185.A05 Advanced Functional Programming SS 22

Monday, 05/02/2022

Assignment 5
Model Checking Project: MiniCheck
All Chapters

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

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

Contact: Samuel Pilz, samuel.pilz@student.tuwien.ac.at, Francesco Pontiggia, francesco.pontiggia@tuwien.ac.at

The goal of the project is to implement a Computational Tree Logic (CTL) 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).
- a verification module which checks the input CTL formula on a Transition System (TS)(**Module 1**, cf. Section **??** and 2.3).

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**) and the verification module (**Module 1**) 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. 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 $(S, Act, \to, I, AP, L)$, where

- S is a set of *states*,
- Act is a set of *actions*,
- $\longrightarrow \subseteq S \times Act \times S$ is a *transition relation*,
- $I \subseteq S$ is a set of *initial states*.
- AP is a set of *atomic propositions*, and
- $L : S \to 2^{AP}$ 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. These are usually either a description of the state (e.g., OFF for a system that describes a lightbuld) or a property that holds in the state (e.g. $nsoda == 0$ for a system that describes the functioning of a soda vending machine). We refer to the example in Section 2.6

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. As introduced earlier, a state may be labelled with different types of properties depending on the context: it will be up to you to define a suitable type to make your program be able to encode and deal with as many different situations as possible.

2.2 Preliminaries - Computational Tree 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"). These temporal modalities allow to constraint the future states (or, in fact, their labels), that can be visited by a path.

Moreover, CTL (differently with respect to some other logics, e.g. LTL) is based on a *branching* notion of time. With this approach, future time corresponds to an infinite tree of states than can be vsited after the current one. 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 ($\forall$).

The atomic proposition $a \in AP$ stands for the state label a in a TS.

CTL Syntax

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

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

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.

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

$$\begin{aligned} \Phi_1 \vee \Phi_2 &\equiv \neg(\neg\Phi_1 \wedge \Phi_2) \\ \Phi_1 \rightarrow \Phi_2 &\equiv \neg\Phi_1 \vee \Phi_2 \\ \Phi_1 \leftrightarrow \Phi_2 &\equiv (\Phi_1 \rightarrow \Phi_2) \wedge (\Phi_2 \rightarrow \Phi_1) \\ \Phi_1 \oplus \Phi_2 &\equiv (\Phi_1 \wedge \neg\Phi_2) \vee (\Phi_2 \wedge \neg\Phi_1) \end{aligned}$$

Likewise, we derive some well-known temporal modalities:

- $\Diamond$ (pronounced “eventually”), and
- $\Box$ (pronounced“generally/always”);

These operators can be derived from the base operators through the followi

$$\begin{aligned} \exists\Diamond\,\Phi &\equiv \exists(\text{true}\,\mathcal{U}\,\Phi) \\ \forall\Diamond\,\Phi &\equiv \forall(\text{true}\,\mathcal{U}\,\Phi) \end{aligned}$$

$$\begin{aligned} \exists\Box\,\Phi &\equiv \neg\forall\Diamond\neg\Phi \\ \forall\Box\,\Phi &\equiv \neg\exists\Diamond\neg\Phi \end{aligned}$$

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

Hint about derived operators

If your code translates formulas with derived operators into formulas with only base operators through these equivalence rules, you will have implemented support of derived operators for free.

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:

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

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

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

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).

Resources

[1, Paragraph 6.2 (in particular 5.1.1 and 5.1.2)]

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.

2.3 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]. We will now outline the main definitions and steps involved in the construction.

Generalized NBA

A GNBA is a tuple $\mathcal{G} := (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where:

- Q is the (finite) set of states.
- Σ is the alphabet (and in our case it will always be the set 2^{AP}).
- $Q_0 \subseteq Q$ is the set of initial states.
- $\delta : Q \times 2^Q$ is the (nondeterministic) transition relation
- $\mathcal{F}$ is a subset of 2^Q.

The definition of a GNBA is the same as the definition of a nondeterministic Buechi Automaton, apart from $\mathcal{F}$. $\mathcal{F}$ is the so called *set of acceptance sets*. The accepted language $\mathcal{L}_\omega(\mathcal{G})$ consists of all ω-words that have at least one infinite run $q_0 q_1 \ldots$ in $\mathcal{G}$ such that for each acceptance set $F \in \mathcal{F}$ there are infinitely many indices i with $q_i \in F$.

Closure of a LTL formula φ

The *closure* of a LTL formula φ is the set $Cl(\varphi)$ which contains all subformulas ψ of φ as well as their negations. Formally , $Cl(\varphi)$ is the smallest set such that:

- $\varphi \in Cl(\varphi)$,
- $a \in AP, a \in Cl(\varphi)$, for all $a \in AP$,
- if $\psi \in Cl(\varphi)$, and $\psi \neq \neg\theta$, then $\neg\psi \in Cl(\varphi)$ (we identify $\neg\neg\psi$ with ψ),
- if $\neg\psi \in Cl(\varphi)$, then $\psi \in Cl(\varphi)$,
- if $\psi \wedge \theta \in Cl(\varphi)$, then $\psi, \theta \in Cl(\varphi)$,
- if $\bigcirc \psi \in Cl(\varphi)$, then $\psi \in Cl(\varphi)$,
- if $\psi \,\mathcal{U}\, \theta \in Cl(\varphi)$, then $\psi, \theta \in Cl(\varphi)$.

Elementary set of formulas

A set $B \subseteq Cl(\varphi)$ is called elementary if:

1. it is *consistent* with respect to propositional logic, i.e., for all $\varphi_1 \wedge \varphi_2, \psi \in Cl(\varphi)$:
   - $\varphi_1 \wedge \varphi_2 \in B$ if and only if $\varphi_1 \in B$ and $\varphi_2 \in B$.
   - $\psi \in B$ implies $\neg\psi \notin B$

- true $\in Cl(\varphi)$ implies true $\in Cl(\varphi)$.

2. it is *locally consistent* with respect to the until operator, i.e., for all $\varphi_1 \,\mathcal{U}\, \varphi_2 \in Cl(\varphi)$:
   - $\varphi_2 \in B$ implies $\varphi_1 \,\mathcal{U}\, \varphi_2 \in B$
   - $\varphi_1 \,\mathcal{U}\, \varphi_2 \in B$ and $\varphi_2 \notin B$ implies $\varphi_1 \in B$.
3. it is *maximal* i.e., for all $\psi \in Cl(\varphi)$:
   - $\psi \notin B$ implies $\neg\psi \in B$.

GNBA for LTL

Given a LTL formula φ over AP, we define as follows the GNBA $\mathcal{G}_\varphi$ which recognizes all and only the strings over the alphabet 2^{AP} that satisfy φ.

$\mathcal{G}_\varphi := (Q, 2^{AP}, \delta, Q_0, \mathcal{F})$ such that:

- $Q \subseteq 2^{Cl(\varphi)}$ is the set of all the elementary sets of formulae $B \subseteq Cl(\varphi)$,
- $Q_0 = \{B \in Q \mid \varphi \in B\}$,
- $\mathcal{F} = \{F_{\varphi_1 \mathcal{U} \varphi_2} \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \in Cl(\varphi)\}$, where
  - $F_{\varphi_1 \mathcal{U} \varphi_2} = \{B \in Q \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \notin B \text{ or } \varphi_2 \in B\}$.

The transition relation $\delta : Q \times 2^{AP} \to 2^Q$ is defined as follows; (given an elementary set B, i.e., a state of the automaton, and a set of AP A):

- if $A \neq B \cap AP$, then $\delta(B, A) = \varnothing$,
- if $A = B \cap AP$, then $\delta(B, A)$ is the set of all elementary sets of formulas B' such that
  1. for all $\bigcirc \psi \in Cl(\varphi)$, $\bigcirc \psi \in B$ if and only if $\psi \in B'$,
  2. for all $\varphi_1 \,\mathcal{U}\, \varphi_2 \in Cl(\varphi)$,

$$\varphi_1 \,\mathcal{U}\, \varphi_2 \in B \text{ if and only if } (\varphi_2 \in B \vee (\varphi_1 \in B \wedge \varphi_1 \,\mathcal{U}\, \varphi_2 \in B'))$$

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 $\neg\varphi$.

2.4 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.

2.5 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?

2.6 Example

write me

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.

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

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, $\models$ is defined for path formulas as follows:

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

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 $\models \Phi$, 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 $\models \Phi$ if and only if $I \subseteq Sat(\Phi)$.

The following algorithm sketches the procedure.

Algorithm 1 CTL Model Checking

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

function $SatFun(\phi)$
    if ϕ contains state subformulas **then**
        $Sat(\psi_1) = SatFun(\psi_1)$
        $Sat(\psi_2) = SatFun(\psi_2)$ for children nodes ψ_1, ψ_2 of the parse tree of ϕ, the so called *maximal proper subformulas*
        combine $Sat(\psi_1), Sat(\psi_2)$ depending on the operator of ϕ ($\wedge, \exists\bigcirc, \exists\mathcal{U}, \exists\Box$).
    else
        compute directly the set $Sat(\phi)$.
    end if
    return $Sat(\phi)$.
end function
return $I \subseteq Sat(\Phi)$.

Definition of $SatFun$

Task

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.