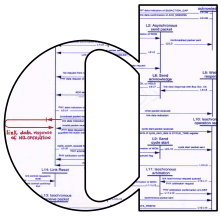
Saarland University
Faculty MI
Computer Science

Verification – WS 2018/2019

Prof. Dr.-Ing. Holger Hermanns
Daniel Stan, PhD
https://dcms.cs.uni-saarland.de/v_1819/



Project Description 2

1 Introduction

The goal of the second project is to implement a LTL model checking algorithm. Some additional files and dependencies have been added to the project template, you can either download an up-to-date copy of this template on the dCMS, or only uncompress an updated version.

The answer to the following project tasks are expected to be provided as implementation of the methods of the class `Part2`. Our grading script will instantiate class `Part2` and run each method corresponding to a question with several models and specifications and check for the correct answer.

2 Persistence checking

In this first part, you are given a transition system $\mathcal{T}$ and an atomic proposition ϕ (provided as a `TFormula.Proposition`) and solving the persistence problem, that is to say: does there exists an infinite path $(s_i)_i$ in $\mathcal{T}$ whose trace $(l(s_i))_i$ satisfies ϕ for infinitely many positions.

We will assume, as in the first project, that $\mathcal{T}$ has a finite number of states and thus, persistence of ϕ can be summarized as the existence of a finite path fragment $s_0 s_1 \dots s_k s_{k+1} \dots s_n$ with $n < |\mathcal{T}|$ and such that $s_k = s_n \vDash \phi$. Such a path is called *a* witness *for* ϕ *on* $\mathcal{T}$.

(a) Implement the function `persistenceWit(LTS, TFormula.Proposition, int): List<State>` returning a witness for a given atomic proposition on a given model, under the assumption that the model has less than n (given as an argument) states. The function should return `null` when:

- No such witness exists.
- The model has more than n states.
- A terminal state is found during the exploration.

Ideally, your implementation should not explore the whole state space if a witness is found at an early stage of the implementation.

3 LTL to NBA: Santa was here !

The second ingredient of a LTL model checker is a translation of a LTL formula to a non-deterministic Büchi automaton. As this will later discussed in the lectures, such a translation is involving and due to time constraint (Christmas break) we provide you with an automatic translation from LTL to NBA, that you can see as a ~~black box~~ Christmas present ! This box works as follows: given a LTL formula φ, it produces a `NBA` object containing:

- `aut`: An ω-automaton object, provided by the jhoafparser library. This library allows to specify more complicated automata, and a full documentation can be found online at this address: `https://automata.tools/hoa/jhoafparser/docs/javadoc/`. Here, our provided translation guarantees you a valid non-deterministic Büchi automaton [1] $\mathcal{A}_\varphi$ recognizing the language of the formula φ.
- `apMap` a mapping from atomic propositions indices specified in the automaton, to actual Muds boolean expressions objects, that were appearing in φ.

[1] In the HOA library terminology, a NBA as defined in our lecture, is an ω-regular automaton with **only one** acceptance condition, which is a **Büchi** acceptance **over states**.

If the input formula is not a valid LTL formula, an exception is thrown.

(b) Construct the automaton for $(a \mathsf{U} z) \wedge (b \mathsf{U} z)$ how many states does our black box produce ? How many states would have been produced by the "closure" translation seen in class ? You can give your answer (with comments) as part of the two methods `nbStatesBb(TFormula): int` and `nbStatesCl(TFormula): int`.

(c) Given a model $\mathcal{T}$, a LTL formula φ and a fresh atomic proposition a_f. Construct the composed model $\mathcal{T}' = \mathcal{T} \otimes \mathcal{A}_\varphi$ as seen in the lecture, such that there exists a path in $\mathcal{T}$ satisfying φ if and only if there exists a path in $\mathcal{A}'$ satisfying $\Box\Diamond a_f$. Your answer will take the form of an implementation of the method: `product(LTS, TFormula, TFormula.Proposition): LTS`.

Your implementation can define a subclass of `LTS` in order to implement the composition construction and should, ideally, construct the state space on the fly.

4 LTL model checking

(d) Conclude by implementing the method `solve(LTS, TFormula, int)` returning `true` if and only if the input formula ϕ is an LTL formula, and the input model has less than n states, and satisfies ϕ, `false` otherwise.

(e) Bonus Question: Can you adapt the algorithm to handle any `TFormula`, that is to say, any CTL* formula. You *may* want to reuse the algorithms of `Part1`.

5 Submission

You have to submit you project until 10$^{\text{th}}$ of January 2019. Submit your project in teams of two to three students. It is your responsibility to form such groups.

Project submission is handled through the dCMS. Submission will be taken in the form of a zip-file that should contain all source files that are part of your implementation. Only one person from your group should submit the project. The project must include a `.txt` file that indicates the participants in that particular group!

Passing the projects is mandatory to be admitted to the exam! Be sure to test your project thoroughly, with your own examples, before submitting your code.