Checking LTL[F,G,X] on Compressed Traces in Polynomial Time

Minjian Zhang[1], Umang Mathur[1], and Mahesh Viswanathan[1]

[1] *University of Illinois at Urbana-Champaign*

Abstract

This document provides instructions on how to evaluate the implementation and reproduce the results from the accompanying FSE'21 article(DOI: 10.5281/zenodo.5081240).

1 Introduction

We have implement the polynomial algorithm that evaluates linear temporal logic formulas without until on SLP-compressed traces as described in the research paper. We also provide all the properties and corresponding manually encoded finite deterministic automata for reproduction of the experiment result.

2 System Requirements

- Machine with Unix based operating system, such as Ubuntu or Mac OS.
- Java - Both algorithms are implemented using java.

3 Getting Started

The directory structure of the package is shown in Figure 1.

```
Artifact_Evaluation_61
|--- README.pdf
|--- compressedAlgorithm/
|--- uncompressed/
|--- properties/
|--- sequitur/
```

Figure 1: Directory structure of package

`README.pdf` is this file, and the contents of each sub-folder can be described as follows:

- `compressedAlgorithm` - contains the algorithm introduced in the paper.
- `uncompressed` - contains 11 manually encoded automata for the corresponding properties to be run directly on uncompressed strings.
- `properties` - contains detail for each property.
- `sequitur` - contains sequitur tool to compress a given String to a SLP grammar.

4 compressed algorithm

The algorithm is written in Java. It is publicly available at https://github.com/minjian233/LTLCompressed . The algorithm takes in a path to a SLP grammar and a LTL formula as input and output a boolean value indicating whether the grammar satisfies the formula and the time taken to run the algorithm in nanoseconds.

4.1 Compiling

To compile the provided source code, simply run:

```
> cd compressedAlgorithm/src/LTL
> javac *java
```

4.2 SLP

The sequitur tool is forked from https://github.com/craignm/sequitur/.
One way the tool works in is that it assumes the input is written with numerical values separated by spaces like:

```
0 1 1 1 1
```

It generates SLP grammars such that all non terminal symbols are ordered numerical values with 0 as the initial symbol and all terminal symbols are encoded by [and]. The corresponding SLP grammar of the previous example is:

```
0 -> [0] 1 1
1 -> [1] [1]
```

To run the sequitur tool:

```
> cd /path/to/sequitur/c++
> make
>/path/to/ziptrack/sequitur/c++/sequitur -d -p -m 2000 < /path/to/base_folder/sub_folder/trace.txt > /pa
```

This command creates the SLP in the file /path/to/basefolder/subfolder/slp.txt.
Our algorithm works with any SLP grammar in the same form.

4.3 Traces

In our experiment, the traces were obtained by javamop[1] instrumenting on open source java projects from Github. All these traces can be found in https://drive.google.com/drive/folders/1rJDzoHoPY5D0-vFcpCs1DbrEt4r9pI-usp=sharing for reproduction of the experiment result. The repository contains four folders:

1. a folder named original traces containing traces like:

   ```
   hasnext_true next hasnext_true next next hasnext_false
   ```

2. a folder named converted-traces-separated-by-white-space. This folder contains traces converted to numerical values so it can be easily compressed by sequitur.

   ```
   0 1 0 1 1 2
   ```

3. a folder named converted-traces-separated-by-newline. This folder contains traces obtained by replacing white space with newlines for traces in folder converted-traces-separated-by-white-space . The traces in this folder are intended to be run directly on the automata.

4. a folder named compressed-traces with all traces compressed by sequitur.

4.4 LTL formulas

The algorithm works with LTL formulas without Until. More precisely, it assumes the following:

1. All allowed operations are ! for negation, & for conjunction, | for disjunction, X for next, G for global, F for final and $\rightarrow$ for implication.
2. The X operator is pushed in as far as it can be pushed with |, &, and F. Example: $X(a|b)$ is not considered as an valid input as $X(a|b) = Xa|Xb$. The algorithm may result in undefined behavior if inputs are not valid.

4.5 projected traces

A String s on alphabet Γ can be projected onto alphabet $\Gamma' \subseteq \Gamma$ by removing all $s[i]$ such that $s[i] \notin \Gamma'$. The experiment was done in the manner that each property has its own Γ' and each trace will be projected before it is run against the property. This is implicitly handled by the algorithms we implemented and users only need to provide Γ'. If the projected trace is empty, the algorithm will answer false regardless of the formula as finite LTL is only defined on non-empty strings.

4.6 usage

We have manually encoded all the 13 properties(indexed from 0-12) in the algorithm, to run the nth property simply call:

```
>cd /path/to/compressedAlgorithm/src
>java -Xss4m LTL.Main path_to_compressed_file n
```

To run the tool with other formulas like $F(a) \rightarrow G(!b)$ with $\Gamma' = \{a, b, c\}$. Call

```
>java -Xss4m LTL.Main path_to_compressed_file <F(a)->G(!b)> <a,b,c>
```

To run it without projection, simply run without the last argument:

```
>java -Xss4m LTL.Main path_to_compressed_file <F(a)->G(!b)>
```

5 uncompressed algorithms

We compared the running time of the algorithm against the running time of running the uncompressed traces on the Rabin automata directly. For every property, Rabinizer-4.0 generates a Rabin automata, which is essentially a finite state machine, together with an acceptance condition for deciding membership of infinite words. For our purpose, however, we transform the acceptance condition of these automata so that they are suitable for analyzing finite traces. All information about this procedure can be found in the property directory. Since we do not count the time to construct the automata, any method that captures satisfiablity of finite LTL formulas would work properly. The reviews are welcome to use their preferred tool. We have also manually encoded 13 automata for all the properties we checked during the experiment. To run the nth property against an uncompressed file, run:

```
>cd /path/to/uncompressed/src
>javac *java
>java Main path_to_uncompressed_file n
```

These automata will output the truth value and the time taken in nanoseconds.

6 Reproducing Results from Paper

We ensured the following while implementing the two approaches:

1. The IO time are excluded. This is important since the the IO time is very heavy when traces are not compressed.

2. No termination criteria other than completion of processing the whole input is set.

To reproduce the experiment result, all needs to be done is to run the compressed algorithm on every trace in the compressed-traces folder against all 13 encoded properties and run the encoded automata on every trace in the converted-traces-separated-by-newline folder. The experiment used real time as the measurement for speedups.
We have added an option to run all the traces in a directory with all the formulas at once. For compressed Algorithm, run:

```
>java -Xss4m LTL.Main -all path_to_directory path_to_output
```

For uncompressed Algorithm, run:

```
>java Main -all path_to_directory path_to_output
```

Note that the time taken to run the compressed algorithm is very short for most compressed traces, and the reviewers are welcome to run the algorithm in single-user mode or change the measurement from real time to CPU time if the result is inconsistent with what was reported in the paper due to the noises added by other processes. We expect that this won't happen in most of the situations.

References

[1] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–613, 2016.