185.A05 Advanced Functional Programming SS 22

Monday, 05/02/2022

Assignment 5

Model Checking Project: MiniCheck

All Chapters

Topic: Building an LTL Model Checker plus Implementing a Choice of Modelling/Verification Extensions

Submission deadline: Friday, 06/13/2022, noon (no second submission)

Contact: Francesco Pontiggia, francesco.pontiggia@tuwien.ac.at

The goal of the project is to implement an automata-based LTL Model Checker, *MiniCheck*. The project consists of two parts: the core model checker and the elective modular extensions. In turn, the core model checker requires resp.:

- a number of preliminary programming tasks (**Module 0**, cf. Section 2.1 and 2.2).
- an automaton construction module which builds an appropriate automaton from the input LTL formula (**Module 1**, cf. Section 2.3 and 2.4).
- a cycle detection module which verifies the cartesian product of the automaton and a given Transition System (**Module 2**, cf. Section 2.3 and 2.5)).

The core is mandatory but is only worth a portion of the points achievable in this project.

In contrast to previous exercises, you are not asked to implement functions with given signatures. Instead, the functionality of the executed program is evaluated. You will have to design your own function signatures and data types and decide which data structures you want to use.

The program should not be written in a single file, but as a `stack` or `cabal` project consisting of multiple modules. It is encouraged to use custom and predefined data structures, as well as external libraries and Haskell language extensions **where appropriate**. It is not required to get the project to run on g0.

There will be a Q&A session where help will be provided in case there are troubles with the selected technologies. The date will be announced via TUWEL news.

1 Grading

The preliminary programming tasks with a parser at its core (**Module 0**), the automaton construction module (**Module 1)** and the cycle detection module (**Module 2**) of MiniCheck are mandatory. Implementing this core awards up to 200 points. The project presentation will be worth up to 100 points. To gain more points on top of the aforementioned, you can choose from some modelling and verification extensions of the MiniCheck tool.

For a positive evaluation of this project at least 200 points are required. A positive grade on the project is necessary in order to receive a positive grade in this course.

The total points for the project are calculated by adding up the points on the core, the selected extensions and the submission talk. A positive grade on the core itself is *not* required. At most 300 points are awarded for the project implementation.

In contrast to previous exercises, there is no second submission.

1.1 Note

We assume a general knowledge of model checking and its purpose, as well as basis of logics, such as propositional logic. Besides this, some previous knowledge in related fields helps and makes the project smaller, but it's not mandatory. This includes also:

- Automata Theory
- Graph Algorithms

See, e.g., [1] for details (a PDF version is available on the Github repository).

2 MiniCheck Core (200P)

Model Checking consists of formally and exhaustively verifying a formula against all the possible executions of a particular type of automata. The automata-based approach we are going to follow has been originally proposed by Vardi and Wolper [2]. In the following, we provide some preliminary information about the type the formalisms you are required to support, and then present the project to carry out. For a complete and deeper introduction to our approach in model checking, please, refer to [1] (a PDF version is available on the Github repository).

2.1 Preliminaries - Transition Systems

As modelling formalism, we use *Transition Systems* (TS). They are a particular variant of *Finite State Automata* which allow to describe adequately both hardware and software systems. They are defined over infinite runs, and they do not have a set of final states. Formally, a TS is a tuple $(S, Act, \rightarrow, I, AP, L)$, where

- S is a set of *states*,
- Act is a set of *actions*,
- $\longrightarrow \subseteq S \times Act \times Act$ is a *transition relation*,
- $I \subseteq S$ is a set of *initial states*.
- AP is a set of *atomic propositions*, and
- $L : S \rightarrow 2^{AP}$ is a *labelling function*.

Actions are used to specify how the system evolves from one state to another. The labelling function maps every state to the set of atomic propositions holding in that state. The initial state is chosen nondeterministically between all states $\in I$. We

adopt a **state-based** approach: we consider (and want to verify) the labels in the state sequence of a run of the TS, and abstract from actions, which are of no use in our verification algorithm.

Note that, for simplicity sake, we consider only TS with no terminal states, i.e., every state has at least one outgoing edge. While this prevents some technical problems from occurring, it is not a limit to the expressive power of TS: for every state s without outgoing edge, define a transition to a sink state s_{sink}, and then define a self transition from s_{sink} to itself.

Resources

[1, Paragraph 2.1]

Task 1

Define:

- a suitable plain-text representation of TS such that TS written in this representation can be passed as input to your tool, and
- a data type to represent them.
- the type of the set of atomic propositions that can be used to label states of the Transition System. [There needs to be some detail on the kind of atomic propositions, I think; otherwise, I can not imagine how to meaningfully define a type for something completely unknown.]

Implement:

- A parsing function from the plain-text representation of TS to the data type.

Take care of making sure that there are no terminal states, that there is at least an initial state, and that your formalisation of the set of atomic proposition is respected. The parsing function should abort when it encounters a non well formed Transition System.

You are allowed to choose one of the parsing approaches presented in the lecture or use an external monadic parsing library from Hackage. Document your choice.

We require that a state is always labelled with itself (i.e., $s_i \in L(s_i)$, which implies $S \subset AP$)). This does not mean that the atomic propositions we are considering are only state identifiers [cp. note above, here needs to be some more detail, I think]).

2.2 Preliminaries - Linear Temporal Logic

To express properties to verify on Transition Systems, we consider a temporal extension of propositional logic, i.e., the introduction of temporal modalities on top of propositional logic. In particular, the temporal modalities we consider are $\bigcirc$ (pronounced "next") and $\mathcal{U}$ (pronounced "until"). The atomic proposition $a \in AP$ stands for the state label a in a TS.

Syntax

Given a set AP of atomic propositions, with $a \in AP$, LTL formulas follow the following syntax:

$$\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

In addition, we introduce the following *derived* Boolean operators:

$$\begin{aligned}
&\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \varphi_2) \\
&\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2 \\
&\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
&\varphi_1 \oplus \varphi_2 \equiv (\varphi_1 \wedge \neg\varphi_2) \vee (\varphi_2 \wedge \neg\varphi_1)
\end{aligned}$$

Likewise, we derive some well-known temporal modalities (pronounced "eventually" and "generally/always", respectively):

$$\begin{aligned}
&\Diamond \varphi \equiv \text{true}\,\mathcal{U}\, \varphi \\
&\Box \varphi \equiv \neg \Diamond \neg\varphi
\end{aligned}$$

You are required to handle all and only these operators (this excludes the weak until operator, for example).

Semantics

We define the semantics of LTL formulas over a trace of a Transition System $(S, Act, \rightarrow, I, AP, L)$. Given an infinite sequence of states $s_0 s_1 s_2 \ldots$ (i.e. a path) of the TS, its trace is just the induced infinite sequence of atomic propositions $\sigma = L(s_0)L(s_1)\cdots \in (2^{AP})^\omega$. Let $\sigma = A_0 A_1 A_2 \ldots$. Then $\vDash$ (pronounced "satisfies") is defined by:

$$\begin{aligned}
&\sigma \vDash \text{ true} \\
&\sigma \vDash a \text{ iff } a \in A_0 \\
&\sigma \vDash \varphi_1 \wedge \varphi_2 \text{ iff } \sigma \vDash \varphi_1 \text{ and } \sigma \vDash \varphi_2 \\
&\sigma \vDash \neg\varphi \text{ iff } \sigma \nvDash \varphi \\
&\sigma \vDash \bigcirc \varphi \text{ iff } \sigma[A_1 \ldots] \cdots \vDash \varphi \\
&\sigma \vDash \varphi_1 \,\mathcal{U}\, \varphi_2 \text{ iff } \exists j \geqslant 0. A_j \cdots \vDash \varphi_2 \text{ and } A_i \cdots \vDash \Phi_1, \text{ for all } 0 \leqslant i < j
\end{aligned}$$

An LTL formula φ holds in a path π if it holds for $trace(\pi)$. IT holds for a state s if it holds for all paths starting in s. It holds for a TS if and only if it holds for all the infinite paths starting in an initial state of the TS, or, in other words, if it holds for all initial states.

Task 2

Define:

- a suitable plain-text representation of LTL formulas such that formulas written in this representation can be passed as input to your tool, and
- a data type to represent them.
- the type of the set of atomic propositions that can be used in the formulas [see above, more detail required?].

Implement:

- A parsing function from the plain-text representation to the data type.

Note that the Atomic Propositions of LTL formulas must match the Atomic Proposition that we have previously defined and implemented to label the states of the Transition System. The parsing function should abort when it encounters a non well formed LTL formula.

You are allowed to choose one of the parsing approaches presented in the lecture or use an external monadic parsing library from Hackage. Document your choice.

Resources

[1, Paragraph 5.1 (in particular 5.1.1 and 5.1.2)]

2.3 Preliminaries - The overall Model Checking algorithm

Algorithm 1 Automaton Based LTL Model Checking

Input: finite transition system TS and LTL formula φ (both over AP)
Output: "yes" if $TS \vDash \varphi$; otherwise, "no"

1: **Module 1**: Construct an NBA $\mathcal{A}_{\neg\varphi}$ such that $\mathcal{L}_\omega(\mathcal{A}_{\neg\varphi}) = \text{Words}(\neg\varphi)$ (refer to [1] for details!)
2: **Module 2**: Construct the product transition system $TS \oplus \mathcal{A}$; determine if there exists a path π in $TS \oplus \mathcal{A}$ satisfying the acceptance condition of $\mathcal{A}$ [$\oplus$ was introduced in Section 2.2 as LTL operator; here it is used as operator on automata: some hint seems required.]
3: **if** there exists such a path **then**
4: return "no"
5: **else**
6: return "yes"
7: **end if**

It is up to you to decide how to combine the two plain text representations of the inputs (the TS and the formula), and how to provide them to the Model Checker via some command-line arguments. Any potential issue with command line arguments must be treated adequately (e.g., missing arguments, too many arguments, wrong arguments, …).

The results of a model check query should be printed on the CLI (command line interface), together with some additional information you find useful.

2.4 Module 1 - Automaton Construction from an LTL formula

In this module, you will have to build a *Generalized Nondeterministic Büchi Automaton*(GNBA) that recognizes all and only the traces that satisfy an LTL formula φ. The algorithm you have to follow is given in [1, Section 5.2]. In particular we refer you to the constructive procedure given by the proof of Theorem 5.37.

elaborate a little bit on this

Note 1

Recall that, in order to prove that the input formula φ holds for all initial paths of a TS, we have to verify the TS against the automaton for the **negation** of the input formula, i.e., an automaton-representation for the negated formula $\neg\varphi$.

Note 2

While in module 2 the product transition system is build upon a Nondeterministic Büchi Automaton (NBA), we note than a GBA and a NBA differ only for the acceptance condition. This difference can be hidden in the data type you use to represent the automata (e.g., through some higher-order functions), therefore it is not needed to implement the transformation "GNBA $\rightsquigarrow$ NBA" given by Theorem 4.56, as indicated by the textbook.

2.5 Module 2 - Fair-cycle Detection of the product transition System

Generating the product Transition System

This task consists of implementing the generation of the product transition system $TS \oplus \mathcal{A}$ starting from the input TS and the (generalized) NBA $\mathcal{A}$ built from the input formula in module 1. For this purpose, we refer you to the product construction of [1, Section 4.4.1].

Detecting Fair cycles

This task consists of implementing an exploration algorithm for the transition relation of the product automaton. The goal is to find a so called *fair* cycle, i.e., a cycle which satifies the acceptance condition of the underlying NBA. The existance of such cycle, and the reachability of such cycle from an initial state, allows us to conclude that there exist an infinite path of the TS which does not satisfy the input formula φ. Two algorithms for this purpose are presented in [1, Section 4.4.2].

Note

You have to choose an approach to handle the global variables of the fair-cycle detection algorithm. There are two main approaches available (but you may come up with a new one):

- continuation: the exploration function calls itself recursively with its updated global variables and a new state to be explored

- monadic: use the state monad presented in the lecture or the state monad as defined in the `transformers` library.

2.6 Test Suite

Projects of this size profit from an automatic test-suite. Write a test-suite for the major components of the project using one of the following frameworks:

- `hspec`, a tutorial can be found `here`
- `tasty`
- `HTF`
- `HUnit`

As before, it is encouraged to utilise external libraries for writing tests.

It is required to write at least three unit tests per major component, where major components are:

- MiniCheck automaton construction
- MiniCheck cycle detection

To test the automaton costruction function, a recommended way is to run some strings on the automaton, and verify that the acceptance of the string is as expected.

To test the cycle detection algorithm, a smart way is to implement the bonus Section 4

What can we do with QuickCheck on the model Checker?

3 MINI Extensions

This section describes some extensions to MiniCheck. Each extension is worth the given amount of points. Note that none of the described extensions are necessary for a positive grade.

3.1 Parsing MINI Programs (100 P)

With this extension, you are required to extend the modelling power of MiniCheck targeting software verication. While TS are good for expressing hardware models, they may be too low-level to model software programs. To help the user of your version of MiniCheck, you will implement an automated translation procedure for MINI programs into TS.

write me

Shall we include also While loops?

A Note about terminal states

write me

3.2 CTL model checking (100 P)

With this extension, you are required to extend the verification power of MiniCheck targeting a new class of properties, those based on a *branching* notion of time. Indeed, while in LTL a property for a state always range over all possible paths starting from that state, in some situations we would like to reason only about some of such paths. With this approach, the notion of time corresponds to that of an infinite tree of states, instead of an infinite sequence. Infact, branching time refers to the fact that at each moment there may be several different possible futures. By fixing a choice of a state at every subtree, we get one of all *possible* futures. Each traversal of the tree starting in its root represent a single path. The tree rooted at state s thus represents all possible infinite computations in the transition system that start in s.

Computation Tree Logic (CTL) thus allows to express properties for *some* or *all* computations that start in a state. For this purpose, it features two operators: an existential path quantifier ($\exists$) and a universal path quantifier ($\forall$).

As modelling formalism, you are required to use the same as in the core modules.

CTL Syntax

As in LTL, at the bottom of CTL there are atomic propositions AP which represent the labels for the states in a TS.

$$\begin{array}{ll} \textit{(state) formulas} & \Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi \\ \textit{path formulas} & \varphi ::= \bigcirc \Phi \mid \Phi_1 \,\mathcal{U}\, \Phi_2 \end{array}$$

Greek capital letters denote CTL (state) formulas, whereas lowercase Greek letters denote CTL path formulas. A well defined CTL formula is a CTL state formula.

As in LTL, you are required to implement also the derived Boolean operators and the temporal modalities “Eventually” and “Always”.

add the translation of this formulas

CTL Semantics

Given atomic proposition $a \in AP$, $TS = (S, Act, \rightarrow, I, AP, L)$, state $s \in S$, CTL state formulas Φ, Ψ, CTL path formula φ, the semantics of CTL is defined in terms of the following satisfaction relation $\models$ for state s (compare it with the LTL satisfaction relation, which was defined over a trace σ):

$$\begin{aligned} &\sigma \models \text{true} \\ &\sigma \models a \text{ iff } a \in L(s) \\ &\sigma \models \neg\Phi \text{ iff not } s \models \Phi \\ &\sigma \models \Phi \wedge \Psi \text{ iff } (s \models \Phi) \text{ and } (s \models \Psi) \\ &\sigma \models \exists\varphi \text{ iff } \pi \models \varphi \text{ for some } \pi \in Paths(s) \\ &\sigma \models \forall\varphi \text{ iff } \pi \models \varphi \text{ for all } \pi \in Paths(s) \end{aligned}$$

Given a path π, $\vDash$ is defined for path formulas as follows:

$$\pi \vDash \bigcirc \Phi \text{ iff } \pi[1] \vDash \Phi$$
$$\pi \vDash \Phi \,\mathcal{U}\, \Psi \text{ iff } \exists j \geqslant 0.(\pi[j] \vDash \Psi \wedge (\forall 0 \leqslant k < j.\pi[k] \vDash \Phi))$$

A TS satisfies CTL formula Φ if and only if Φ holds for all initial states.

Task

Define:

- a suitable plain-text representation of CTL formulas such that formulas written in this representation can be passed as input to your tool, and
- a data type to represent them.
- the type of the set of atomic propositions (APs) that can be used in the formulas (the same reasoning for LTL APs applies).

Model Checking procedure

The model checking procedure for CTL formulas differs completely from LTL verification. It's essentially based on a bottom-up traversal of the parse tree of the formula at hand, and it is considered way more efficient and straightforward, since it does not involve any notion of corresponding automaton for the input formula. Instead, given a TS and a CTL formula Φ, to verify whether TS $\vDash \Phi$, we establish whether Φ is valid in each initial state s of TS. Therefore, the procedure is roughly composed of two steps:

- computing the set $Sat(\Phi)$ of all states satisfying Φ (recursively), and
- checking whether all initial states $s \in I$ belong to $Sat(\Phi)$;

In other words, TS $\vDash \Phi$ if and only if $I \subseteq Sat(\Phi)$.

The general algorithm

The following algorithm sketches the procedure, where it makes use of the set $Sub(\Phi)$ of subformulas of Φ.

Algorithm 2 CTL Model Checking

Input: finite transition system TS and CTL formula Φ (both over AP)
Output: "yes" if $TS \vDash \Phi$; otherwise, "no"

```
1: function SatFun(φ)
2:     if φ contains state subformulas then
3:         Sat(ψ1) = SatFun(ψ1)
4:         Sat(ψ2) = SatFun(ψ2) for children nodes ψ1, ψ2 of the parse tree of φ, the so called
   maximal proper subformulas
5:         combine Sat(ψ1), Sat(ψ2) depending on the operator of φ (∧, ∃○, ∃𝒰, ∃□).
6:     else
7:         compute directly the set Sat(φ).
8:     end if
9:     return Sat(φ).
10: end function
11: return I ⊆ Sat(Φ).
```

Definition of $SatFun$

4 Bonus Task: Related LTL problems (10P)

4.1 Satisfiability Checking

On the `--sat` flag, the Model Checker constructs the FSA starting from the input formula and perform the fair-cycle detection module on **this automaton only**. This procedure amounts at verifying whether the formula is satisfiable by any string.

4.2 Validity Checking

On the `--val` flag, the Model Checker constructs the FSA starting from **negation of** the input formula and perform the fair-cycle detection module on **this automaton only**. Then, it must return the negation of the result. This procedure amounts at verifying whether the formula is valid, i.e., it is satisfied by any string.

Note

It is up to you to decide whether changing the format of the input file(s) (e.g., by forbidding to include the Transition System in the file) that can go in hand with the `--sat` and `--val` flags.

Resources

[1, Paragraph 5.2.2]

5 Bonus Task - Returning a Counterexample for LTL properties (10 P)

On the `--ce` flag, the Model Checker returns a Counterexample (CE) if the formula is not satisfied. A CE is a sequence of states of the TS of the form $s_0 s_1 \dots (s_n \dots s_m)^\omega$ such that $L(s_0)L(s_1)\dots(L(s_n)\dots L(s_m))^\omega$ does not satisfy the input property φ.

6 Bonus Task: Project Management (25P)

There is more to a successful project than just writing code. Usually, you also need to write proper documentation, distribute it, etc...

In this section, you will try some project management mechanisms for Haskell: In particular, you will provide documentation for your program, find the coverage of your test-suite and learn more about basic profiling with GHC.

6.1 Command-Line Interface

The program so far only has very rudimentary argument parsing, allowing a single filepath. However, anyone who did not write this program has no idea about how to invoke the Model Checker correctly, thus, we want to have a proper command-line interface

Implementation Suggestions

Common libraries for such tasks are `optparse-applicative` and `cmdargs` but it is also valid to not use any libraries at all and design your own solution.

Required Flags

The program should be able to understand the following flags:

- On `-h`/`--help`, a help message should be displayed, explaining how the program can be invoked correctly.
- On `--ts`, the input transition system
- On `--extensions`, a list of supported MINI extensions is printed.
- Other flags as you see appropriate. (optional. not graded)

6.2 Documentation

Document the most important types and functions of your project `haddock`-conformly and provide the documentation via HTML.

Helpful commands:

- `cabal haddock` for building the documentation and inspecting it locally.

- `cabal haddock --haddock-for-hackage` for building a `.tar.gz` containing the documentation.
- `stack haddock --open`: builds documentation and opens it in the browser upon completion.

6.3 Test Coverage

Generate the test-coverage of your program's test-suite. Discuss your findings and investigate any unexpected results.

Helpful commands:

- `cabal test --enable-coverage`
- `stack test --coverage`

6.4 Profiling

Being able to profile your code is of great importance in real-world projects. Thus, we want you to experiment with some profiling in Haskell.

To do that, you might have to build your project in profiling mode:

- `cabal build --enable-profiling`
- `stack build --profile`

Then you can pass RTS arguments to the GHC program to obtain run-time information, such as which function you spend the most amount of time in, etc...

Refer to the GHC documentation for the exact `profiling flags` to obtain relevant information.

Answer the following questions:

- What is the memory usage over time?
- What is the peak memory usage?
- Which function is the most time spent in?
- Which type requires the most amount of memory?

7 Submission artefacts

You should submit a zip-archive containing all project source files and a PDF with detailed project documentation in your group submission directory.

7.1 Project Implementation

The project should be written as a `cabal` or `stack` project consisting of multiple modules.

7.2 Test Suite

Unit tests and property tests need to be submitted as well, and it must be possible to run the whole test-suite with either `cabal test` or `stack test`.

Project Documentation

The project documentation pdf should cover at least these topics and explain your choices.

Add some appropriate questions here

- Which project build tool is used for the project? (`cabal` or `stack`)
- Which GHC version is used?
- How can the program binary be built? How can it be run?
- Which libraries are included as dependencies and which Haskell language extensions are enabled?
- Which Framework and libraries are used for writing tests?
- Which MiniCheck extensions are implemented?
- How is the functionality partitioned into different modules?
- How do you test your program? Which parts are the focus of your tests? Do there exist parts of the code that cannot be tested?
- Are there known issues and limitations of your program?

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X.

[2] Moshe Vardi and Pierre Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In: Jan. 1986, pp. 332–344.