



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

© 2018 Francesco Fuggitti. All rights reserved

This thesis has been typeset by L^AT_EX 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
2.2	Linear Temporal Logic on Finite Traces (LTL_f)	6
2.3	Past Linear Temporal Logic (PLTL)	6
2.4	LTL_f2DFA	6
2.5	PLTL2DFA	6
2.6	LTL_f2FOL and MONA	6
2.7	PLTL2FOL and MONA	6
3	LTL_f2DFA	7
3.1	Introduction	7
3.2	Package Structure	8
3.2.1	Lexer.py	8
3.2.2	Parser.py	10
3.2.3	Translator.py	13
3.2.4	DotHandler.py	20
3.3	Comparison with FLLOAT	22
3.4	Discussion	24
4	Janus	25
5	DFAgame	27
6	Conclusions and Future Work	29

Bibliography	31
--------------	----

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. Hence, this kind of formal languages are able to specify properties about how a system changes over time.

In this scenario, we find the *Linear Temporal Logic* (LTL) (Pnueli, 1977) which is a modal temporal logic with modalities referring to time. LTL is a very well known temporal logic since it has been extensively used in AI and CS. 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:

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

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 \perp to denote *true* and *false* respectively. Moreover, all classical logic operators $\vee, \Rightarrow, \Leftrightarrow$ 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 $\Diamond\varphi \doteq \text{true} \mathcal{U} \varphi$, *always* as $\Box\varphi \doteq \neg\Diamond\neg\varphi$, *weak-next* as $\mathcal{W} \doteq \neg\bigcirc\neg\varphi$ 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*: $\Box\neg\varphi$, which means "it is always true that property in φ will never happen" or "something bad will not happen". For instance, $\Box\neg(\text{reactor} - \text{temp} > 1000)$ (the temperature of the reactor must never be over 1000).
- *Liveness*: $\Diamond\varphi$, which means "sooner or later φ will hold" or "something good will happen". For instance, $\Diamond\text{rich}$ (eventually I will become rich).
- *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, $\Box\Diamond\text{ready} \Rightarrow \Box\Diamond\text{run}$ (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:

$$\begin{aligned} \pi, i &\models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i) \\ \pi, i &\models \neg\varphi \text{ iff } \pi, i \not\models \varphi \\ \pi, i &\models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2 \\ \pi, i &\models \bigcirc\varphi \text{ iff } \pi, i+1 \models \varphi \\ \pi, i &\models \varphi_1 \mathcal{U} \varphi_2 \text{ iff } \exists j. (j \geq i) \wedge \pi, j \models \varphi_2 \wedge \forall k. (i \leq k < j) \Rightarrow \pi, k \models \varphi_1 \end{aligned}$$

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.

Finally, 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)

2.3 Past Linear Temporal Logic (PLTL)

2.4 LTL_f 2DFA

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_f2DFA 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 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

```

1  import ply.lex as lex
2
3  class MyLexer(object):
4
5      reserved = {
6          'true':      'TRUE',
7          'false':     'FALSE',
8          'X':         'NEXT',
9          'W':         'WEAKNEXT',
10         'R':         'RELEASE',
11         'U':         'UNTIL',
12         'F':         'EVENTUALLY',
13         'G':         'GLOBALLY',
14         'Y':         'PASTNEXT', #PREVIOUS
15         'S':         'PASTUNTIL', #SINCE
16         'O':         'PASTEVENTUALLY', #ONCE
17         'H':         'PASTGLOBALLY' #HISTORICALLY
18     }
19     # List of token names. This is always required
20     tokens = (
21         'TERM',
22         'NOT',
23         'AND',

```

```

24         'OR',
25         'IMPLIES',
26         'DIMPLIES',
27         'LPAR',
28         'RPAR'
29     ) + tuple(reserved.values())
30
31     # Regular expression rules for simple tokens
32     t_TRUE = r'true'
33     t_FALSE = r'false'
34     t_AND = r'\&'
35     t_OR = r'\|'
36     t_IMPLIES = r'\->'
37     t_DIMPLIES = r'\<->'
38     t_NOT = r'\~'
39     t_LPAR = r'\('
40     t_RPAR = r'\)'
41     # FUTURE OPERATORS
42     t_NEXT = r'X'
43     t_WEAKNEXT = r'W'
44     t_RELEASE = r'R'
45     t_UNTIL = r'U'
46     t_EVENTUALLY = r'F'
47     t_GLOBALLY = r'G'
48     # PAST OPERATOR
49     t_PASTNEXT = r'Y'
50     t_PASTUNTIL = r'S'
51     t_PASTEVENTUALLY = r'O'
52     t_PASTGLOBALLY = r'H'
53
54     t_ignore = r'\n+'
55
56     def t_TERM(self, t):
57         r'(?![a-z])(?!true|false)[a-z]+'
58         t.type = MyLexer.reserved.get(t.value, 'TERM')
59         return t # Check for reserved words
60
61     def t_error(self, t):
62         print("Illegal character '%s' in the input formula" % t.value[0])
63         t.lexer.skip(1)
64
65     # Build the lexer
66     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 BNF² 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

```
1 import ply.yacc as yacc
2 from ltlf2dfa.Lexer import MyLexer
3
4 class MyParser(object):
5
6     def __init__(self):
7         self.lexer = MyLexer()
8         self.lexer.build()
9         self.tokens = self.lexer.tokens
10        self.parser = yacc.yacc(module=self)
11        self.precedence = (
12
13            ('nonassoc', 'LPAR', 'RPAR'),
14            ('left', 'AND', 'OR', 'IMPLIES', 'DIMPLIES', 'UNTIL', \
15             'RELEASE', 'PASTUNTIL'),
```

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

```

16         ('right', 'NEXT', 'WEAKNEXT', 'EVENTUALLY', \
17         'GLOBALLY', 'PASTNEXT', 'PASTEVENTUALLY', 'PASTGLOBALLY'),
18         ('right', 'NOT')
19     )
20
21     def __call__(self, s, **kwargs):
22         return self.parser.parse(s, lexer=self.lexer.lexer)
23
24     def p_formula(self, p):
25         '''
26         formula : formula AND formula
27                 | formula OR formula
28                 | formula IMPLIES formula
29                 | formula DIMPLIES formula
30                 | formula UNTIL formula
31                 | formula RELEASE formula
32                 | formula PASTUNTIL formula
33                 | NEXT formula
34                 | WEAKNEXT formula
35                 | EVENTUALLY formula
36                 | GLOBALLY formula
37                 | PASTNEXT formula
38                 | PASTEVENTUALLY formula
39                 | PASTGLOBALLY formula
40                 | NOT formula
41                 | TRUE
42                 | FALSE
43                 | TERM
44         '''
45
46         if len(p) == 2: p[0] = p[1]
47         elif len(p) == 3:
48             if p[1] == 'F': # F(a) == true UNTIL A
49                 p[0] = ('U', 'true', p[2])
50             elif p[1] == 'G': # G(a) == not(eventually (not A))
51                 p[0] = ('~', ('U', 'true', ('~', p[2])))
52             elif p[1] == 'O': # O(a) = true SINCE A
53                 p[0] = ('S', 'true', p[2])
54             elif p[1] == 'H': # H(a) == not(pasteventually(not A))
55                 p[0] = ('~', ('S', 'true', ('~', p[2])))
56             elif p[1] == 'W':
57                 p[0] = ('~', ('X', ('~', p[2])))
58         else:

```

```

59         p[0] = (p[1], p[2])
60     elif len(p) == 4:
61         if p[2] == '>':
62             p[0] = ('|', ('~', p[1]), p[3])
63         elif p[2] == '<->':
64             p[0] = ('&', ('|', ('~', p[1]), p[3]), ('|', ('~', p[3]), \
65                 p[1]))
66         elif p[2] == 'R':
67             p[0] = ('~', ('U', ('~', p[1]), ('~', p[3])))
68         else:
69             p[0] = (p[2], p[1], p[3])
70     else: raise ValueError
71
72
73     def p_expr_group(self, p):
74         '''
75         formula : LPAR formula RPAR
76         '''
77         p[0] = p[2]
78
79     def p_error(self, p):
80         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.

```

1 from ltlf2dfa.Parser import MyParser
2
3 formula = "G(a->Xb)"
4 parser = MyParser()
5 parsed_formula = parser(formula)
6
7 print(parsed_formula) # syntax tree as tuple of tuples

```

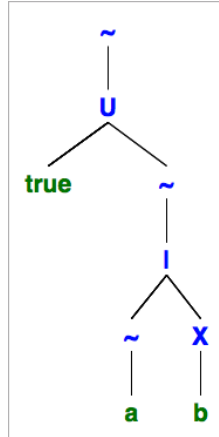



Figure 3.1. The syntax tree generated for the formula " $G(a \sim Xb)$ ". Symbols are in green while operators are in blue.

3.2.3 Translator.py

The `Translator.py` module contains the majority of APIs that the `LTLf2DFA` package exposes. Indeed, this module consists of a `Translator` class which concerns the core feature of the package: the translation of an `LTLf` 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 Graphviz³ 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` or `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 `LTLf` formula to the MONA syntax is done passing through FOL as shown in 3.1.

$$\text{LTL}_f \rightarrow \text{FOL} \rightarrow \text{MONA} \quad (3.1)$$

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

```

1 import ...
2
3 class Translator:
4     ...
5
6     def translate(self):
7         self.translated_formula = translate_bis(self.parsed_formula, \
8         var='v_0')+";\n"
9
10    ...
11
12 def translate_bis(formula_tree, var):
13     if type(formula_tree) == tuple:
14         #enable this print to see the tree pruning
15         # print(self.parsed_formula)
16         # print(var)
17         if formula_tree[0] == '&':
18             # print('computed tree: ' + str(self.parsed_formula))
19             if var == 'v_0':
20                 a = translate_bis(formula_tree[1], '0')
21                 b = translate_bis(formula_tree[2], '0')
22             else:
23                 a = translate_bis(formula_tree[1], var)
24                 b = translate_bis(formula_tree[2], var)
25             if a == 'false' or b == 'false':
26                 return 'false'
27             elif a == 'true':
28                 if b == 'true': return 'true'
29                 else: return b
30             elif b == 'true': return a
31             else: return '('+a+'&' +b+')'
32         elif formula_tree[0] == '|':
33             # print('computed tree: ' + str(self.parsed_formula))
34             if var == 'v_0':
35                 a = translate_bis(formula_tree[1], '0')
36                 b = translate_bis(formula_tree[2], '0')
37             else:
38                 a = translate_bis(formula_tree[1], var)
39                 b = translate_bis(formula_tree[2], var)

```

```

40     if a == 'true' or b == 'true':
41         return 'true'
42     elif a == 'false':
43         if b == 'true': return 'true'
44         elif b == 'false': return 'false'
45         else: return b
46     elif b == 'false': return a
47     else: return '('+a+'|'+b+')'
48 elif formula_tree[0] == '~':
49     # print('computed tree: ' + str(self.parsed_formula))
50     if var == 'v_0': a = translate_bis(formula_tree[1], '0')
51     else: a = translate_bis(formula_tree[1], var)
52     if a == 'true': return 'false'
53     elif a == 'false': return 'true'
54     else: return '~('+ a +')'
55 elif formula_tree[0] == 'X':
56     # print('computed tree: ' + str(self.parsed_formula))
57     new_var = _next(var)
58     a = translate_bis(formula_tree[1],new_var)
59     if var == 'v_0':
60         return '('+ 'ex1'+new_var+':'+ new_var +'_=1'+ '&'+ \
61             a +')'
62     else:
63         return '('+ 'ex1'+new_var+':'+ new_var +'_=' + var + \
64             '+1'+ '&'+ a +')'
65 elif formula_tree[0] == 'U':
66     # print('computed tree: ' + str(self.parsed_formula))
67     new_var = _next(var)
68     new_new_var = _next(new_var)
69     a = translate_bis(formula_tree[2],new_var)
70     b = translate_bis(formula_tree[1],new_new_var)
71
72     if var == 'v_0':
73         if b == 'true': return '('+ 'ex1'+new_var+':0<=' + \
74             new_var+ '&'+ new_var+ '<=max($)&'+ a +')'
75         elif a == 'true': return '('+ 'ex1'+new_var+':0<=' + \
76             new_var+ '&'+ new_var+ '<=max($)&all1'+ \
77             new_new_var+':0<=' + new_new_var+ '&'+ \
78             new_new_var+ '<'+ new_var+ '>'+ b +')'
79         elif a == 'false': return 'false'
80         else: return '('+ 'ex1'+new_var+':0<=' + new_var+ \
81             '&'+ new_var+ '<=max($)&'+ a + '&all1'+ \
82             new_new_var+':0<=' + new_new_var+ '&'+ \

```

```

83     new_new_var+'_<_' + new_var+'_>_' + b+'_')'
84 else:
85     if b == 'true': return '(' + 'ex1_' + new_var+'_' + var + \
86         '_<_' + new_var+'_' + new_var+'_<=max($)' + a + '_' + ')'
87     elif a == 'true': return '(' + 'ex1_' + new_var+'_' + var + \
88         '_<_' + new_var+'_' + new_var+'_<=max($)' + a + 'all1_' + \
89         new_new_var+'_' + var+'_<_' + new_new_var+'_' + \
90         new_new_var+'_<_' + new_var+'_>_' + b+'_')'
91     elif a == 'false': return 'false'
92     else: return '(' + 'ex1_' + new_var+'_' + var+'_<=_' + \
93         new_var+'_' + new_var+'_<=max($)' + a + \
94         '_&all1_' + new_new_var+'_' + var+'_<_' + new_new_var+'_' + \
95         '_&_' + new_new_var+'_<_' + new_var+'_>_' + b+'_')'
96 elif formula_tree[0] == 'Y':
97     # print('computed tree: ' + str(self.parsed_formula))
98     new_var = _next(var)
99     a = translate_bis(formula_tree[1], new_var)
100    if var == 'v_0':
101        return '(' + 'ex1_' + new_var+'_' + new_var + \
102            '_<=max($)' + a + 'all1_' + '&max($)' + a + '_' + ')'
103    else:
104        return '(' + 'ex1_' + new_var+'_' + new_var + \
105            '_<=' + var + '_<all1_' + '&_' + new_var+'_>0_' + a + '_' + ')'
106 elif formula_tree[0] == 'S':
107     # print('computed tree: ' + str(self.parsed_formula))
108     new_var = _next(var)
109     new_new_var = _next(new_var)
110     a = translate_bis(formula_tree[2], new_var)
111     b = translate_bis(formula_tree[1], new_new_var)
112
113    if var == 'v_0':
114        if b == 'true': return '(' + 'ex1_' + new_var+'_' + '0_<=_' + \
115            new_var+'_' + new_var+'_<=max($)' + a + '_' + ')'
116        elif a == 'true': return '(' + 'ex1_' + new_var+'_' + \
117            '0_<=_' + new_var+'_' + new_var+'_' + \
118            '_<=max($)' + a + 'all1_' + new_new_var+'_' + new_var+'_<_' + \
119            new_new_var+'_' + new_new_var+'_<=max($)' + b + '_' + ')'
120        elif a == 'false': return 'false'
121        else: return '(' + 'ex1_' + new_var+'_' + '0_<=_' + \
122            new_var+'_' + new_var+'_<=max($)' + a + \
123            '_&all1_' + new_new_var+'_' + new_var+'_<_' + \
124            new_new_var+'_' + new_new_var+'_<=max($)' + b + '_' + ')'
125    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 appropriate 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.

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

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.

```

1  ...
2  def compute_declare_assumption(self):
3      pairs = list(it.combinations(self.alphabet, 2))
4
5      if pairs:
6          first_assumption = "~(ex1_␣y:␣0<=y_␣&_␣y<=max($)_␣&_␣~("
7          for symbol in self.alphabet:
8              if symbol == self.alphabet[-1]: first_assumption += \
9                  'y_␣in_␣'+ symbol +'))'
10             else : first_assumption += 'y_␣in_␣'+ symbol +'_␣|_␣'
11
12         second_assumption = "~(ex1_␣y:␣0<=y_␣&_␣y<=max($)_␣&_␣~("
13         for pair in pairs:
14             if pair == pairs[-1]: second_assumption += '(y_␣notin_␣'+ \
15                 pair[0]+'_␣|_␣y_␣notin_␣'+pair[1]+ ')))';
16             else: second_assumption += '(y_␣notin_␣'+ pair[0]+ \
17                 '_␣|_␣y_␣notin_␣'+pair[1]+ ')_␣&_␣'
18
19         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_f2DFA 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.

```

1  ...
2      def invoke_mona(self, path='./inter-automa'):
3          if sys.platform == 'linux':
4              package_dir = os.path.dirname(os.path.abspath(__file__))
5              mona_path = pkg_resources.resource_filename('ltlf2dfa', 'mona')
6              if os.access(mona_path, os.X_OK): #check if mona is executable
7                  try:
8                      subprocess.call(package_dir+'./mona_u-gw' + \
9                                  './automa.mona>' + path + '.dot', shell=True)
10                 except subprocess.CalledProcessError as e:
11                     print(e)
12                     exit()
13                 except OSError as e:
14                     print(e)
15                     exit()
16             else:
17                 print('[ERROR]: MONA tool is not executable...')
18                 exit()
19         else:
20             try:
21                 subprocess.call('mona_u-gw./automa.mona>' + path + \
22                                 '.dot', shell=True)
23             except subprocess.CalledProcessError as e:
24                 print(e)
25                 exit()
26             except OSError as e:
27                 print(e)
28                 exit()
29  ...

```

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 GitHub⁴) 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.

```
1 from dotpy.parser.parser import MyParser
2 import os
3
4 class DotHandler:
5
6     def __init__(self, path='./inter-automa.dot'):
7         self.dot_path = path
8         self.new_digraph = None
9
10    def modify_dot(self):
11        if os.path.isfile(self.dot_path):
12            parser = MyParser()
13            with open(self.dot_path, 'r') as f:
14                dot = f.read()
15                f.close()
16
17            graph = parser(dot)
18            if not graph.is_singleton():
19                graph.delete_node('0')
20                graph.delete_edge('init', '0')
21                graph.delete_edge('0', '1')
22                graph.add_edge('init', '1')
23            self.new_digraph = graph
24        else:
25            print('[ERROR] No file DOT exists')
26            exit()
27
28    def delete_intermediate_automaton(self):
29        if os.path.isfile(self.dot_path):
30            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 $\varphi = \Box(a \rightarrow \Box b)$.

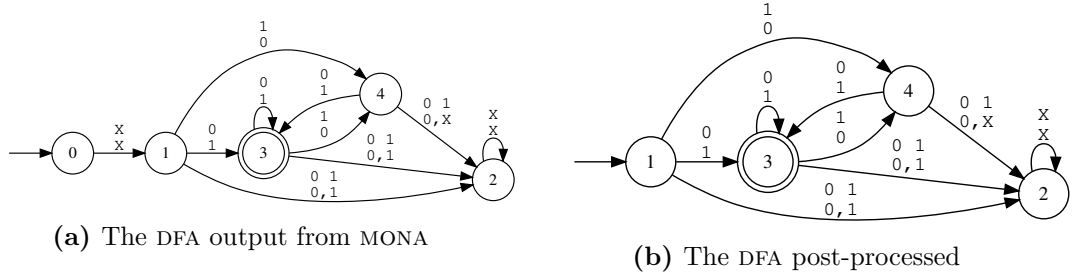


Figure 3.2. Before and after DFA post-processing

3.3 Comparison with FLLOAT

In this section, we will see how LTL_f2DFA 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_f2DFA. 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

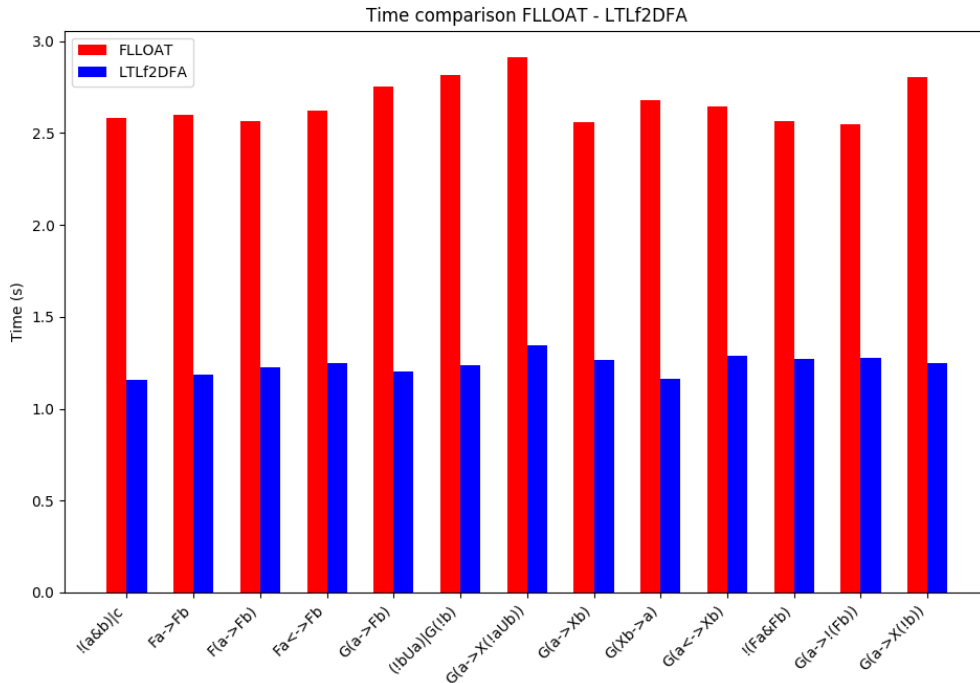


Figure 3.3. Time benchmarking of LTL_f2DFA wrt FLLOAT.

histogram, FLLOAT results are coloured in red, while LTL_f2DFA 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_f2DFA is overall 2x faster than FLLOAT. This behaviour is due to the fact that these two packages operate in a different way. Indeed, while FLLOAT is a pure Python package, LTL_f2DFA 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_f2DFA is much faster than FLLOAT,

its time execution depends on the I/O system performance which can drastically reduce it. Thus, LTL_f2DFA 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_f2DFA 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.
- 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>.