'+ \
            new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
            elif a == 'false': return 'false'
            else: return '(␣'+ 'ex1␣'+new_var+':␣'+var+'␣<=␣'+ \
            new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a + \
            '␣&␣all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
            '␣&␣'+new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
    elif formula_tree[0] == 'W':
        new_var = _next(var)
        a = translate_bis(formula_tree[1], _type, new_var)
        if var == 'v_0':
            return '(0␣=␣max($))␣|␣('+ 'ex1␣'+new_var+':␣'+new_var+\
            '␣=␣1␣'+ '&␣'+ a +')'
        else:
            return '('+ var +'␣=␣max($))␣|␣('+ 'ex1␣'+new_var+':␣'+\
            new_var +'␣=␣'+ var + '␣+␣1␣'+ '&␣'+ a +')'

    elif formula_tree[0] == 'R':
        new_var = _next(var)
        new_new_var = _next(new_var)
        a = translate_bis(formula_tree[2], _type, new_new_var)
        b = translate_bis(formula_tree[1], _type, new_var)

        if var == 'v_0':
            if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'\
            +new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+\
            new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+\
            new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
            '(all1␣'+new_new_var+':␣0␣<=␣'+new_new_var+\
            '␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
            elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+\
            new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+b+')'
            elif b == 'false': return '(all1␣'+new_new_var+':␣0␣<=␣'+\
            new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
```

```
                else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+new_var+\
                '␣&␣'+new_var+'␣<=␣max($)␣&␣'+ b +'␣&␣all1␣'+\
                new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
                '(all1␣'+new_new_var+':␣0␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
            else:
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+\
                '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+\
                new_new_var+':␣'+var+'␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
                '(all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+\
                '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+b+')'
                elif b == 'false': return '(all1␣'+new_new_var+':␣'+\
                var+'␣<=␣'+new_new_var+'␣&␣'+new_new_var+\
                '␣<=␣max($)␣=>␣'+a+'␣)'
                else: return '(␣'+ 'ex1␣'+new_var+':␣'+var+'␣<=␣'+\
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ b +\
                '␣&␣all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
                '(all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
        elif formula_tree[0] == 'Y':
            new_var = _next(var)
            a = translate_bis(formula_tree[1], _type, new_var)
            if var == 'v_0':
                return '('+ 'ex1␣'+new_var+':␣'+ new_var + \
                '␣=␣max($)␣-␣1␣'+ '&␣max($)␣>␣0␣&␣'+ a +')'
            else:
                return '('+ 'ex1␣'+new_var+':␣'+ new_var + \
                '␣=␣'+ var + '␣-␣1␣'+ '&␣'+new_var+'␣>␣0␣&␣'+ a +')'
        elif formula_tree[0] == 'S':
            new_var = _next(var)
            new_new_var = _next(new_var)
            a = translate_bis(formula_tree[2], _type, new_var)
            b = translate_bis(formula_tree[1], _type, new_new_var)

            if var == 'v_0':
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+ \
```

```
170                 '␣:␣0␣<=␣'+new_var+'␣&␣'+new_var+ \
171                 '␣<=␣max($)␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
172                 new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+b+'␣)'
173                 elif a == 'false': return 'false'
174                 else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
175                 new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a + \
176                 '␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
177                 new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+b+'␣)'
178             else:
179                 if b == 'true': return '(␣'+ 'ex1␣'+new_var+ \
180                 ':␣0␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
181                 elif a == 'true': return '(␣'+ 'ex1␣'+new_var+ \
182                 ':␣0␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣'+var+ \
183                 '␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
184                 new_new_var+'␣&␣'+new_new_var+'␣<=␣'+var+'␣=>␣'+b+'␣)'
185                 elif a == 'false': return 'false'
186                 else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
187                 new_var+'␣&␣'+new_var+'␣<=␣'+var+'␣&␣'+ a +'␣&␣all1␣'+ \
188                 new_new_var+':␣'+new_var+'␣<␣'+new_new_var+'␣&␣'+ \
189                 new_new_var+'␣<=␣'+var+'␣=>␣'+b+'␣)'
190     else:
191         # handling non-tuple cases
192         if formula_tree == 'true': return 'true'
193         elif formula_tree == 'true': return 'false'
194 
195         # BASE CASE OF RECURSION
196         else:
197             if var == 'v_0':
198                 if _type == 2:
199                     return 'max($)␣in␣'+ formula_tree.upper()
200                 else:
201                     return '0␣in␣'+ formula_tree.upper()
202             else:
203                 return var + '␣in␣' + formula_tree.upper()
204 
205 def _next(var):
206     if var == '0' or var == 'max($)': return 'v_1'
207     else:
208         s = var.split('_')
209         s[1] = str(int(s[1])+1)
210         return '_'.join(s)
```

As we can see, the `translate` method is actually very simple. In fact, it just calls the `translate_bis` function (line 12) to perform the proper translation. The function works in a recursive fashion taking as input the parsed formula and a variable and outputting a string containing the result. Obviously, when an instance of the `Translator` class is created the input formula is checked to have either only future or past operators. The base case of the recursion handles the translation of symbols as they are the leaves of the syntax tree composed in the parsing phase (Figure 3.1). On the other hand, the recursive step regards the handling of operators (non leaf components of the syntax tree) which are in our case $\wedge$, $\vee$, $\neg$, $\bigcirc$, $\mathcal{U}$, $\ominus$, $\mathcal{S}$. During the translation, we simplify the resulting formula by avoiding pieces of the expression that are logically `True` or `False`. This simplification has two main advantages. First, it substantially reduces the length of the resulting formula, improving its readability. Second, it increases the computation performances of MONA. Additionally, since the MONA syntax requires the declaration of the free variables, the `translate_bis` function has to compute also the appriopriate free variables declaration. In this terms, the translation function uses the `_next` function to compute the next variable each time is needed.

The `createMonafile` method

The `createMonafile` method is employed to write the program *.mona* and save it in the main directory. It takes as input a boolean flag that, as stated before, stands for indicating whether one would like to compute and add the DECLARE assumption or not. In particular, in formal logic, as stated in (De Giacomo et al., 2014), the DECLARE assumption is expressed as in 3.2.

$$\Box(\bigvee_{a\in\mathcal{P}} a) \wedge \Box(\bigwedge_{a,b\in\mathcal{P},a\neq b} a \Rightarrow \neg b) \tag{3.2}$$

It consists essentially in two parts joined by the $\wedge$ operator. The former indicates that it is always true that at each point in time only one symbol is *true*, while the latter means that always for each couple of different symbols in the formula if one is *true* the other must be *false*. The practical part can be seen in Listing 3.5.

Listing 3.5. The `createMonafile` method.

```
...
    def compute_declare_assumption(self):
        pairs = list(it.combinations(self.alphabet, 2))

        if pairs:
            first_assumption = "~(ex1␣y:␣0<=y␣&␣y<=max($)␣&␣~("
            for symbol in self.alphabet:
                if symbol == self.alphabet[-1]: first_assumption += \
                '␣y␣in␣'+ symbol +'))'
                else : first_assumption += '␣y␣in␣'+ symbol +'␣|␣'

            second_assumption = "~(ex1␣y:␣0<=y␣&␣y<=max($)␣&␣~("
            for pair in pairs:
```

```
                if pair == pairs[-1]: second_assumption += '(y notin '+ \
                pair[0]+' | y notin '+pair[1]+ ')));'
                else: second_assumption += '(y notin '+ pair[0]+ \
                ' | y notin '+pair[1]+ ') & '

            return first_assumption +' & '+ second_assumption
        else:
            return None

    def buildMonaProgram(self, flag_for_declare):
        if not self.alphabet and not self.translated_formula:
            raise ValueError
        else:
            if flag_for_declare:
                if self.compute_declare_assumption() is None:
                    if self.alphabet:
                        return self.headerMona + \
                        'var2 ' + ", ".join(self.alphabet) + ';\n' + \
                         self.translated_formula
                    else:
                        return self.headerMona + self.translated_formula
                else: return self.headerMona + 'var2 ' +\
                 ", ".join(self.alphabet) + ';\n' + \
                 self.translated_formula + \
                 self.compute_declare_assumption()
            else:
                if self.alphabet:
                    return self.headerMona + 'var2 ' +\
                     ", ".join(self.alphabet) + ';\n' + \
                     self.translated_formula
                else:
                    return self.headerMona + self.translated_formula

    def createMonafile(self, flag):
        program = self.buildMonaProgram(flag)
        try:
            with open('./automa.mona', 'w+') as file:
                file.write(program)
                file.close()
        except IOError:
            print('Problem with the opening of the file!')
...
```

As shown in the code, the `createMonafile` method calls another method, the `buildMonaProgram` (line 23), which literally builds the *.mona* program by joining all pieces that should belong to it. Instead, regarding the DECLARE assumption, if needed, it is added to the *.mona* program directly translated through `compute_declare_assumption` method at line 2.

The `invoke_mona` method

Finally, the `invoke_mona` method is the one that executes the MONA compiled executable giving it the *.mona* program. Consequently, the DFA resulting from the computation of MONA will be stored in the main directory. As stated in 3.1, the LTL$_f$2DFA package requires MONA to be installed. Indeed, without this requirements the `invoke_mona` method will raise an error. The implementation can be seen in Listing 3.6.

Listing 3.6. The `invoke_mona` method.

```
...
    def invoke_mona(self, path='./inter-automa'):
        if sys.platform == 'linux':
            package_dir = os.path.dirname(os.path.abspath(__file__))
            mona_path = pkg_resources.resource_filename('ltlf2dfa','mona')
            if os.access(mona_path, os.X_OK): #check if mona is executable
                try:
                    subprocess.call(package_dir+'/./mona␣-u␣-gw␣' + \
                    './automa.mona␣>␣' + path + '.dot', shell=True)
                except subprocess.CalledProcessError as e:
                    print(e)
                    exit()
                except OSError as e:
                    print(e)
                    exit()
            else:
                print('[ERROR]:␣MONA␣tool␣is␣not␣executable...')
                exit()
        else:
            try:
                subprocess.call('mona␣-u␣-gw␣./automa.mona␣>␣' + path + \
                '.dot', shell=True)
            except subprocess.CalledProcessError as e:
                print(e)
                exit()
            except OSError as e:
                print(e)
                exit()
...
```

To the execute of the MONA tool we have leveraged the built-in module `subprocess` that enables to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

Unfortunately, the DFA resulting from MONA needs to be post-processed because of some extra states added for other purposes not relevant for us. This aspect will be better explained in the following subsection 3.2.4.

3.2.4 DotHandler.py

The `DotHandler` class has been created in order to manage separately and better the post-processing of the DFA, in *.dot* format, resulting from the computation of MONA. Indeed, since MONA has been developed for different purposes, its output has an additional initial state and transition that to our intent are completely meaningless.

Additionally, the interaction with the *.dot* format has been implemented thanks to the `dotpy` library (available on GitHub[4]) developed for this specific purpose paying particular attention to performances.

As we can see in the implementation of the `DotHandler` class in Listing 3.7, the main methods are `modify_dot` and `output_dot`.

Listing 3.7. The `DotHandler` class.

```
from dotpy.parser.parser import MyParser
import os

class DotHandler:

    def __init__(self, path='./inter-automa.dot'):
        self.dot_path = path
        self.new_digraph = None

    def modify_dot(self):
        if os.path.isfile(self.dot_path):
            parser = MyParser()
            with open(self.dot_path, 'r') as f:
                dot = f.read()
                f.close()

            graph = parser(dot)
            if not graph.is_singleton():
                graph.delete_node('0')
                graph.delete_edge('init', '0')
                graph.delete_edge('0', '1')
                graph.add_edge('init', '1')
```

[4] https://github.com/Francesco17/dotpy

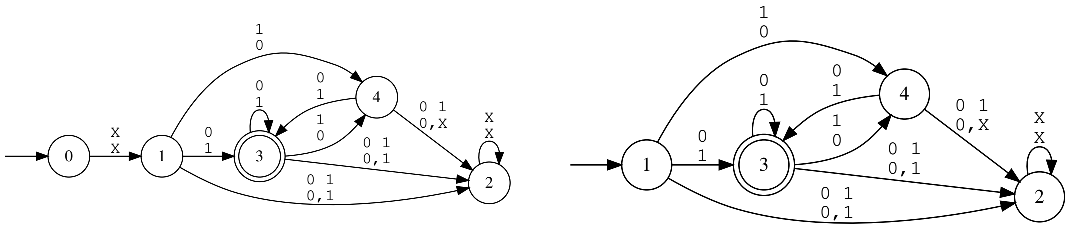
```
23            else:
24                graph.delete_edge('init', '0')
25                graph.add_edge('init', '0')
26            self.new_digraph = graph
27        else:
28            print('[ERROR]␣-␣No␣file␣DOT␣exists')
29            exit()
30
31    def delete_intermediate_automaton(self):
32        if os.path.isfile(self.dot_path):
33            os.remove(self.dot_path)
34            return True
35        else:
36            return False
37
38    def output_dot(self, result_path='./automa.dot'):
39        try:
40            if self.delete_intermediate_automaton():
41                with open(result_path, 'w+') as f:
42                    f.write(str(self.new_digraph))
43                    f.close()
44            else:
45                raise IOError('[ERROR]␣-␣Something␣wrong␣occurred␣in␣'+ \
46                'the␣elimination␣of␣intermediate␣automaton.')
47        except IOError:
48            print('[ERROR]␣-␣Problem␣with␣the␣opening␣of␣the␣file␣%s!' \
49            %result_path)
```

The former method at line 10 takes advantage of the APIs exposed by `dotpy`. Especially, it parses the *.dot* file output of MONA (Figure 3.2a), deletes the starting node 0 and the edge from node 0 to node 1 and, finally, makes node 1 initial. Consequently, the latter method at line 38 manages the output of the final post-processed DFA (Figure 3.2b) and stores it in the main directory. For instance, in Figure 3.2 we can see graphically what is the outcome of the post-processing of the automaton corresponding to the formula $\varphi = \Box(a \Rightarrow \bigcirc b)$.

3.3 Interpreting LTL$_f$2DFA output

In this section, we explain through examples how to interpret and read the output DFA resulting from the LTL$_f$2DFA computation.

To begin with, circle nodes represents automaton states and doubled circle nodes represents those state that are accepting or final for the automaton. Labels on transitions stand for all possible values of formula symbols. A specific formula symbol in a transition must have one of the following values:



(a) The DFA output from MONA

(b) The DFA post-processed

Figure 3.2. Before and after DFA post-processing

- **1**: means that the formula symbol is *true* in that transition;
- **0**: means that the formula symbol is *false* in that transition;
- **X**: means *don't care*, i.e. the formula symbol can be both *true* or *false* in that transition. In other words, it means that the transition can be done no matter is the actual value of the formula symbol.

Finally, when a formula has multiple symbols, the value of each symbol has to be read vertically in order of symbols declaration in the formula. In the following, we will give some examples.

Example 3.1. Let us consider the formula $\varphi = \Diamond g$ and its corresponding automaton depicted in Figure 3.3. The first transition without label indicates the initial state.

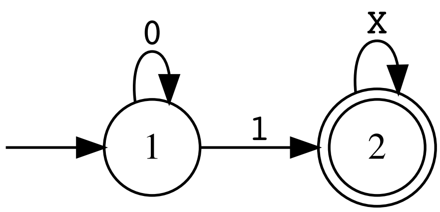


Figure 3.3. Minimum DFA for the formula $\varphi = \Diamond g$.

Then, the first loop on state 1 is done when g is *false*. Afterwards, the transition from state 1 to state 2 can be done only if g is *true*. Finally, the loop on state 2 has the label "X" meaning that once the automaton has arrived on state 2, whatever action it does (also g and $\neg g$) it remains on state 2, which is, by the way, final for the automaton.

Example 3.2. Let us consider the formula $\varphi = \Box(a \Rightarrow \Diamond b)$ and its corresponding automaton depicted in Figure 3.4. As usual, state 1 is the starting state. However, this case is a little bit different from the previous one. Indeed, now the formula has two symbols, namely a and b. Since the order of declaration is a, b, labels on transition has to be read vertically following this order. For instance, the label on transition from state 1 to state 2 reports $\substack{0 \\ \mathrm{X}}$ meaning that the automaton can walk this transition only if a

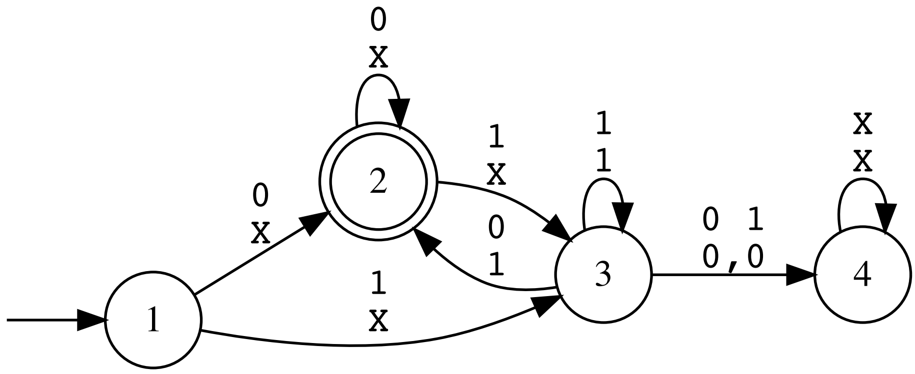


Figure 3.4. Minimum DFA for the formula $\varphi = \Diamond g$.

is *false* (in this case, b is *don't care*, i.e. it can assume whatever value). Additionally, another interesting transition to comment is the one that goes from state 3 to state 4. Its label reports $\genfrac{}{}{0pt}{}{0}{0},\genfrac{}{}{0pt}{}{1}{0}$ meaning that the automaton will do that transition only if either a and b are *false* or a is *true* and b is *false*.

3.4 Comparison with FLLOAT

In this section, we will see how LTL$_f$2DFA performs compared to FLLOAT[5], which is another Python package having the conversion of an LTL$_f$ formula to a DFA as one of its features. In particular, FLLOAT handles LTL$_f$ and LDL$_f$ (*Linear Dynamic Logic on Finite Traces*) formulas, except to PLTL ones, but it provides support for syntax, semantics and parsing of PL (*Propositional Logic*), LTL$_f$ and LDL$_f$ formal languages. Additionally, its conversion is based on a different theoretical result with respect to LTL$_f$2DFA. Especially, FLLOAT directly implements the algorithm seen in Section 2.4.1 and, therefore, it could also accept traces of length 0 (empty traces). This is not the same for LTL$_f$2DFA, since it assumes traces to have at least one element. Nevertheless, we can compare them on the generation of a DFA from an LTL$_f$ formula just by forcing FLLOAT to produce DFAs concerning traces with at least one element. This could be achieved adding *true* to FLLOAT formulas with the $\wedge$ connective. For instance, given an LTL$_f$ formula φ, FLLOAT will compute the DFA for $\psi = true \wedge \varphi$.

The time execution benchmarks between these two packages was done over a set of 13 different interesting LTL$_f$ formulas of different length. The comparison consisted of executing each package over the same set of formulas n number of times and, then, repeating the multiple execution m number of times. Thus, for each formula to be converted we obtained $n \times m$ results and, finally, we kept the minimum one (i.e. the best time execution result). After gathering the results, we can show them on a histogram where on the $x-$axis there are the LTL$_f$ formulas and on the $y-$axis there is the minimum time (in seconds) needed for the package to convert it into a DFA (Figure 3.5). In the

[5]https://github.com/MarcoFavorito/flloat

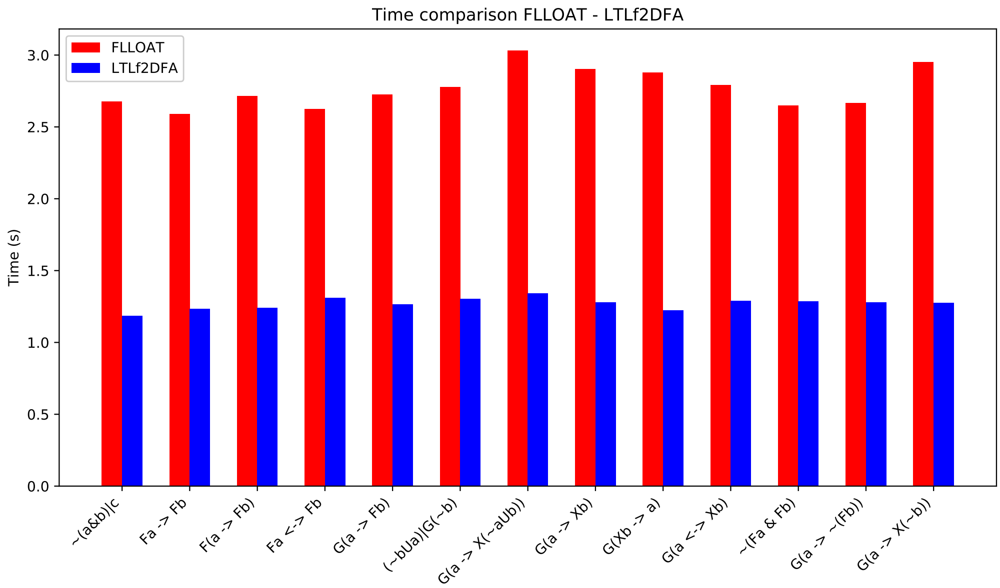


Figure 3.5. Time benchmarking of LTL$_f$2DFA wrt FLLOAT.

histogram, FLLOAT results are coloured in red, while LTL$_f$2DFA ones are depicted in blue.

As we can see from the bar chart, in FLLOAT the time needed to convert the formula increases as the length of the formula grows, whereas in LTL$_f$2DFA it is almost steady. Furthermore, it is notable that LTL$_f$2DFA is overall more than twice as fast as FLLOAT. This behaviour is due to the fact that these two packages operates in a different way. Indeed, while FLLOAT is a pure Python package, LTL$_f$2DFA uses, for the heavy task of the generation of the automaton, MONA that is written in C/C++. Hence, the real difference relies on the performance differences between C/C++ and Python programs. As a final remark, although LTL$_f$2DFA is much faster than FLLOAT, its time execution depends on the I/O system performance which can drastically reduce it. Thus, LTL$_f$2DFA results may arise depending on various factors such as disk speed, caching and filesystem.

3.5 Discussion

In this chapter, we have presented the LTL$_f$2DFA Python package. We have also described the structure of the package, discussed in detail its implementation highlighting all the main features and, finally, seen how it performs in time relatively to the FLLOAT Python package.

Chapter 4

Planning for Extended Temporal Goals

In this chapter, we will define a new approach to the problem of non-deterministic planning for extended temporal goals. In particular, we will give a solution to this problem reducing it to a fully observable non-deterministic (FOND) problem and taking advantage of our tool LTL$_f$2DFA, presented in Chapter 3. First of all, we will introduce the main idea and motivations supporting our approach. Then, we will give some preliminaries explaining the Planning Domain Definition Language (PDDL) language and the FOND planning problem formally. After that, we will illustrate our solution with the encoding of temporal goals into a PDDL domain and problem. Finally, we will present our practical implementation of the proposed solution.

4.1 Idea and Motivations

Planning for temporally extended goals with *deterministic* actions has been well studied during the years starting from (Bacchus and Kabanza, 1998) and (Doherty and Kvarnstram, 2001). Two main reasons why temporally extended goals have been considered over the classical goals, viewed as a desirable set of final states to be reached, are because they are not limited in what they can specify and they allows us to restrict the manner used by the plan to reach the goals. Indeed, temporal extended goals are fundamental for the specification of a collection of real-world planning problems. Yet, many of these real-world planning problems have a *non-deterministic* behavior owing to unpredictable environmental conditions. However, planning for temporally extended goals with *non-deterministic* actions is a more challenging problem and has been of increasingly interest only in recent years with (Camacho et al., 2017).

In this scenario, we have devised a solution to this problem that exploits the translation of a temporal formula to a DFA, using LTL$_f$2DFA. In particular, our idea is the following: given a non-deterministic planning problem and a temporal formula, we first obtain the corresponding DFA of the temporal formula through LTL$_f$2DFA, then, we encode such a DFA into the non-deterministic planning domain. As a result, we have reduced the original problem to a classic FOND planning problem. In other words, we compile extended temporal goals together with the original planning domain, specified in (PDDL), which is suitable for input to standard (FOND) planners.

4.2 Preliminaries

4.2.1 PDDL

4.2.2 Fully Observable Non Deterministic Planning

4.3 Encoding of Temporal Goals in PDDL

4.4 Implementation

4.4.1 Package Structure

4.4.2 PDDL

4.4.3 Automa

4.4.4 Main Module

4.5 Summary

Chapter 5

Janus

In this chapter, we will illustrate how our tool LTL$_f$2DFA presented in Chapter 3 can be efficiently employed in the field of Business Process Management, with particular attention to Process Mining. First of all, we will formally describe the theoretical framework of declarative process mining. We will introduce a new theorem that generalizes concepts of reactive constraints and separated formulas present in (Cecconi et al., 2018) and we will illustrate the Janus algorithm in (Cecconi et al., 2018) modified accordingly with the new proposed theorem. Then, in this context, we will thoroughly describe the implementation of this new version of the Janus algorithm, employing our tool LTL$_f$2DFA, for computing the interestingness degree of traces in real event logs. Finally, we will provide such a computation for a real log as an example.

5.1 Declarative Process Mining

In this section, we will present the theoretical framework of Business Process Management focusing our attention to declarative process mining. We will extend what described in Chapter 2 providing all additional concepts, definitions and theorems necessary to clearly understand the context.

Business Process Management (BPM) deals with discovering, modeling, analyzing and managing business processes in order to measure their productivity and to improve their performance. These tasks are carried out thanks to logging facilities that, nowadays, all BPM systems have. The extraction and the validation of temporal constraints from event logs (i.e. multi-sets of finite traces) are techniques consisting declarative process mining (Montali, 2010). Temporal constraints are expressed using LTL$_f$ and/or PLTL and refers to activities present in traces. In the following, we will formally introduce what event logs and DECLARE (Pesic, 2008) are. Another important aspect to notice is that these constraints are meant to be checked upon the activation satisfying specific conditions. For these reasons, they are referred as *reactive constraints*.

Event Logs The event log is a collection of meaningful data that is the entry point for the consequent process mining. Formally, we consider this meaningful data expressed as a multiple traces containing a sequence of events belonging to the alphabet of symbols Σ. A single trace can be represented as $t = \langle e_1, e_2, \ldots, e_n \rangle$ where e_i is the event occurring at instant i and $n \in \mathbb{N}$ is the length of the trace t. Now, we can give the following definition:

Definition 5.1. An event log $\mathcal{L}$ is defined as $\mathcal{L} = \{t_1, \ldots, t_m\} \in \mathbb{M}(\Sigma^*)$ is a multi-set of traces t_j with $1 \leq j \leq m$, where $m \in \mathbb{N}$.

To better indicate the *multiplicity* of traces in $\mathcal{L}$, we can denote it as a superscript compacting the notation. For example, t_2^{10} stands for trace t_2 occurs 10 times in $\mathcal{L}$.

Example 5.1. $\mathcal{L} = \{t_1^{25}, t_2^{10}, t_3^{15}, t_4^{20}, t_5^{5}, t_6^{10}\}$ is an event log of 85 traces, defined over the alphabet $\Sigma = \{a, b, c, \ldots, i\}$. In $\mathcal{L}$ we have the following traces:

$$
\begin{aligned}
t_1 &= \langle d, f, a, f, c, a, f, b, a, f \rangle \\
t_2 &= \langle f, e, d, c, b, a, g, h, i \rangle \\
t_3 &= \langle a, d, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, c \rangle \\
t_4 &= \langle d, b, a, b \rangle \\
t_5 &= \langle a, d, a, c, a \rangle \\
t_6 &= \langle b, c, d, e \rangle
\end{aligned}
$$

Furthermore, the event e_i occurring at instant i is denoted by $t(i)$, whereas the segment of t (i.e. the sub-trace) ranging from instant i to instant j, where $1 \leq i \leq j \leq n$ is denoted by $t_{[i:j]}$.

Apart from the formal model of event logs, we have real-world event logs that are logs with real data coming from different kind of data sources (e.g. databases, transaction logs, audit log, etc.). All available tools are evaluated against real-world logs or synthetic logs, i.e. automatically generated logs that mimic real logs in shape and content. In practice, as we will see in the Section 5.3, the main way of representing real logs is the eXtensible Event Stream (XES) Standard[1], which is based on the well known XML.

DECLARE DECLARE is a language concerning declarative process modeling (Pesic, 2008) and consisting of standard templates based on (Dwyer et al., 1999) that was introduced to simplify the complexity of constraints semantics. Indeed, DECLARE constraints are expressed in LTL$_f$, but we will extend LTL$_f$ with Past temporal operators (LTLp$_f$) for capturing also past modalities. In Table 5.1, we can see what are the corresponding LTL$_f$ or LTLp$_f$ formulas for the most important DECLARE constraints.

Parameters in a template define tasks and they occurs as events in traces. In Example 5.2 we provide a glimpse of DECLARE patterns.

Example 5.2. Interesting DECLARE templates (Maggi et al., 2013)

- PRECEDENCE(a,b) means *if b occurs then a occurs before b.*

[1] http://www.xes-standard.org

Table 5.1. The most important DECLARE constraints expressed as LTLp$_f$ formulas and *reactive constraints*.

DECLARE **constraint**	LTL$_f$ **expression**	**RCon**
PARTICIPATION(a)	$\Diamond a$	$t_{start} \mapsto \Diamond a$
INIT(a)	a	$t_{start} \mapsto a$
END(a)	$\Box\Diamond a$	$t_{end} \mapsto a$
RESPONDEDEXISTENCE(a,b)	$\Diamond a \Rightarrow \Diamond b$	$a \mapsto (\Diamond b \vee \diamondminus b)$
RESPONSE(a,b)	$\Box(a \Rightarrow \Diamond b)$	$a \mapsto \Diamond b$
ALTERNATERESPONSE(a,b)	$\Box(a \Rightarrow \Diamond b) \wedge \Box(a \Rightarrow \bigcirc(\neg a \bullet b))$	$a \mapsto \bigcirc(\neg a\, \mathcal{U}\, b)$
CHAINRESPONSE(a,b)	$\Box(a \Rightarrow \Diamond b) \wedge \Box(a \Rightarrow \bigcirc b)$	$a \mapsto \bigcirc b$
PRECEDENCE(a,b)	$\neg b \bullet a$	$b \mapsto \diamondminus a$
ALTERNATEPRECEDENCE(a,b)	$(\neg b \bullet a) \wedge \Box(a \Rightarrow \bigcirc(\neg b \bullet a))$	$b \mapsto \ominus(\neg b\, \mathcal{S}\, a)$
CHAINPRECEDENCE(a,b)	$(\neg b \bullet a) \wedge \Box(\bigcirc b \Rightarrow a)$	$b \mapsto \ominus a$

- RESPONCE(a,b) means *if a occurs then eventually b occurs after a.*
- CHAINPRECEDENCE(a,b) means *the occurrence of b imposes a to occur immediately before.*
- ALTERNATERESPONCE(a,b) means *if a occurs then eventually b occurs after a without other occurrences of a in between.*

In addition, one can create his own DECLARE patterns tailored for his purposes. In this way, the DECLARE standard template can be customized.

A given DECLARE constraint is verified over traces and those traces *satisfy* it if they do not *violate* it. Here, it is important to notice that these constraints are prone to the principle of *ex falso quod libet*, namely they can be satisfied even without being activated. This represents a big issue for process mining because mining techniques might misunderstand the actual behavior of a process. The solution to this problem is to compute whether a constraint is satisfied or not only upon activation. However, we will see later how to overcome this problem in the Section 5.2.

Now, we give some definitions:

Definition 5.2. (Gabbay, 1989) Given an LTLp$_f$ formula φ, we call it *pure past* formula ($\varphi^{\blacktriangleleft}$) if it consists of only past operators; *pure present* formula ($\varphi^{\blacktriangledown}$) if it has not any temporal operators; *pure future* formula ($\varphi^{\blacktriangleright}$) if it consists of only future operators.

Example 5.3. Pure formulas:

- $\boxminus(a \Rightarrow \diamondminus b)$ is a **pure past** formula;
- $a \Rightarrow (b \wedge c)$ is a **pure present** formula
- $\Box(a \Rightarrow \bigcirc b)$ is a **pure future** formula

The separation of an LTLp$_f$ formula to pure past/present/future formulas allows to conduct the analysis on sub-traces (i.e. one referring to the past and the other referring to the future) upon the activation. This is also known as bi-directional on-line analysis. To this extent, we rely on the Separation Theorem stated as follows:

Theorem 5.1. *(Gabbay, 1989) Any propositional temporal formula φ can be rewritten as a boolean combination of pure temporal formulas.*

Therefore, following Theorem 5.1, we can give the Definition of *separated formula* as follows:

Definition 5.3. (Cecconi et al., 2018) Let φ an LTLp$_f$ formula over Σ. A temporal separation is a function $\mathcal{S} : \text{LTLp}_f \rightarrow 2^{\text{LTLp}_f \times \text{LTLp}_f \times \text{LTLp}_f}$ such that: $\mathcal{S}(\varphi) = \{(\varphi^{\blacktriangleleft}, \varphi^{\blacktriangledown}, \varphi^{\blacktriangleright})_1, \ldots, (\varphi^{\blacktriangleleft}, \varphi^{\blacktriangledown}, \varphi^{\blacktriangleright})_m\}$ such that:

$$\varphi \equiv \bigvee_{j=1}^{m} (\varphi^{\blacktriangleleft} \wedge \varphi^{\blacktriangledown} \wedge \varphi^{\blacktriangleright})_j \tag{5.1}$$

where $\varphi^{\blacktriangleleft}$, $\varphi^{\blacktriangledown}$ and $\varphi^{\blacktriangleright}$ are pure formulas over Σ as in Definition 5.2.

Notice that Equation 5.1 is a disjunction of conjunction. Moreover, each triple consisting the image function of $\mathcal{S}(\varphi)$ is generally called *separated formula*. In the following, we give an example of separated formula.

Example 5.4. The separated formulas for $(\ominus a \vee \Diamond b)$:

$$(\ominus a \wedge True \wedge True) \bigvee (True \wedge True \wedge \Diamond b)$$

Since the Janus algorithm relies on the construction of the automata for separated LTLp$_f$ formulas, we will refer to notions explained previously in Section 2.4. The crucial point is that given a separated LTLp$_f$ formula φ we can build a minimum DFA that *accepts* all and only the traces satisfying formula φ.

In the following sections, we will describe in details our modified version of the Janus approach giving fundamentals definitions and theorems and highlighting major differences with respect to the original one in (Cecconi et al., 2018). Then, we will illustrate the modified algorithm and its practical implementation.

5.2 The Janus Approach

Declarative process modeling defines a list of DECLARE constraints to be satisfied during the execution of the process model. These constraints are of a reactive nature in the sense that the occurrence of some task bounds the occurrence of other activities. As anticipated in the previous Section, this kind of behavior might lead to the principle of *ex falso quod libet*, namely a constraint can be satisfied even though it is never activated. The Janus approach (Cecconi et al., 2018) solves this problem allowing the user to indicate the activation condition for the constraint directly in the constraint. In this way, constraints are activated only if the activation condition holds. Therefore, we can refer to these constraints as *reactive constraints* (RCon).

Definition 5.4. (Cecconi et al., 2018) Given an alphabet Σ, let $\alpha \in \Sigma$ be an *activation* and φ be an LTLp$_f$ formula over Σ. A Reactive Constraint (RCon) Ψ is a pair (α, φ), denoted as $\Psi \doteq \alpha \mapsto \varphi$. We represent all the set of RCons over Σ as $\mathcal{R}$.

Hereafter, we will assume traces, automata, LTLp$_f$ formulas and RCons to be defined over the same alphabet Σ. In addition, in Table 5.1, we can see that DECLARE constraints can be converted in RCons. In Definition 5.4, we have seen that α in an RCon is called the *activation*. Indeed, it actually *activates* the corresponding constraint. As in (Cecconi et al., 2018), we give the following definitions that are the core concepts upon which the Janus algorithm is built.

Definition 5.5. (Cecconi et al., 2018) Given a finite trace $t \in \Sigma$ of length n, and an instant i, with $1 \leq i \leq n$, an RCon $\Psi \doteq \alpha \mapsto \varphi$ is activated at i if $t, i \models \alpha$. Thus, the event $t(i)$ is called the *activator* of Ψ. A trace in which at least an activator of Ψ exists, is *triggering* for Ψ.

Definition 5.6. (Cecconi et al., 2018) Given a finite trace $t \in \Sigma$ of length n, an instant i, with $1 \leq i \leq n$, an RCon $\Psi \doteq \alpha \mapsto \varphi$, Ψ is *interesting fulfilled* at i if $t, i \models \alpha$ and $t, i \models \varphi$. The RCon is *violated* at instant i if $t, i \models \alpha$ and $t, i \not\models \varphi$. Otherwise, the RCon is unaffected.

Definition 5.6 is called *interesting fulfilment*, since it formally solves the problem of constraint satisfaction without activation by identifying only those events where the activation condition holds and the RCon is fulfilled. Therefore, every time an event is the activator of an RCon, the RCon is checked for fulfilment. After these two definitions we have to define also an empirical method to compute the *interesting fulfilment* of an RCon for an event log.

Definition 5.7. (Cecconi et al., 2018) Given a finite trace $t \in \Sigma$ of length n and an RCon $\Psi \doteq \alpha \mapsto \varphi$, we define the *interestingness degree* function $\zeta : \mathcal{R} \times \Sigma^* \to [0, 1] \subseteq \mathbb{R}$ as follows:

$$\zeta(\Psi, t) = \begin{cases} \dfrac{|\{i : t, i \models \alpha \text{ and } t, i \models \varphi\}|}{|\{i : t, i \models \alpha\}|}, & \text{if } |\{i : t, i \models \alpha\}| \neq 0; \\ 0, & \text{otherwise} \end{cases}$$

Intuitively, the $\zeta(\Psi, t)$ function measures how many times the RCon Ψ is interesting fulfilled with respect to the total number of activations within the trace t. Now, we give an example to better capture the concepts just defined.

Example 5.5. Let us consider the RCon $\Psi = b \mapsto \diamondminus a$ and traces in the Example 5.1, we have the following:

- Ψ is activated in trace t_1 by $t_1(8)$, in t_2 by $t_2(5)$, in t_4 by $t_4(2)$ and $t_4(4)$ and in t_6 by $t_6(1)$. Hence, t_1, t_2, t_4 and t_6 are *triggering* for Ψ, while Ψ is not activated in t_3 and t_5.

- Ψ is *interestingly fulfilled* by $t_1(8)$ in t_1, by only $t_4(4)$ in t_4. Moreover, Ψ is *violated* by $t_2(5)$ in t_2, by $t_4(2)$ in t_4 and by $t_6(1)$ in t_6. Finally, it is *unaffected* both in t_3 and t_5.
- The *interestingness degree* of Ψ in t_1 is $\zeta(\Psi, t_1) = 1$, since it is activated and fulfilled only once. Then, the *interestingness degree* of Ψ in t_4 is $\zeta(\Psi, t_4) = 0.5$ because it is activated twice, but fulfilled only once. Finally, in all the other traces t_2, t_3, t_5 and t_6 is $\zeta(\Psi, t) = 0$.

However, the representation of an RCon, as in Definition 5.4, has two main drawbacks:

- the activation condition α could be only a single task;
- α is not incorporated in the formula φ.

These two disadvantages limits the Janus approach to what it can mine. In order to overcome this limitation, we have devised a generalization of the RCon constraint definition. We can state the following Theorem:

Theorem 5.2. *Given an RCon $\Psi \doteq \alpha \mapsto \varphi$, expanded by Definition 5.3 as $\alpha \mapsto \bigvee_{j=1}^{m}(\varphi^{\blacktriangleleft} \wedge \varphi^{\blacktriangledown} \wedge \varphi^{\blacktriangleright})_j$, it is equivalent to:*

$$\Psi \equiv (False \wedge \alpha \wedge False) \vee \mathcal{S}'(\varphi) \tag{5.2}$$

where α is a propositional formula and $\mathcal{S}'(\varphi) \doteq \bigvee_{j=1}^{m}[\varphi^{\blacktriangleleft} \wedge (\varphi^{\blacktriangledown} \wedge \alpha) \wedge \varphi^{\blacktriangleright}]_j$.

In other words, Theorem 5.2 says that a certain activation condition (expressed as a separated formula) of RCon can be directly embedded on the constraint formula itself. This generalized representation solves disadvantages of simple RCons described above. Indeed, the first triple $(False \wedge \alpha \wedge False)$ is only needed to count all occurrences of activations whereas the constraint is fulfilled if for at least a separated formula is verified over the trace. Notice that when we check for constraint fulfillment, the triple $(False \wedge \alpha \wedge False)$ evaluates always to *false*.

In Section 5.3, we will see the implementation of the Janus algorithm for computing the *interestingness degree* of traces in real-world event logs, using RCons declared as in 5.2.

As we have just seen, the fulfilment of an RCon, in a trace, relies on the verification of the corresponding LTLp$_f$ formula over such a trace at the instant of activation. This process of verification of a formula φ on a trace can be achieved by constructing the related DFA $\mathcal{A}_\varphi$ and checking whether such trace is accepted by $\mathcal{A}_\varphi$ or not. To this extent, in the following, we have to give some other definitions and theorems.

First of all, since an LTLp$_f$ formula could have both past and future temporal operators, in order to build its corresponding DFA we exploit the Theorem 5.1 by first splitting the LTLp$_f$ formula into its separated formulas and, then, constructing the corresponding DFAs of that separated formulas. However, we need to know how to evaluate the separated formulas over a trace. We can now give the following Lemma and Theorem:

Lemma 5.3. *(Cecconi et al., 2018) Given a pure past formula $\varphi^{\blacktriangleleft}$, a pure present formula $\varphi^{\blacktriangledown}$, a pure future formula $\varphi^{\blacktriangleright}$, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, the following is holds true:*

- $t, i \models \varphi^{\blacktriangleleft} \equiv t_{[1,i]}, i \models \varphi^{\blacktriangleleft}$
- $t, i \models \varphi^{\blacktriangledown} \equiv t_{[i,i]}, i \models \varphi^{\blacktriangledown}$
- $t, i \models \varphi^{\blacktriangleright} \equiv t_{[i,n]}, i \models \varphi^{\blacktriangleright}$

The Lemma follows from the definition of the LTLp$_f$ semantics. It is trivial to see that having, at instant i, a pure past formula, its semantics only cares about events preceding i, whereas a pure future formula cares only about events following the instant i.

Theorem 5.4. *(Cecconi et al., 2018) Given an LTLp$_f$ formula φ, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, we have that $t, i \models \varphi$ iff $t_{[1,i]}, i \models \varphi^{\blacktriangleleft}, t_{[i,i]}, i \models \varphi^{\blacktriangledown}$ and $t_{[i,n]}, i \models \varphi^{\blacktriangleright}$ for* at least *a* $(\varphi^{\blacktriangleleft}, \varphi^{\blacktriangledown}, \varphi^{\blacktriangleright}) \in \mathcal{S}(\varphi)$.

The proof follows from Theorem 5.1 and Lemma 5.3.

Example 5.6. Let us consider the RCon $\Psi = a \mapsto (\ominus b \vee \Diamond c)$ with $\varphi = (\ominus b \vee \Diamond c)$, its separated formulas $\mathcal{S}(\varphi) = \{(\ominus b, True, True), (True, True, \Diamond c)\}$ and trace $t_1 = \langle d, f, a, f, c, a, f, b, a, f\rangle$ taken from Example 5.1.

- $t_1, 3 \models \varphi$ if, apart from the $True$ formulas that are satisfied, one of the following holds *true*:

  1. $\langle d, f, a\rangle, 3 \models \ominus b$
  2. $\langle a, f, c, a, f, b, a, f\rangle, 3 \models \Diamond c$

  since the latter holds *true*, φ is satisfied by $t_1(3)$.

- $t_1, 6 \models \varphi$ if, apart from the $True$ formulas that are satisfied, one of the following holds *true*:

  1. $\langle d, f, a, f, c, a\rangle, 6 \models \ominus b$
  2. $\langle a, f, b, a, f\rangle, 6 \models \Diamond c$

  since both are not satisfied, we can conclude that φ is not satisfied by $t_1(6)$.

- $t_1, 9 \models \varphi$ if, apart from the $True$ formulas that are satisfied, one of the following holds *true*:

  1. $\langle d, f, a, f, c, a, f, b, a\rangle, 9 \models \ominus b$
  2. $\langle a, f\rangle, 9 \models \Diamond c$

  since the former holds *true*, φ is satisfied by $t_1(9)$.

At this point, we can start talking about separated formulas verification on a trace using their corresponding DFAs.

Definition 5.8. Given a LTLp$_f$ formula φ, we define as *separated automata set* (sep.aut.set) $\mathcal{A}^{◀▼▶} \in 2^{\mathcal{A}\times\mathcal{A}\times\mathcal{A}}$ the set of triples $\mathcal{A}^{◀▼▶} = (\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A}\times\mathcal{A}\times\mathcal{A}$ such that $\mathcal{A}^{◀} \doteq \mathcal{A}_{\varphi^{◀}}$, $\mathcal{A}^{▼} \doteq \mathcal{A}_{(\varphi^{▼}\wedge\alpha)}$ and $\mathcal{A}^{▶} \doteq \mathcal{A}_{\varphi^{▶}}$ for every $(\varphi^{◀}, \varphi^{▼}, \varphi^{▶}) \in \Psi$.

The *sep.aut.set* just defined is a modified version of the one in (Cecconi et al., 2018). As in Example 5.4, here we give its automata version.

Example 5.7. Given the RCon $\Psi = (False \wedge a \wedge False) \vee (\ominus b \wedge (True \wedge a) \wedge True) \vee (True \wedge (True \wedge a) \wedge \Diamond c)$, its sep.aut.set is:

$$\mathcal{A}^{◀▼▶} = \{(\mathcal{A}_{False}, \mathcal{A}_a, \mathcal{A}_{False}), (\mathcal{A}_{\ominus b}, \mathcal{A}_{(True\wedge a)}, \mathcal{A}_{True}), (\mathcal{A}_{True}, \mathcal{A}_{(True\wedge a)}, \mathcal{A}_{\Diamond c})\}$$

Similarly to what we have seen before with Theorem 5.4, we can state the following:

Theorem 5.5. *(Cecconi et al., 2018) Given an* LTLp$_f$ *formula φ, its sep.aut.set $\mathcal{A}^{◀▼▶}$, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, we have that $t, i \models \varphi$ iff $t_{[1,i]}, i \in \mathcal{L}(\mathcal{A}^{◀}), t_{[i,i]}, i \in \mathcal{L}(\mathcal{A}^{▼})$ and $t_{[i,n]}, i \in \mathcal{L}(\mathcal{A}^{▶})$ for* at least *a* $(\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A}^{◀▼▶}$.

So far, we have described all theoretical results necessary for introducing and understanding how the Janus algorithm works. Now, we talk about automata generation given a pure past, pure present and a pure future formula possible thanks to our developed tool LTL$_f$2DFA.

Differently from what has been done in (Cecconi et al., 2018) for the automata construction, in this thesis we propose a version of the Janus algorithm that works with LTL$_f$2DFA. Indeed, as already seen in Chapter 3, LTL$_f$2DFA is able to directly generate the minimum DFA for a pure past formula (PLTL) without passing through its pure future (LTL$_f$) reversed formula. In particular, LTL$_f$2DFA has been employed in the Janus algorithm for the generation of the automaton corresponding to each formula in the triple $(\varphi^{◀}, (\varphi^{▼} \wedge \alpha), \varphi^{▶})$, for every triple belonging to $\mathcal{S}'(\varphi)$. In Example 5.8, there are DFAs output from LTL$_f$2DFA.

Example 5.8. Let us consider the RCon $\Psi = (False \wedge a \wedge False) \vee (\ominus b \wedge (True \wedge a) \wedge True) \vee (True \wedge (True \wedge a) \wedge \Diamond c)$. The corresponding sep.aut.set $\mathcal{A}^{◀▼▶}$ for each triple in Ψ is depicted in Table 5.2.

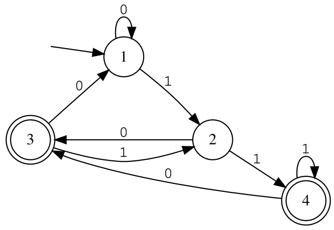
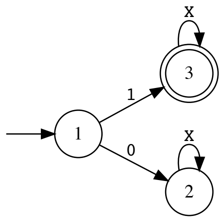
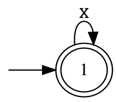
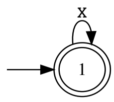
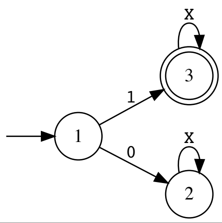
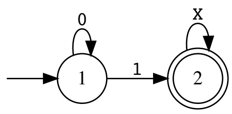
5.2.1 Algorithm

Here, we illustrate a modified version of the Janus algorithm present in (Cecconi et al., 2018), which is able to deal with our new generalization of RCons introduced previously with Theorem 5.2.

Algorithm 5.1. JANUS algorithm: given a a trace t, an RCon and its sep.aut.set $\mathcal{A}^{\blacktriangleleft\blacktriangledown\blacktriangleright}$, it returns the *interestingness degree*

$\mathcal{O} \leftarrow$ empty bag;
foreach event $t(i) \in t$ **do**
  foreach $\mathcal{A}^{\blacktriangleleft} \in \mathcal{A}^{\blacktriangleleft\blacktriangledown\blacktriangleright}$ **do** make transition $t(i)$ on $\mathcal{A}^{\blacktriangleleft}$;
  end for
  if ($\mathcal{A}_\alpha$ *accepts* $t(i)$) **then** ▷ Activation
    $\mathcal{J} \leftarrow$ empty set of pairs;
    foreach $(\mathcal{A}^{\blacktriangleleft}, \mathcal{A}^{\blacktriangledown}, \mathcal{A}^{\blacktriangleright}) \in \mathcal{A}^{\blacktriangleleft\blacktriangledown\blacktriangleright}$ **do**
      if ($\mathcal{A}^{\blacktriangleleft}$ *is in an accepting state and* $\mathcal{A}^{\blacktriangledown}$ *accepts* $t(i)$) **then**
        Add $(s_0^{\blacktriangleright}, \mathcal{A}^{\blacktriangleright})$ to $\mathcal{J}$;
      end if
    end for
    Add $\mathcal{J}$ to $\mathcal{O}$;
  end if
  foreach $\mathcal{J} \in \mathcal{O}$ **do**
    foreach $(s^{\blacktriangleright}, \mathcal{A}^{\blacktriangleright}) \in \mathcal{J}$ **do** $s^{\blacktriangleright} \leftarrow \delta^{\blacktriangleright}(s^{\blacktriangleright}, t(i))$ ▷ Make transition $t(i)$ on $\mathcal{A}^{\blacktriangleright}$
    end for
  end for
end for
if $\mid \mathcal{O} \mid > 0$ **then**
  return $\dfrac{\mid \{\mathcal{J} \in \mathcal{O} : \text{at least a } (s^{\blacktriangleright}, \mathcal{A}^{\blacktriangleright}) \in \mathcal{J}\} \text{ is such that } s^{\blacktriangleright} \in \mathcal{F}^{\blacktriangleright} \mid}{\mid \mathcal{O} \mid}$
else
  return 0
end if

Table 5.2. Representation of the separated automata set for $\Psi = a \mapsto (\ominus b \vee \Diamond c)$

$\mathcal{A}^{\blacktriangleleft} = \mathcal{A}_{\ominus b}$	$\mathcal{A}^{\blacktriangledown} = \mathcal{A}_{(true \wedge a)}$	$\mathcal{A}^{\blacktriangleright} = \mathcal{A}_{true}$
		
$\mathcal{A}^{\blacktriangleleft} = \mathcal{A}_{true}$	$\mathcal{A}^{\blacktriangledown} = \mathcal{A}_{(true \wedge a)}$	$\mathcal{A}^{\blacktriangleright} = \mathcal{A}_{\Diamond c}$
		

5.3 Implementation

In this section, we fully describe the practical implementation of the Janus algorithm given in Section 5.2.1. In particular, we give some general information about its features, dependencies and usage. Then, we focus on the package explaining how is structured and commenting highlights on the code. The implementation on this thesis is called JANUS, is written in Python and is a porting of the Janus proof-of-concept software project written in Java.

To begin with, the main goal of JANUS, as stated at the beginning, is to compute the *interestingness degree* of traces on event log. As a consequence, it also provides I/O facilities for three different event log formats, namely simple *.txt* files, *.csv* files and *.xes* files for real-world event logs. Furthermore, the formula φ in the RCon to be satisfied by traces is manually separated following Definition 5.3.

JANUS requires Python>=3.6 and has the following dependencies:

- LTL$_f$2DFA, presented in Chapter 3. As stated before, it has been used for the generation of DFAs;
- OpyenXES, an open-source complete Python library for the XES Standard published in (Valdivieso et al., 2018). It has been used for dealing with XES parsing and management.

The JANUS software is an open-source project and available to download on GitHub[2].

[2]https://github.com/Francesco17/janus

5.3.1 Package Structure

The structure of the JANUS package is relatively simple. It consists of the following:

- `janus.py`: it is the main module of the package. It contains the actual implementation of the Janus algorithm.
- `janus/`: it is the directory containing all the necessary code to correctly implement the algorithm. It has three subfolders:
  - `io/`: it contains the `InputHandler.py` which is in charge of handling the event log given as input.
  - `automata/`: it consists of the `automa.py` file, the `parserAutoma.py` file and the `sepautset.py` file. In this folder, we find all the code for dealing with automata.
  - `formulas/`: it comprises the `formula.py` file and the `separatedFormula.py`. These files defines the logic for LTLp$_f$ formulas and RCons.
- `files/`: this folder is the place where there are event logs. From this folder, a specific event log is parsed.

5.3.2 I/O

The `InputHandler.py` file, included in the `io/` folder, has been developed separately from the main module since we wanted to use our algorithm regardless of the input file format. In particular, thanks to the relative `InputHandler` class (Listing 5.1), the tool can import a log from a simple text file, from a csv and, finally, from a XES file. Hence, the JANUS tool can be used not only with the XES format, but also with other more manageable file formats.

Listing 5.1. The `InputHandler.py` module

```python
import csv
from opyenxes.data_in.XUniversalParser import XUniversalParser
from opyenxes.classification.XEventAttributeClassifier import \
XEventAttributeClassifier

class InputHandler:

    def __init__(self, input_path):
        self.input_path = input_path
        self._event_log = None
        self.load()

    @property
    def event_log(self):
```

```
        return self._event_log

    def load_txt(self):
        try:
            with open(self.input_path, 'r') as f:
                self._event_log = set(tuple(i) for i in \
                [f.read().splitlines()])
                f.close()
        except:
            raise IOError('[ERROR]:␣Unable␣to␣import␣text␣file')

    def load_csv(self):
        self._event_log = []
        try:
            with open(self.input_path, newline='', encoding='utf-8-sig') \
            as f:
                reader = csv.reader(f)
                for row in reader:
                    self._event_log.append(row[0])
        except:
            raise IOError('[ERROR]:␣Unable␣to␣import␣csv␣file')

    def load_xes(self):
        try:
            with open(self.input_path) as log_file:
                log = XUniversalParser().parse(log_file)[0]

            # get classifiers
            classifiers = []
            for cl in log.get_classifiers():
                classifiers.append(str(cl))

            classifier = XEventAttributeClassifier("activity", \
            [classifiers[0]])
            log_list = list(map(lambda trace: \
            (map(classifier.get_class_identity, trace)), log))

            self._event_log = set(tuple(trace) for trace in log_list)

        except:
            raise IOError('[ERROR]:␣Unable␣to␣import␣xes␣file')

    def load(self):
```

```
58         if self.input_path.endswith('.txt'):
59             self.load_txt()
60         elif self.input_path.endswith('.csv'):
61             self.load_csv()
62         elif self.input_path.endswith('.xes'):
63             self.load_xes()
64         else:
65             raise ValueError('[ERROR]:␣File␣extension␣not␣recognized')
```

From Listing 5.1, we can see that the `InputHandler` class has a main method called `load` that depending on the format of the file given as input calls the corresponding method specific for that format. If the format is not among *.txt*, *.csv* and *.xes*, it raises an error. Consequently, every specific method parses the event log. In particular, at line 37, the `load_xes` method takes advantage of the OpyenXES library APIs using its parser and classifier. In Section 5.3.5, we will look at how an `InputHandler` object can be instantiated.

5.3.3 Automata

In the `automata/` folder there are files devoted to handle and manage automata. Firstly, the `parserAutoma.py` module is a collection of functions used for parsing the *.dot* file and instantiating the data structure representing the automaton. In Listing 5.2 is shown that collection of functions.

Listing 5.2. The `parserAutoma.py` module

```
import pydot
from janus.automata.automa import Automa

def get_file(path):
    try:
        with open(path, 'r') as file:
            lines = file.readlines()
            file.close()
        return lines
    except IOError:
        print('[ERROR]:␣Not␣able␣to␣open␣the␣file␣from␣{}'.format(path))

def get_graph_from_dot(path):
    try:
        dot_graph = pydot.graph_from_dot_file(path)
        return dot_graph[0]
    except IOError:
        print('[ERROR]:␣Not␣able␣to␣import␣the␣dot␣file')

```

```
def get_final_label(label):

    s1 = label.replace("␣", "")
    s2 = s1.replace('"','')

    if s2 == '':
        return ['X']
    elif len(s2) < 2:
        return [s2]
    else:
        s3 = s2.replace(",","")
        s4 = s3.split('\\n')

        leng_elem = len(s4[0])
        temp = ''
        inter_label = []
        for i in range(leng_elem):
            for elem in s4:
                temp += elem[i]
            inter_label.append(temp)
            temp = ''

        return inter_label

def parse_dot(path, symbols):

    graph = get_graph_from_dot(path)

    nodes = []
    for node in graph.get_nodes():
        if node.get_name().isdigit():
            nodes.append(node.get_name())
        else: continue

    states = set(nodes)
    initial_state = sorted(nodes, key=int)[0]

    lines = get_file(path)
    accepting_states = set() # all accepting states of the automaton
    for line in lines[7:]:
        if line.strip() != 'node␣[shape=circle];':
            temp = line.replace(";\n", "")
            accepting_states.add(temp.strip())
```

```
63          else:
64              break
65
66      sources = []
67      for elem in graph.get_edges():
68          if elem.get_source().isdigit():
69              sources.append(elem.get_source())
70          else: continue
71
72      i = 0
73      transitions = dict()
74      for source in sources:
75          label = graph.get_edges()[i].get_label()
76          final_label = get_final_label(label)
77          destination = graph.get_edges()[i].get_destination()
78          i += 1
79          for lab in final_label:
80              if source in transitions:
81                  transitions[source][lab] = destination
82              else:
83                  transitions[source] = dict({lab: destination})
84
85      #instantiation of automaton
86      automaton = Automa(
87          symbols=symbols,
88          alphabet={'0', '1', 'X'},
89          states=states,
90          initial_state=initial_state,
91          accepting_states=accepting_states,
92          transitions=transitions
93      )
94      return automaton
```

The most important function is called `parse_dot` (at line 44). It works as follows: given the path of a *.dot* file (the output of LTL$_f$2DFA) and symbols used in the formula, it returns an instantiation of the `Automa` class retrieving all information about the DFA, namely all its states, the initial state, accepting states and, finally, its transitions.

Afterwards, there is the `automa.py` in which the `Automa` class is implemented. This class is the data structure representing the DFA that is output from our tool LTL$_f$2DFA. It follows that in the `Automa` class there are methods able to perform transitions over the DFA, to tell whether if the automaton is in an accepting state or not and to tell whether an input symbols can be read by the automaton or not. In addition, when an object is instantiated, it is checked to be a valid DFA. In Listing 5.3 the `Automa` class implementation is shown.

Listing 5.3. The `automa.py` module

```python
import re

class Automa:
    """
        DFA Automa:
        - symbols         => list() ;
        - alphabet        => set() ;
        - states          => set() ;
        - initial_state   => str() ;
        - accepting_states => set() ;
        - transitions     => dict(), where
        **key**: *source* in states
        **value**: {*action*: *destination*)
    """

    def __init__(self, symbols, alphabet, states, initial_state, \
    accepting_states, transitions):
        self.symbols = symbols
        self.alphabet = alphabet
        self.states = states
        self._initial_state = initial_state
        self.accepting_states = accepting_states
        self.transitions = transitions
        self._current_state = self._initial_state
        self.validate()

    def valide_transition_start_states(self):
        for state in self.states:
            if state not in self.transitions:
                raise ValueError(
                    'transition start state {} is missing'.format(
                        state))

    def validate_initial_state(self):
        if self._initial_state not in self.states:
            raise ValueError('initial state is not defined as state')

    def validate_accepting_states(self):
        if any(not s in self.states for s in self.accepting_states):
            raise ValueError('accepting states not defined as state')

    def validate_input_symbols(self):
```

```
        alphabet_pattern = self.get_alphabet_pattern()
        for state in self.states:
            for action in self.transitions[state]:
                if not re.match(alphabet_pattern, action):
                    raise ValueError('invalid transition found')

    def get_alphabet_pattern(self):
        return re.compile("(^["+''.join(self.alphabet)+"]+$)")

    def validate(self):
        self.validate_initial_state()
        self.validate_accepting_states()
        self.valide_transition_start_states()
        self.validate_input_symbols()
        return True

    ...

    @property
    def current_state(self):
        return self._current_state

    @property
    def initial_state(self):
        return self._initial_state

    def make_transition(self, action):
        if action in self.symbols:
            for act in self.transitions[self._current_state].keys():
                temp = dict(zip(self.symbols,[value for value in act]))
                additional = temp.copy()
                del additional[action]
                if (temp[action] == '1' or temp[action] == 'X') and \
                all(value in {'0', 'X'} for value in additional.values()):
                    self._current_state = \
                    self.transitions[self._current_state][act]
                else:
                    continue
        else:
            number_of_symbols = len(self.symbols)
            if number_of_symbols == 0: # true when there is True automa
                self._current_state = \
                self.transitions[self._current_state]['X']
```

```
86              else:
87                  if 'X'*number_of_symbols in \
88                  self.transitions[self._current_state]:
89                      self._current_state = \
90                      self.transitions[self._current_state]\
91                      ['X'*number_of_symbols]
92                  elif '0'*number_of_symbols in \
93                  self.transitions[self._current_state]:
94                      self._current_state = \
95                      self.transitions[self._current_state]\
96                      ['0'*number_of_symbols]
97                  else:
98                      raise ValueError('[ERROR]:␣could␣not␣make␣transition')
99
100     def is_accepting(self):
101         if self._current_state in self.accepting_states:
102             return True
103         else:
104             return False
105
106     def accepts(self, input_symbol):
107         _current_state = self._current_state
108         self._current_state = self._initial_state
109         self.make_transition(input_symbol)
110         if self.is_accepting():
111             self._current_state = _current_state
112             return True
113         else:
114             self._current_state = _current_state
115             return False
```

Once an `Automa` object is instantiated, the method `validate` (line 52) checks whether the object is a valid DFA or not. In particular, it checks if the initial state and final states are actually states and it verifies that transitions are not made by invalid symbols. Then, the `make_transition` method at line 69 takes as input an action and make this action on the automaton, therefore modifying its current state. After that, the `is_accepting` method (line 100) simply tells whether the current state is accepting for the automaton itself or not. Finally, there is the `accepts` method at line 106, which given an input symbol returns *true* if it is accepted by the DFA.

As last module about automata, we illustrate the `sepautset.py`. It is the direct implementation of the *sep.aut.set* defined in 5.8. Indeed, this module contains the definition of the `SeparatedAutomataSet` class, namely the data structure that allow us to generate the corresponding *sep.aut.set*. Hence, it takes care of generating the set of separated automata starting from a list of separated formulas. We can see its implementation in Listing 5.4.

Listing 5.4. The `sepautset.py` module

```
from ltlf2dfa.Translator import Translator
from ltlf2dfa.DotHandler import DotHandler
from janus.automata.parserAutoma import parse_dot
import os, re

class SeparatedAutomataSet:

    def __init__(self, separated_formulas_set):
        self.separated_formulas_set = separated_formulas_set
        self._automa_set = self.compute_automa()

    @property
    def automa_set(self):
        return self._automa_set

    def build_automaton(self, triple):
        automata_list = []
        for formula in triple:
            symbols = re.findall('[a-z]+', str(formula))
            trans = Translator(formula)
            trans.formula_parser()
            trans.translate()
            trans.createMonafile(True) # true for DECLARE assumptions
            trans.invoke_mona("automa.mona") # returns inter-automa.dot
            dot = DotHandler("inter-automa.dot")
            dot.modify_dot()
            dot.output_dot() # returns automa.dot
            automata_list.append(parse_dot("automa.dot", symbols))
            os.remove("automa.mona")
            os.remove("automa.dot")
            symbols = []
        return automata_list

    def compute_automa(self):
        result = set()
        for triple in self.separated_formulas_set:
            past, present, future = self.build_automaton(triple)
            result.add( (past, present, future) )
        return result
```

In this module, it has been employed the $\text{LTL}_f\text{2DFA}$ tool. In fact, a `SeparatedAutomataSet` object receives a given set of separated formulas as input and it uses LTL$_f$2DFA to generate the DFA corresponding to each formula. Specifically, for each separated formula, i.e. a triple, (line 36) we compute the equivalent DFA on-line (line 16). This specific aspect represents a novelty with respect to what has been done in the original Java version of JANUS. So, unlike our JANUS version, the original one does not compute DFAs at execution time, but they are predefined (it only supports the most important DECLARE constraints) and given before the actual execution. Thus, this is a big step towards a complete generalization of the Janus algorithm implementation.

5.3.4 Formulas

Strictly connected to what we have just talked about in the previous section, in the `formulas/` folder there are modules in which we have defined the logic for managing and representing separated formulas and the formula constraint. In particular, we have the `separatedFormula.py` which comprises the `SeparatedFormula` class. Such class has the task of representing each triple resulting from the temporal separation (Definition 5.3). We show its implementation in Listing 5.5

Listing 5.5. The `separatedFormula.py` module

```python
class SeparatedFormula:

    def __init__(self, triple):
        self.triple = triple
        self.validate()

    def validate(self):
        if len(self.triple) == 3:
            return True
        else:
            raise ValueError('[ERROR]: input is not a triple')

    def __str__(self):
        return str(self.triple)

    def __iter__(self):
        for elem in self.triple:
            yield elem
```

When instantiating a `SeparatedFormula` object, this is validated checking whether the given triple is valid.

After that, the other module contained in the `formulas/` folder is the `Formula.py` in which it is defined the `Formula` class. This class represents the formula of the RCon to be satisfied by traces. Actually, since the JANUS package works with already separated formulas, the `Formula` class gets a list of separated formulas, namely a list of triples.

The implementation (Listing 5.6) of this class is quite similar to the `SeparatedFormula` class seen before.

Listing 5.6. The `Formula.py` module

```
from janus.formulas.separatedFormula import SeparatedFormula

class Formula:

    def __init__(self, separatedFormulas):
        self.separatedFormulas = separatedFormulas
        self.validate()

    def validate(self):
        if all(isinstance(x, SeparatedFormula) for x in \
        self.separatedFormulas) and self.separatedFormulas:
            return True
        else:
            raise ValueError('[ERROR]: Different types for conjuncts')

    def __str__(self):
        return ', '.join(map(str, self.separatedFormulas))

    def __iter__(self):
        for triple in self.separatedFormulas:
            yield triple
```

5.3.5 Main Module

Here we describe the main module of the JANUS package. It is called `janus.py` and it contains the principal logic covering the Janus pseudocode anticipated in Section 5.2.1. We recall that in this version of JANUS we can specify any type of LTLp$_f$ formula as long as it is already separated following Definition 5.3.

5.3.6 Results

After having illustrated the whole implementation of JANUS, we are ready to present results of its execution where we have evaluated our tool against a real-world event log. Hence, we have analyzed the real-world event log called *Sepsis*[3], which reports the trajectories of patients showing symptoms of sepsis in a Dutch hospital.

5.4 Summary

[3] https://doi.org/10.4121/uuid:915d2bfb- 7e84- 49ad- a286- dc35f063a460

Chapter 6

Conclusions and Future Work

Continue the introduction and possible future work

6.1 Overview

6.2 Main Contributions

6.3 Future Works

6.4 Final Remarks

Bibliography

Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *International Workshop on Implementing Automata*, pages 6–25. Springer, 1996.

Alberto Camacho, Eleni Triantafillou, Christian J Muise, Jorge A Baier, and Sheila A McIlraith. Non-deterministic planning with temporally extended goals: Ltl over finite and infinite traces. In *AAAI*, pages 3716–3724, 2017.

Alessio Cecconi, Claudio Di Ciccio, Giuseppe De Giacomo, and Jan Mendling. Interestingness of traces in declarative process mining: The janus ltlpf approach. In *International Conference on Business Process Management*, pages 121–138. Springer, 2018.

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, volume 13, pages 854–860, 2013.

Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for ltl and ldl on finite traces. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1558–1564. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL http://dl.acm.org/citation.cfm?id=2832415.2832466.

Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on ltl on finite traces: Insensitivity to infiniteness. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 1027–1033. AAAI Press, 2014. URL http://dl.acm.org/citation.cfm?id=2893873.2894033.

Patrick Doherty and Jonas Kvarnstram. Talplanner: A temporal logic-based planner. *AI Magazine*, 22(3):95, 2001.

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*, pages 411–420. ACM, 1999.

Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, June/July 1998.

D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. Technical report, Jerusalem, Israel, Israel, 1997.

Dov Gabbay. The declarative past and imperative future. In *Temporal logic in specification*, pages 409–448. Springer, 1989.

Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1980.

Valentin Goranko and Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2015 edition, 2015.

Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 89–110. Springer, 1995.

Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California Los Angeles, 1968. URL http://www.ims.uni-stuttgart.de/archiv/kamp/files/1968.kamp.thesis.pdf. Published as Johan Anthony Willem Kamp.

Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from http://www.brics.dk/mona/. Revision of BRICS NS-98-3.

Fabrizio M Maggi, RP Jagadeesh Chandra Bose, and Wil MP van der Aalst. A knowledge-based integrated approach for discovering and repairing declare maps. In *International Conference on Advanced Information Systems Engineering*, pages 433–448. Springer, 2013.

Nicolas Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79:122–128, 2003.

Marco Montali. Declarative process mining. In *Specification and verification of declarative open interaction models*, pages 343–365. Springer, 2010.

Maja Pesic. Constraint-based workflow management systems: shifting control to users. 2008.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32.

M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959. ISSN 0018-8646. doi: 10.1147/rd.32.0114. URL http://dx.doi.org/10.1147/rd.32.0114.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3837. URL http://doi.acm.org/10.1145/3828.3837.

Hernan Valdivieso, Wai Lam Jonathan Lee, Jorge Munoz-Gama, and Marcos Sepúlveda. Opyenxes: A complete python library for the extensible event stream standard. In *Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018.*, pages 71–75, 2018. URL http://ceur-ws.org/Vol-2196/BPM_2018_paper_15.pdf.

Shufang Zhu, Pu Geguang, and Moshe Y. Vardi. First-order vs. second-order for ltlf-to-automata: An extended abstract. Women in Logic, 2018.