Sapienza
Università di Roma

Title

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

Title

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

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:

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

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

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

$$\pi, i \models \circ\varphi \text{ iff } \pi, i+1 \models \varphi$$

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

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 Complexity

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, formulas of LTL$_f$ 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:

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

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

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

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

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

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

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 $\Diamond(Last \wedge \varphi)$ 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 $\Diamond(Last \wedge \varphi)$ 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 $length(\pi)$. We denote the *positions*, i.e. instants, on the trace as $\pi(i)$ with $0 \leq i \leq last$ where $last = length(\pi) - 1$ 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 $0 \leq i \leq j \leq last$. 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 Definiiton 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 Complexity

About LTL_f complexity, we can state the following theorem:

Theorem 2.2 (De Giacomo and Vardi (2013))**.** *Satisfiability, validity and entailment 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)

2.4 LTL_f2DFA

talk about theory behind conversion to automata in future

2.5 PLTL2DFA

talk about theory behind conversion to automata in past

2.6 LTL$_f$2FOL and MONA

talk about theory behind translation and intro with mona future

2.7 PLTL2FOL and MONA

talk about theory behind translation and intro with mona past

Chapter 3

LTL_f2DFA

In this chapter we will present LTL_f2DFA, a software package written in Python.

3.1 Introduction

LTL_f2DFA is a Python tool that processes a given LTL_f formula (with past and future operators) 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 formula with past or future operators;
- translation of an LTL_f formula to MONA program;
- conversion of an LTL_f formula to DFA automaton.

LTL_f2DFA 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_f2DFA 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-z]+'
        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])))
            elif p[1] == 'W':
                p[0] = ('~', ('X', ('~', p[2])))
            else:
```

```python
                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]))
            elif p[2] == 'R':
                p[0] = ('~', ('U', ('~', p[1]), ('~', p[3])))
            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.

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

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

```python
from ltlf2dfa.Parser import MyParser

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

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

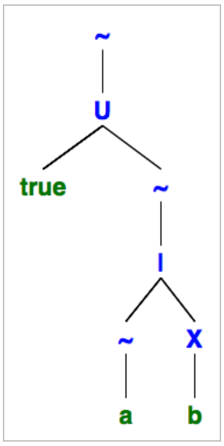


Figure 3.1. The syntax tree generated for the formula ”$G(a \sim Xb)$”. Symbols are in green while operators are in blue.

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$ formula into a 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$_f$ formula to the MONA syntax is done passing through FOL as shown in 3.1.

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

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

The former translation from LTL$_f$ to FOL is done accordingly to (De Giacomo and Vardi, 2013), while the latter follows from (Klarlund and Møller, 2001). In Listing 3.4 we can see the translation's implementation. Three dots ... represent omitted code.

Listing 3.4. The `translate` method.

```python
import ...

class Translator:
    ...

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

    ...

def translate_bis(formula_tree, var):
    if type(formula_tree) == tuple:
        #enable this print to see the tree pruning
        # print(self.parsed_formula)
        # print(var)
        if formula_tree[0] == '&':
            # print('computed tree: '+ str(self.parsed_formula))
            if var == 'v_0':
                a = translate_bis(formula_tree[1], '0')
                b = translate_bis(formula_tree[2], '0')
            else:
                a = translate_bis(formula_tree[1], var)
                b = translate_bis(formula_tree[2], 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] == '|':
            # print('computed tree: '+ str(self.parsed_formula))
            if var == 'v_0':
                a = translate_bis(formula_tree[1], '0')
                b = translate_bis(formula_tree[2], '0')
            else:
                a = translate_bis(formula_tree[1], var)
                b = translate_bis(formula_tree[2], 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] == '~':
            # print('computed tree: '+ str(self.parsed_formula))
            if var == 'v_0': a = translate_bis(formula_tree[1], '0')
            else: a = translate_bis(formula_tree[1], var)
            if a == 'true': return 'false'
            elif a == 'false': return 'true'
            else: return '~('+ a +')'
        elif formula_tree[0] == 'X':
            # print('computed tree: '+ str(self.parsed_formula))
            new_var = _next(var)
            a = translate_bis(formula_tree[1],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':
            # print('computed tree: '+ str(self.parsed_formula))
            new_var = _next(var)
            new_new_var = _next(new_var)
            a = translate_bis(formula_tree[2],new_var)
            b = translate_bis(formula_tree[1],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($)␣&␣'+ a + \
                '␣&␣all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
        elif formula_tree[0] == 'Y':
            # print('computed tree: '+ str(self.parsed_formula))
            new_var = _next(var)
            a = translate_bis(formula_tree[1],new_var)
            if var == 'v_0':
                return '('+ 'ex1␣'+new_var+':␣'+ new_var + \
                '␣=␣max($)␣-␣1␣'+ '&␣max($)␣>␣0␣&␣'+ a +')'
            else:
                return '('+ 'ex1␣'+new_var+':␣'+ new_var + \
                '␣=␣'+ var + '␣-␣1␣'+ '&␣'+new_var+'␣>␣0␣&␣'+ a +')'
        elif formula_tree[0] == 'S':
            # print('computed tree: '+ str(self.parsed_formula))
            new_var = _next(var)
            new_new_var = _next(new_var)
            a = translate_bis(formula_tree[2],new_var)
            b = translate_bis(formula_tree[1],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+':␣'+new_var+'␣<␣'+ \
                new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+b+'␣)'
                elif a == 'false': return 'false'
                else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a + \
                '␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
                new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+b+'␣)'
            else:
```

```
126                 if b == 'true': return '( '+ 'ex1 '+new_var+ \
127                 ': 0 <= '+new_var+' & '+new_var+' <= max($) & '+ a +' )'
128                 elif a == 'true': return '( '+ 'ex1 '+new_var+ \
129                 ': 0 <= '+new_var+' & '+new_var+' <= '+var+ \
130                 ' & all1 '+new_new_var+': '+new_var+' < '+ \
131                 new_new_var+' & '+new_new_var+' <= '+var+' => '+b+' )'
132                 elif a == 'false': return 'false'
133                 else: return '( '+ 'ex1 '+new_var+': 0 <= '+ \
134                 new_var+' & '+new_var+' <= '+var+' & '+ a +' & all1 '+ \
135                 new_new_var+': '+new_var+' < '+new_new_var+' & '+ \
136                 new_new_var+' <= '+var+' => '+b+' )'
137     else:
138         # handling non-tuple cases
139         if formula_tree == 'true': return 'true'
140         elif formula_tree == 'true': return 'false'
141 
142         # enable if you want to see recursion
143         # print('computed tree: '+ str(self.parsed_formula))
144 
145         # BASE CASE OF RECURSION
146         else:
147             if formula_tree.isalpha():
148                 if var == 'v_0':
149                     return '0 in '+ formula_tree.upper()
150                 else:
151                     return var + ' in ' + formula_tree.upper()
152             else:
153                 return var + ' in ' + formula_tree
154 
155 def _next(var):
156     if var == '0': return 'v_1'
157     else:
158         s = var.split('_')
159         s[1] = str(int(s[1])+1)
160         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.

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

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
```

```
20         else:
21             return None
22 
23     def buildMonaProgram(self, flag_for_declare):
24         if not self.alphabet and not self.translated_formula:
25             raise ValueError
26         else:
27             if flag_for_declare:
28                 if self.compute_declare_assumption() is None:
29                     if self.alphabet:
30                         return self.headerMona + \
31                         'var2␣' + ",␣".join(self.alphabet) + ';\n' + \
32                          self.translated_formula
33                     else:
34                         return self.headerMona + self.translated_formula
35                 else: return self.headerMona + 'var2␣' +\
36                  ",␣".join(self.alphabet) + ';\n' + \
37                  self.translated_formula + \
38                  self.compute_declare_assumption()
39             else:
40                 if self.alphabet:
41                     return self.headerMona + 'var2␣' +\
42                      ",␣".join(self.alphabet) + ';\n' + \
43                      self.translated_formula
44                 else:
45                     return self.headerMona + self.translated_formula
46 
47     def createMonafile(self, flag):
48         program = self.buildMonaProgram(flag)
49         try:
50             with open('./automa.mona', 'w+') as file:
51                 file.write(program)
52                 file.close()
53         except IOError:
54             print('Problem␣with␣the␣opening␣of␣the␣file!')
55 ...
```

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`.

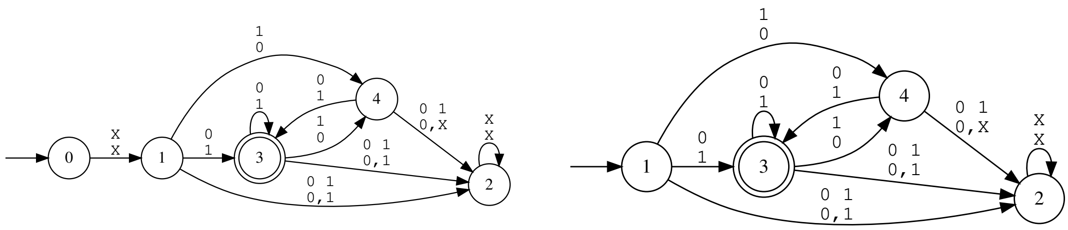
Listing 3.7. The `DotHandler` class.

```
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')
            self.new_digraph = graph
        else:
            print('[ERROR]␣-␣No␣file␣DOT␣exists')
            exit()

    def delete_intermediate_automaton(self):
        if os.path.isfile(self.dot_path):
            os.remove(self.dot_path)
```

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

```
31            return True
32        else:
33            return False
34
35    def output_dot(self, result_path='./automa.dot'):
36        try:
37            if self.delete_intermediate_automaton():
38                with open(result_path, 'w+') as f:
39                    f.write(str(self.new_digraph))
40                    f.close()
41            else:
42                raise IOError('[ERROR] - Something wrong occurred in '+ \
43                'the elimination of intermediate automaton.')
44        except IOError:
45            print('[ERROR] - Problem with the opening of the file %s!' \
46            %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 35 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 $\varphi = \Box(a \rightarrow \bigcirc b)$.



(a) The DFA output from MONA

(b) The DFA post-processed

Figure 3.2. Before and after DFA post-processing

3.3 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$_f$/LDL$_f$ formulas, but not PLTL ones (i.e.

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

LTL$_f$ formulas with past temporal operators), but it provides support for syntax, semantics and parsing of PL, LTL$_f$ and LDL$_f$ formal languages. Additionally, its conversion is based on a different theoretical result with respect to LTL$_f$2DFA. Nevertheless, we can compare them on the generation of a DFA from an LTL$_f$ formula.

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 an 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 3.3). In the

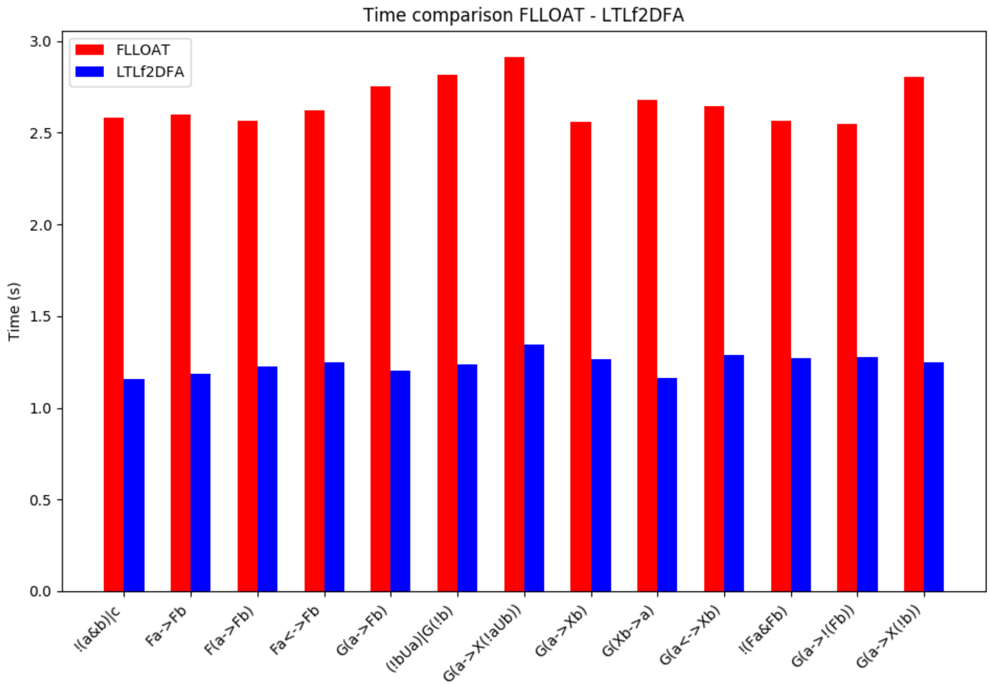


Figure 3.3. Time benchmarking of LTL$_f$2DFA wrt FLLOAT.

histogram, FLLOAT results are coloured in red, while LTL$_f$2DFA ones are depicted in blue. As we can see from the bar chart, in both packages the time needed to convert the formula increases as the length of the formula grows. However, it is notable that LTL$_f$2DFA is overall $2x$ faster that 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$2DFA uses, for the heavy task of the generation of the automaton, MONA that is written in C++. Hence, the real difference relies on the performance differences between C++ and Python programs. As a final remark, although LTL$_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.

3.4 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

Janus

Chapter 5

DFAgame

Chapter 6

Conclusions and Future Work

Continue the introduction and possible future work

Bibliography

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, 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.

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.

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.

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.

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.

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.