SAPIENZA
UNIVERSITÀ DI ROMA

FOND Planning for LTL_f Goals: Theory and Implementation

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

FOND Planning for LTL_f Goals: Theory and Implementation
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$_f$

intro to the chapter

2.1 LTL, LTL$_f$ and PLTL

talk about theory of logics future and past finite and infinite

2.2 LTL$_f$2DFA

talk about theory behind conversion to automata in future

2.3 PLTL2DFA

talk about theory behind conversion to automata in past

2.4 LTL$_f$2FOL and MONA

talk about theory behind translation and intro with mona future

2.5 PLTL2FOL and MONA

talk about theory behind translation and intro with mona past

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$ 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$_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',
        'U':       'UNTIL',
        'E':       'EVENTUALLY',
        'G':       'GLOBALLY',
        'Y':       'PASTNEXT', #PREVIOUS
        'S':       'PASTUNTIL', #SINCE
        'O':       'PASTEVENTUALLY', #ONCE
        'H':       'PASTGLOBALLY'
    }
    # 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'T'
    t_FALSE = r'F'
    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_UNTIL = r'U'
    t_EVENTUALLY = r'E'
    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]+'
        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):
        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 52. 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

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

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

```python

    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 PASTUNTIL formula
                | NEXT 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] == 'E': # E(a) == true UNITL A
                p[0] = ('U','T', p[2])
            elif p[1] == 'G': # G(a) == not(eventually (not A))
                p[0] = ('~',('U', 'T', ('~',p[2])))
            elif p[1] == 'O': # O(a) = true SINCE A
                p[0] = ('S','T', p[2])
            elif p[1] == 'H': # H(a) == not(pasteventually(not A))
                p[0] = ('~',('S', 'T', ('~',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 $\varphi = G(a \to Xb)$ is represented as $('\sim', ('U', 'T', ('\sim', ('|', ('\sim', 'a'), ('X', 'b')))))$ 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.

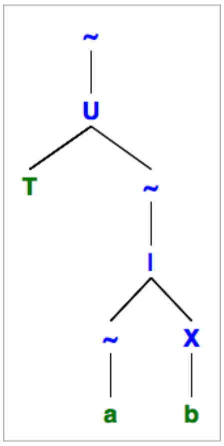


Figure 3.1. The syntax tree generated for the formula "$G(a \sim Xb)$". Symbols are in green while operators are in blue.

LTL$_f$2DFA can be used just for the parsing phase of an LTL$_f$ 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

```

```python
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$ 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.

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

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:
    ...
```

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

```

    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')
                # a = translate_bis(self.parsed_formula[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'
            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($)␣=>␣'+b+'␣)'
            else:
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+ \
                ':␣0␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+ \
                ':␣0␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣'+var+ \
                '␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
                new_new_var+'␣&␣'+new_new_var+'␣<=␣'+var+'␣=>␣'+b+'␣)'
                elif a == 'false': return 'false'
                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[0] == 'T': return 'true'
140         elif formula_tree[0] == 'F': 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.

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

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:
```

```
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.

```
1  ...
2      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

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

particular attention to performances.

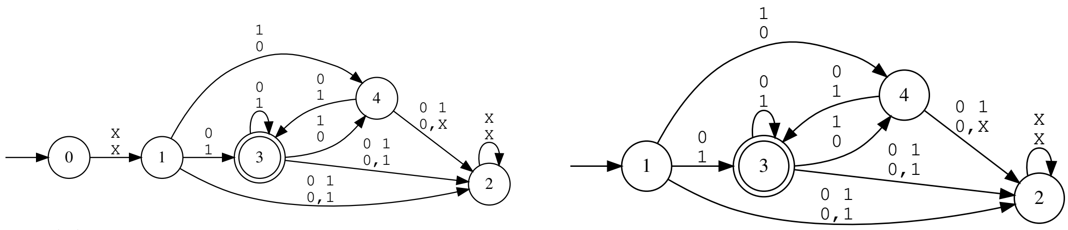
As we can see in the implementation of the `DotHandler` class in Listing 3.7, the main methods are `modify_dot` and `output_dot`.

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)
            return True
        else:
            return False

    def output_dot(self, result_path='./automa.dot'):
        try:
            if self.delete_intermediate_automaton():
                with open(result_path, 'w+') as f:
                    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

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 with respect to time and memory 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.

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.