191.106 – Runtime Verification : SS'2018

Project A: Offline LTL+Past Monitors

Mehmet Ozgan
Matr.Nr.: 0526530
{mozgan}@gmail.com

August 17, 2018

1 Introduction

In this work we concentrated on the *LTL+Past*, i.e. the *Linear Temporal Logic* (LTL) with extending *Past* operators on the finite trace. LTL+Past provides temporal operators that refer to both the past and the future states of an execution relative to a current point of reference.

Formulas of LTL+Past are built from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives. An LTL+Past formula φ is defined by the following grammar:

$$\begin{aligned} \varphi ::=\, & p && \text{atomic proposition} \\ & \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 && \text{boolean connectives} \\ & \mid \Diamond\varphi \mid \Box\varphi \mid \bigcirc^s\varphi \mid \bigcirc^w\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 && \text{LTL operators} \\ & \mid \ominus^s\varphi \mid \ominus^w\varphi \mid \varphi_1 \,\mathcal{S}\, \varphi_2 \mid \diamondminus\varphi \mid \boxminus\varphi && \text{Past operators} \end{aligned}$$

The semantics of LTL+Past is given in terms of finite traces denoting a finite sequence of consecutive instants of time, i.e., finite words π over the alphabet of $2^{\mathcal{P}}$, containing all possible interpretations of the propositional symbols in $\mathcal{P}$. For an interpretation π, we inductively define when an LTL+Past formula φ is *true* at an instant $i \in \{0, 1, \ldots, n-1\}$ written $\pi[i] \vDash \varphi$ (or $\pi[i], \varphi \vDash true$) as follows:

- $\pi[i] \vDash p \iff p \in \pi(i), p \in \mathcal{P}$.
- $\pi[i] \vDash \neg\varphi \iff \pi[i] \nvDash \varphi$.
- $\pi[i] \vDash \varphi_1 \vee \varphi_2 \iff \pi[i] \vDash \varphi_1$ or $\pi[i] \vDash \varphi_2$.
- $\pi[i] \vDash \varphi_1 \wedge \varphi_2 \iff \pi[i] \vDash \varphi_1$ and $\pi[i] \vDash \varphi_2$.
- $\pi[i] \vDash \varphi_1 \rightarrow \varphi_2 \iff \pi[i] \vDash \neg\varphi_1$ or $\pi[i] \vDash \varphi_2$.
- $\pi[i] \vDash \Diamond\varphi \iff \exists j \in [i, n] : \pi[j] \vDash \varphi$.
- $\pi[i] \vDash \Box\varphi \iff \forall j \in [j, n] : \pi[j] \vDash \varphi$.
- $\pi[i] \vDash \bigcirc^s\varphi \iff i < n$ and $\pi[i+1] \vDash \varphi$.
- $\pi[i] \vDash \bigcirc^w\varphi \iff i = 0$ or $\pi[i+1] \vDash \varphi$.

- $\pi[i] \vDash \varphi_1 \,\mathcal{U}\, \varphi_2 \iff \exists j \in [i, n) : \pi[j] \vDash \varphi_2$ and $\forall k \in [i, j) : \pi[k] \vDash \varphi_1$.
- $\pi[i] \vDash \ominus^s \varphi \iff i > 0$ and $\pi[i-1] \vDash \varphi$.
- $\pi[i] \vDash \ominus^w \varphi \iff i = 0$ or $\pi[i-1] \vDash \varphi$.
- $\pi[i] \vDash \varphi_1 \,\mathcal{S}\, \varphi_2 \iff \exists j \in [0, i] : \pi[j] \vDash \varphi$ and $\forall k \in (j, i] : \pi[k] \vDash \varphi$.
- $\pi[i] \vDash \diamondminus \varphi \iff \exists j \in [0, i] : \pi[j] \vDash \varphi$.
- $\pi[i] \vDash \boxminus \varphi \iff \forall j \in [0, i] : \pi[j] \vDash \varphi$.

Let φ, ψ be LTL+Past formulae, then the following rules are equivalent:

- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\Diamond\varphi \equiv \varphi \vee \bigcirc^s(\Diamond\varphi)$
- $\Box\varphi \equiv \varphi \wedge \bigcirc^w(\Box\varphi)$
- $\varphi \,\mathcal{U}\, \psi \equiv \psi \vee (\varphi \wedge \bigcirc^s(\varphi \,\mathcal{U}\, \psi))$
- $\varphi \,\mathcal{S}\, \psi \equiv \psi \vee (\varphi \wedge \ominus^s(\varphi \,\mathcal{S}\, \psi))$
- $\diamondminus\varphi \equiv \varphi \vee \ominus^s(\diamondminus\varphi)$
- $\boxminus\varphi \equiv \varphi \wedge \ominus^w(\boxminus\varphi)$

2 Monitoring

In this project we implemented an algorithm which is based on *on-the-fly rewriting formulas* to compute the given LTL+Past formulas on the given finite sequences in Python[1] . The above defined equivalences of LTL+Past allow us very efficient way to implement a recursive *rewriting-based* algorithms. For example; it is given an LTL+Past formula $\varphi =$ "Every $p1$ occurs $p2$ strictly on the next time":

$$\varphi = \Box(p1 \rightarrow \bigcirc^s p2). \tag{1}$$

We already know from the equivalences of LTL+Past that φ can be expressed as follows:

$$\varphi = (p1 \rightarrow \bigcirc^s p2) \wedge \bigcirc^w(\Box\varphi). \tag{2}$$

At this point $p1 \rightarrow \bigcirc^s p2$ is computed for the current position and then $\Box\varphi$ will be computed for the next position because $\bigcirc^w$ shifts the formula $\Box\varphi$ to the next position on the given sequence. This means that we update the current position for the next one and write the formula again for the updated position. This computation goes to the last position of the sequence because the formula φ is already defined in recursively way. When we have a recursive definition of the formula, then we can write the following algorithm for LTL+Past monitoring:

[1]version 3.7.

Algorithm 1: Rewriting formulas

```
Input: LTL+Past formula $f$, finite sequence $\pi$
Output: Pass iff $f$ is satisfied on $\pi$, Fail if $f$ is violated on $\pi$
1 Use equivalent rules to split $f$
    - Current obligations
    - Next step obligations
2 Read inputs and substitute the values of current obligations
3 Simplify the rest of the formula
    - Next step obligations become current obligations
4 Repeat step 1
```

2.1 LTL+Past Operators

The following table shows that how LTL+Past operator can be used in this project:

LTL+Past Operator	Interpreted in Python	Using in Python
$\neg$	`not`	`not` φ
$\vee$	`or`	φ `or` ψ
$\wedge$	`and`	φ `and` ψ
$\rightarrow$	`implies`	φ `implies` ψ
$\Diamond$	`eventually`	`eventually` φ
$\Box$	`always`	`always` φ
$\bigcirc^s$	`s_next`	`s_next` φ
$\bigcirc^w$	`w_next`	`w_next` φ
$\mathcal{U}$	`until`	φ `until` ψ
$\ominus^s$	`s_prev`	`s_prev` φ
$\ominus^w$	`w_prev`	`w_prev` φ
$\mathcal{S}$	`since`	φ `since` ψ
$\diamondminus$	`once`	`once` φ
$\boxminus$	`historically`	`historically` φ

The all LTL+Past operators are implemented in the file `LTLPast.py`. The above given formula $\varphi = \Box(p1 \rightarrow \bigcirc^s p2)$ must be written as `always(p1 implies s_next p2)` in our implementation.

2.2 Execution

We use the following usage to execute the implementation of LTL+Past monitoring:

```
python3 ltl-past-monitor.py specification.ltl inputs.csv.
```

inputs.csv contains a finite sequence, and *specification.ltl* contains an LTL+Past formula in the *infix* form which is computed on the given finite sequence. In both *inputs.csv* and *specification.ltl* the propositions must be in the form of $p[0-9]^+$, i.e. $p1, p2, p3, \ldots$.

2.3 Grammar

In this project we used TatSu which is a tool that takes grammars as input, and outputs memoizing (Packrat) PEG parsers in Python. TatSu can be installed for Python3 as follows:

`pip3 install tatsu`.

The above LTL+Past grammar is written in TatSu as follows:

```
GRAMMAR = '''
        # ignore C style comments, i.e.
        # /* this is a comment and
        #    must be ignored!
        # */
        @@comments :: /\/\*(\*(?!\/)|[^*])*\*\//

        # ignore Python style comments
        @@eol_comments :: /#.*?$/

        # LTL+Past Grammar
        @@grammar :: LTL_Past

        start = expression $ ;

        expression =
                | expression 'or' expression
                | expression 'and' expression
                | expression 'implies' expression
                | expression 'since' expression
                | expression 'until' expression
                | formula
                ;

        formula =
                | 'not' formula
                | 'w_prev' formula
                | 's_prev' formula
                | 'once' formula
                | 'w_next' formula
                | 's_next' formula
                | 'historically' formula
                | 'eventually' formula
                | 'always' formula
                | factor
                ;

        factor =
                | '(' ~ @:expression ')'
                | atom
                ;

        atom = /p\d+/ ;
'''
```

As we see the atoms are in the form $p[0-9]^+$, i.e. $p1, p2, p3, \ldots$. For example if the given LTL+Past formula is `always(p1 implies s_next p2)`, then TatSu returns the result in the structure list of list as `['always', ['p1', 'implies', ['s_next', 'p2']]]`, which is still in the infix form. To convert a formula from infix to prefix form we wrote a function namely `prefix(f)` which can be found in the file `spec.py`. This returns the formula in the prefix form as `['always', ['implies', 'p1', ['s_next', 'p2']]]`.

2.4 Data Structures

The given formula after parsing is converted from infix to prefix form and its data structure is list of lists. It is easy to see that the first element of each list is either an atomic proposition or an LTL+Past operation, which means that we can split each list in the simple way by using an `if-else` condition as follows:

```
        if isinstance(formula, str):
                # formula: an atomic proposition
                return sequence[formula][index]
        elif len(formula) == 2:
                # formula[0]: unary opereation
                f = getattr(LTLPast, formula[0].upper())
                return f(formula[1], sequence, index)
        else:
                # formula[0]: binary operation
                f = getattr(LTLPast, formula[0].upper())
                return f(formula[1], formula[2], sequence, index)
```

More general implementation for splitting of one or two formulas, namely `SplitPSI()`, can be found in the file `LTLPast.py`.

For example if the first element of the current list is an atomic proposition then we get the current value from the given sequence. If the first element of the current list is an unary operation, then we call this operation with the second element of the list. In the same way, if the first element of the current list is a binary operation, then we call this operation with the second and third elements of the list. The all of both unary and binary LTL+Past operations are implemented in the file `LTLPast.py`.

For the given finite trace we used the data structure of dictionary which is very useful in Python. For example the *inputs.csv* contains the following finite trace:

p1,	p2,	p3
1,	0,	0
0,	1,	0
0,	0,	1

After the reading the *input.csv* file, the trace is converted in the structure of dictionary form and given as follows:

```
s =
{
        'p1': [True, False, False],
        'p2': [False, True, False],
        'p3': [False, False, True]
}
```

Dictionary allows us that the keys are atomic propositions which are given in the formula, and each proposition has a finite sequence. For example if we want to access to the value of $p2$ on the i^{th} position, then we write just only `s[`$p2$`][`i`]`.

3 Tests

In this section, we want to represent the following test which is discussed in the text above. Let us write the following formula in the file *specification.ltl*: `always(p1 implies s_next p2)`. The finite trace in the file *inputs.csv* is as follows:

p1,	p2
1,	0
0,	1
0,	0
0,	0

We execute the monitoring program and this gives us the result **Pass** if the given formula is satisfied on the given trace otherwise it returns **Fail**.

```
$ python3 ltl-past-monitor.py specification.ltl inputs.csv
Pass
```

If we change the last line of the trace from $0, 0$ to $1, 0$ then the formula is violated on this trace:

```
$ python3 ltl-past-monitor.py specification.ltl inputs.csv
Fail
```

References

[1] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, Sriram Sankaranarayanan. *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications.* Lectures on Runtime Verification, Springer, 2018.

[2] Klaus Havelund, Grigore Rosu. *Synthesizing Monitors for Safety Properties.*

[3] Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila. *Simple is Better: Efficient Bounded Model Checking for Past LTL.*

[4] Marco Benedetti, Alessandro Cimatti. *Bounded Model Checking for Past LTL.* Springer-Verlag, TACAS 2003, LNCS 2619, pp. 18–33, 2003.

[5] TatSu v.4.2.6 (grammar compiler). `http://tatsu.readthedocs.io/en/stable/`