185.A05 Advanced Functional Programming SS 22

Tuesday, 05/24/2022
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 Model Checker, *MiniCheck*, for the so called Computational Tree Logic (CTL). The project consists of two parts: the core model checker and the elective modular extensions.

Part I - The MiniCheck Core. The core model checker requires:

- a number of preliminary programming tasks dealing with Transition Systems (TS) and Computational Tree Logic (CTL) (cf. Section 2.1 and 2.2).

- a verification module which checks an input CTL formula on a TS (cf. Section 2.3).

- a test suite allowing to validate the usability and correctness of the various components of the core (cf. Section 2.4).

An illustrating example highlights the essence of model-checking in our setting (cf. Section 2.5). Note that the core is mandatory but is only worth a portion of the points achievable in this project. To achieve the full amount of points some of the extensions have to be chosen and implemented, too.

Part II - The MiniCheck Extensions. The extensions of the MiniCheck core concern

- software model checking (cf. Section 3)

- bounded linear time logic model-checking (cf. Section 4)

Each extension is worth the shown amount of points. Note that none of the described extensions are necessary for a positive grade. Additionally, there is a bonus task allowing to achieve even more than the full amount of points for the project (cf. Section 5).

In Section 6, finally, the submission artefacts are detailed. In contrast to previous assignments, 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 two parsers at its core (cf. Section 2.1 and 2.2), the verification module (cf. Section 2.3) and the test suite (cf. Section 2.4) 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 (cf. Section 3 and 4) and the bonus task (cf. Section 5).

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 (200 P)

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 of 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].

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*,

- $\longrightarrow \subseteq S \times Act \times S$ 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, you can get rid of terminal states by defining 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 lightbulb) 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 2.5.

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 propositions 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 constrain 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 visited after the current one. In fact, 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 ($\forall$). Intuitively, $\exists\varphi$ holds in a state if there exists *some* path satisfying φ that starts in that state. Dually, $\forall\varphi$ holds in a state if *all* states starting in that state satisfy φ.

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 formulae follow the following syntax:

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

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

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

$$\Phi_1 \vee \Phi_2 \equiv \neg(\neg\Phi_1 \wedge \Phi_2)$$
$$\Phi_1 \rightarrow \Phi_2 \equiv \neg\Phi_1 \vee \Phi_2$$
$$\Phi_1 \leftrightarrow \Phi_2 \equiv (\Phi_1 \rightarrow \Phi_2) \wedge (\Phi_2 \rightarrow \Phi_1)$$
$$\Phi_1 \oplus \Phi_2 \equiv (\Phi_1 \wedge \neg\Phi_2) \vee (\Phi_2 \wedge \neg\Phi_1)$$

Likewise, we derive some well-known temporal modalities for path formulae:

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

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

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

$$\exists \Box \Phi \equiv \neg\forall \Diamond \neg\Phi$$
$$\forall \Box \Phi \equiv \neg\exists \Diamond \neg\Phi$$

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 formulae with derived operators into formulae with only base operators through these equivalence rules, you will have implemented support of derived operators for free (even if performances will suffer).

CTL Semantics

Given atomic proposition $a \in AP$, $TS = (S, Act, \rightarrow, I, AP, L)$, state $s \in S$, CTL state formulae Φ, Ψ, CTL path formula φ, the semantics of CTL is defined in terms of the following satisfaction relation $\vDash$ (pronounced "satisfies") for state s. We recall that $s \vDash \Phi$ if and only if Φ holds in s.

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

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

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

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 formulae such that formulae 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 formulae (the same reasoning for LTL APs applies).

Task

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.

Resources

[1, Paragraph 6.2 (in particular 6.1.1 and 6.1.2)]

2.3 The Model Checking procedure

The model checking procedure for CTL formulae differs completely from automata-based procedures, like in LTL verification. It's essentially 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 directly using s and TD. 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 $I_{TS} \subseteq Sat(\Phi)$.

2.3.1 Remark about the notation

In the following, we present a procedure which computes the satisfaction sets for formulae of CTL expressed in the so called *existential normal form*(ENF). This form is the smallest subset of CTL formulae that preserves the same expressive power of full CTL. In other words, for each CTL formula, there exists an equivalent CTL formula expressed in ENF. The set of formulae in ENF is given by:

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

We see that the $\forall$ operator is not present in CTL formulae in ENF. Again, you can either define by yourself the satisfaction set for all CTL formulae, or you can sacrify performances and translate the remaining formulae in EFN (which is the easier way). The translation rules can be found in [1, Paragraph 6.2.4]:

$$\forall\bigcirc\Phi \equiv \neg\exists\bigcirc\neg\Phi$$
$$\forall\big(\Phi\,\mathcal{U}\,\Psi\big) \equiv \neg\exists\big(\neg\Psi\,\mathcal{U}\,(\neg\Phi\wedge\neg\Psi)\big)\wedge\neg\exists\square\neg\Psi$$

For derived temporal operators, one can use:

$$\exists\lozenge\,\Phi \equiv \exists\big(\text{true}\,\mathcal{U}\,\Phi\big)$$
$$\forall\lozenge\,\Phi \equiv \neg\exists\square\neg\Phi$$
$$\forall\square\,\Phi \equiv \neg\exists\lozenge\neg\Phi \equiv \neg\exists\big(\text{true}\,\mathcal{U}\,\neg\Phi\big)$$

2.3.2 The algorithm

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 $TS \vDash \Phi$; otherwise, "no"

1: **function** $SatFun(\phi)$
2:     **if** ϕ contains state subformulae **then**
3:         // for children nodes ψ_1, ψ_2 of the parse tree of ϕ, the so called *maximal proper subformulae*
4:         $Sat(\psi_1) = SatFun(\psi_1)$; $Sat(\psi_2) = SatFun(\psi_2)$
5:         combine $Sat(\psi_1), Sat(\psi_2)$ depending on the operator of ϕ ($\wedge, \exists\bigcirc, \exists\mathcal{U}, \exists\square$).
6:     **else**
7:         compute directly the set $Sat(\phi)$.
8:     **end if**
9:     return $Sat(\phi)$.
10: **end function**
11: return $I \subseteq Sat(\Phi)$.

Characterization of $Sat(\cdot)$ for CTL formulae in ENF

We now formally define the set $Sat(\cdot)$ for every formula, possibly depending on the satisfaction set of subformulae. Let TS $= (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states. For all CTL formulae Φ, Ψ over AP it holds:

$Sat(\text{true}) := S$

$Sat(a) := \{s \in S \mid a \in L(s)\}$, for any a $\in AP$

$Sat(\Phi \wedge \Psi) := Sat(\Phi) \cap Sat(\Psi)$

$Sat(\neg\Phi) := S \setminus Sat(\Phi)$

$Sat(\exists \bigcirc \Phi) := \{s \in S \mid Post(s) \cap Sat(\Phi) \neq \varnothing$

$Sat\big(\exists(\Phi \, \mathcal{U} \, \Psi)\big)$ is the smallest subset T of S such that :

(1) $Sat(\Psi) \subseteq T$ and (2) $s \in Sat(\Phi)$ and $Post(s) \cap T \neq \varnothing$ implies $s \in T$

$Sat(\exists \Box \Phi)$ is the largest subset T of S, such that

(3) $T \subseteq Sat(\Phi)$ and (4) $s \in T$ implies $Post(s) \cap T \neq \varnothing$

where

$$Post(s) := \bigcup_{\alpha \in Act} \{s' \in S \mid s \xrightarrow{\alpha} s'\}$$

is just the set of successors of a state.

2.3.3 How to compute $Sat(\cdot)$

For most cases, you can just stick to the definition. There are only two critical situations. To compute $Sat\big(\exists(\Phi \, \mathcal{U} \, \Psi)\big)$ and $Sat(\exists \Box \Phi)$, an iterative approach is required to compute respectively the smallest subset and the greatest subset that verify the property. The two algorithms are sketched here.

Algorithm 2 Computation of the satisfaction set for Until formulae

---

    Input: finite transition system TS with state set S and CTL formula $\Phi_1 := \exists(\Phi \, \mathcal{U} \, \Psi)$ (both over AP)

    Output: $Sat(\Phi_1) = \{s \in S \mid s \vDash \Phi_1\}$

1: T $:= Sat(\Psi)$; (* compute the smallest fixpoint *)
2: **while** $\{s \in Sat(\Phi) \setminus T \mid Post(s) \cap T \neq \varnothing\} \neq \varnothing$ **do**
3:    $T := T \cup \{s \in Sat(\Phi) \setminus T \mid Post(s) \cap T \neq \varnothing\}$;
4: **end while**
5: return T;

2.3.4 Resources

[1, Paragraph 6.4]

Algorithm 3 Computation of the satisfaction set for Existential Always formulae
_____

    Input: finite transition system TS with state set S and CTL formula $\Phi_1 := \exists(\square\, \Phi)$ (both over *AP*)

    Output: $Sat(\Phi_1) = \{s \in S \mid s \vDash \Phi_1\}$

1: $T := Sat(\Phi)$; (* compute the greatest fixpoint *)
2: **while** $\{s \in T \mid Post(s) \cap T = \varnothing\} \neq \varnothing$ **do**
3:     $T := T \cup \{s \in T \mid Post(s) \cap T = \varnothing\}$;
4: **end while**
5: return T;
_____

2.3.5 Resources

[1, Paragraph 6.4]

Task

Implement the model checking algorithm presented here.

2.4 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 for the components of Section 2.1 and 2.2, and a test unit for the components of Section 2.3.

2.5 Example

We give now an example of a Transition System and some formulae that hold for it. The example has been taken from [1, Paragraph 2.1]. The following picture represents a preliminary design of a beverage vending machine. The machine can either deliver beer or soda. States are indicated by ovals and transitions by labelled edges. Every state is labelled with some description of the state itself. Initial states are indicated by having an incoming arrow without source.
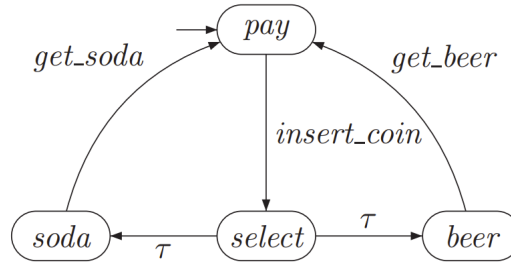
The following formulae hold for this TS (note, τ denotes an internal transition of the TS):

- $\{\forall \diamond \, \text{pay}\}$:= every path will eventually return to the initial state, which is labelled with "pay";

- $\{\exists \diamond \, \text{soda}\}$:= it exists a path which eventually reaches the granting of a soda;

- $\{\exists \square \,(\, \text{select} \implies (\forall \bigcirc \, \text{soda} \,))\}$:= it exists a path such that all selections are followed by the granting of a soda;

The following formula does not hold for the TS:

- $\{\forall(\diamond \, \text{soda})\}$:= every path with eventually reach the granting of a soda. This is not true, because the path which always selects the beer is feasible, and violates the specification.

3 Extension 1 - Parsing MINI^{--} Programs (100 P)

With this extension, you are required to extend the modelling power of MiniCheck targeting software verification. 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, we will implement an automated translation procedure for MINI^{--} programs into TS. MINI^{--} is a flattened version of the programming language MINI of the other project assignment of this course. In the following, we detail this version of MINI, called MINI^{--}, which is even smaller that MINI, and then present how to translate a MINI^{--} program into an equivalent Transition System.

3.1 Informal Description

This section informally describes the core of the MINI programming language, only covering the language core.

MINI is a procedural programming language. The syntax and semantics of MINI are inspired by the programming language C.

All code is contained in a single procedure `main` and variables are of the same type, namely the boolean type (Haskell type `Bool`). This means that the type of the variables is not explicitly specified in MINI. Variables are not required to be declared before definition.

The procedure `main` implements an impure function, where input/output functionality **is included in the core of MINI**. Procedure `main` takes a defined set of named variables and returns a result. The last statement of the procedure must be a `return` statement and no other `return` statement can occur in the procedure. In contrast to common programming languages, `return` statements in MINI are syntactically restricted to return the value of a single variable instead of a general expression.

The `if` statement is defined as in C. It executes either the first or optionally the second block.

It is not necessary to declare variables before usage, however, it is necessary to define their value before usage, i.e. a variable must be initialised before it can be used in boolean expressions.

Regarding the variables in the argument lists, you need to consider all possible combinations of assignments of values for them, as we will see shortly with the definition of the semantics of a MINI program.

Modern programming languages usually have a way to interact with a user via terminal or file system. We consider two statements: one that prints a boolean and one that reads a boolean value. We will see that, while these statements are a major issue for the interpreter, and it's hard to define their semantics in terms of a pure evaluation function, in the model checking problem they can be treated just as a source of non-determinism, which is already there in the type of TS we are using.

3.2 Syntax

$$
\begin{aligned}
program &:= \texttt{procedure main}(argument\_vars)\texttt{\{}procedure\_body\texttt{\}} \\
argument\_vars &:= var \mid var\texttt{,}argument\_vars \\
procedure\_body &:= statements \quad return\_stmt \\
\\
statements &:= \varepsilon \mid statement \quad statements \\
statement &:= if\_stmt \mid assign\_stmt \mid print\_bool\_stmt \mid read\_bool\_stmt \\
if\_stmt &:= \texttt{if}(bool\_expr)\texttt{\{}statements\texttt{\}} \\
&\quad \mid \texttt{if}(bool\_expr)\texttt{\{}statements\texttt{\}else\{}statements\texttt{\}} \\
assign\_stmt &:= var\texttt{=}bool\_expr\texttt{;} \\
print\_bool\_stmt &:= \texttt{print\_bool}(bool\_expr)\texttt{;} \\
read\_bool\_stmt &:= var\texttt{=read\_bool();} \\
bool\_expr &:= bool\_expr\_nested \mid \neg bool\_exp\_nested \\
&\quad \mid bool\_expr\_nested \quad relator \quad bool\_expr\_nested \\
bool\_expr\_nested &:= bool \mid var \mid (bool\_expr) \\
relator &:= \wedge \mid \vee \mid \implies \mid \iff \mid \oplus \\
return\_stmt &:= \texttt{return} \ var\texttt{;} \\
bool &:= \texttt{true} \mid \texttt{false} \\
var &:= ident \\
ident &:= char \quad ident\_rest \\
ident\_rest &:= \varepsilon \mid ident\_char \quad ident\_rest \\
ident\_char &:= char \mid digit \mid \_ \\
char &:= \texttt{a} \mid \ldots \mid \texttt{z} \\
digit &:= \texttt{0} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

Additional whitespace is allowed everywhere except within identifier names, integer literals, the binary comparison operators, and the keywords (`procedure`, `return`, `if`, `else`, `while`).

3.3 Semantics

We define now the semantics of a MINI program P in terms of a Transition System, through the so-called **structural operational semantics (SOS)**. With this semantics, we mimic the step-by-step execution of the program through the transition relation of a Transition System. We have to follow the inference rules of the SOS to construct the state set and the transition relation of the corresponding TS.

A state of the TS corresponds to the current program counter (i.e., the next instruction to be executed), and the current valuation of all the already occurred program variables. Formally, a state is given by $\langle P, \gamma \rangle$, where program P is just viewed as a sequence of statements (i.e., the remaining program to be executed).

3.3.1 The set of initial states

The set of initial state of a Transition System I is composed by all possible combinations of initial boolean values for the set of arguments of the MINI program P. This can be represented in set notation by $2^{arg(P)}$, where arg is the set of arguments of P. Informally a subset of arg contains all and only the variables that are evaluated to true in the current evaluation. The set of initial states is then defined as:

$$I := \{\langle P, \gamma \rangle \mid \gamma \in 2^{arg(P)}\}$$

3.3.2 The Transition Relation

The transition relation is then defined by the following transition rules, where the symbol $\downarrow$ indicates a state where there is no more program to parse.

$$\frac{\gamma \vDash \varphi}{\langle \text{if}(\varphi)\{\text{B}\};\ \text{P}, \gamma \rangle \rightarrow \langle\ \text{B;P}\ , \gamma \rangle}$$

$$\frac{\gamma \nvDash \varphi}{\langle\ \text{if}(\varphi)\{\text{B}\};\text{P}, \gamma \rangle \rightarrow \langle \text{P}, \gamma \rangle}$$

$$\frac{\gamma \vDash \varphi}{\langle \text{if}(\varphi)\{\text{B}\}\text{else}\{\text{E}\};\ \text{P}, \gamma \rangle \rightarrow \langle\ \text{B;P}\ , \gamma \rangle}$$

$$\frac{\gamma \nvDash \varphi}{\langle \text{if}(\varphi)\{\text{B}\}\text{else}\{\text{E}\};\ \text{P}, \gamma \rangle \rightarrow \langle\ \text{E;P}\ , \gamma \rangle}$$

$$\frac{\gamma \vDash Expr \mapsto v}{\langle x := Expr; P, \gamma \rangle \rightarrow \langle P, \gamma[x \mapsto v] \rangle}$$

$$\frac{}{\langle print\_bool(expr); P, \gamma \rangle \rightarrow \langle P, \gamma \rangle}$$

$$\frac{}{\langle x = read\_bool; P, \gamma \rangle \rightarrow \langle P, \gamma[x \mapsto T] \rangle \text{ and } \langle x = read\_bool; P, \gamma \rangle \rightarrow \langle P, \gamma[x \mapsto F] \rangle}$$

$$\frac{}{\langle return\ x; \gamma \rangle \rightarrow \langle \downarrow, \gamma \rangle}$$

$$\frac{}{\langle \downarrow, \gamma \rangle \rightarrow \langle \downarrow, \gamma \rangle}$$

We use the notation $\gamma\{x \mapsto v\}$ to denote the state that is obtained when updating the value of the variable x to v.

The model checker aborts the verification of invalid programs. A program is invalid if a variable x is not defined but used in an expression, which, in our semantics, corresponds to the fact that we are not able to establish, given an expression *Expr* over program variables which includes x, if $\gamma \models Expr$, because there is no value for x saved in γ.

A Note about terminal states

The last rule has been added due to a technical problem. While MINI programs are guaranteed to terminate for any input, the Transition Systems we have considered so far are defined over infinite runs, and require that there are no terminal states, i.e. that every state has at least one outgoing edge. To accommodate the two formalisms, we introduce a self loop in the final state of the program, which induces the required infinite run in the Transition System.

3.4 Examples

A procedure calculating the difference of two numbers:

```
procedure main(a, b) {
    c = a − b;
    return c;
}
```

A procedure calculating the minimum of two numbers:

```
procedure main(a, b) {
    if (a < b) {
        c = a;
    } else {
        c = b;
    }
    return c;
}
```

4 Extension 2 - Bounded LTL Model Checking (100 P)

With this extension, you are required to extend the verification power of MiniCheck targeting a different class of properties, namely, those expressed through the so called *Linear Temporal Logic*. LTL embraces a *linear* view of the time, which means that at each moment of time there is only one possible successor state and thus each moment has a unique possible future. This follows from the fact that LTL is path-based, i.e. LTL formula are interpreted only in terms of sequences of states. The interpretation of a LTL formula φ over a state s requires that φ is satisfied by all possible computations (paths) that start in s.

The full LTL algorithm requires some more involved formalisms, hence we will not deal with it. Instead, we will implement a **bounded** model checking algorithm.

To do so, we enumerate all possible paths that start from initial states with size up to a certain size k (where the bound k is provided by the user), and verify only these paths against the input formula. This means that the overall procedure is not complete, i.e. we are not formally guaranteeing that the formula holds for the TS at hand. However, this approach has some advantages: it is simple to apply and it is much more scalable.

As modelling formalism, we will use the same as the core modules, i.e. Transition Systems without terminal states, and where Actions are irrelevant.

In the following, we will provide some information about LTL, and then present the tasks to be implemented with this extension.

Syntax

Given a set AP of atomic propositions, with $a \in AP$, LTL formulae 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 again the following *derived* Boolean operators:

$$\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \varphi_2)$$
$$\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$$
$$\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$$
$$\varphi_1 \oplus \varphi_2 \equiv (\varphi_1 \wedge \neg\varphi_2) \vee (\varphi_2 \wedge \neg\varphi_1)$$

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

$$\Diamond\varphi \equiv \text{true} \, \mathcal{U} \, \varphi$$
$$\Box\varphi \equiv \neg\Diamond\neg\varphi$$

You are required to handle all and only these operators.

Semantics

We define the semantics of LTL formulae 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 $\sigma = A_0 A_1 A_2 \ldots$ be an infinite word over 2^{AP}.

Then $\vDash$ is defined by:

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

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.

Semantics over non maximal paths

While the semantics of LTL is defined over infinite paths, with our approach we deal with **finite** paths. Since the TS has no terminal state, these also have the property to be **non maximal**, i.e. they can always be prolonged, since every state has at least one successor. This raises an issue with the semantics of the $\mathcal{U}$ operator we have defined, since it's an unbounded operator. With the current semantics, we may not be able to establish if a finite path σ of size k satisfies $\varphi_1 \, \mathcal{U} \, \varphi_2$, since the index j such that $\sigma[j\ldots] \vDash \varphi_2$ may be greater than the bound k. Therefore, we have to restrict the semantics of our LTL formulae to force the satisfaction of φ_2 whithin at most k steps. This leads us to:

$$\sigma \vDash_k \varphi_1 \, \mathcal{U} \, \varphi_2 \text{ iff } \exists j.0 \leqslant j < k \text{ and } \sigma[j\ldots(k-1)] \vDash_k \varphi_2 \text{ and } \sigma[i\ldots(k-1)] \vDash_k \Phi_1, \text{ for all } 0 \leqslant i < j$$

Likewise, we update the semantics of the other operators for coherence.

$$\sigma \vDash_k \text{ true}$$
$$\sigma \vDash_k a \text{ iff } a \in A_0$$
$$\sigma \vDash_k \varphi_1 \wedge \varphi_2 \text{ iff } \sigma \vDash_k \varphi_1 \text{ and } \sigma \vDash \varphi_2$$
$$\sigma \vDash_k \neg\varphi \text{ iff } \sigma \nvDash_k \varphi$$
$$\sigma \vDash_k \bigcirc\varphi \text{ iff } \sigma[1\ldots(k-1)] = A_1 \ldots A_{k-1} \vDash_k \varphi$$

Task 1 - Parsing LTL

Define:

- a suitable plain-text representation of LTL formulae such that formulae 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 formulae.

Implement:

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

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.

Task 2 - Bounded LTL Model Checking

We now outline the algorithm to be implemented. With a naive description, it is divided into three steps. Given a path $\pi := s_0 \ldots s_k$, we define its corresponding trace $\sigma := L(s_0) \ldots L(s_k)$ as $\sigma(\pi)$

Algorithm 4 k-bounded Model Checking

Input: finite transition system TS with initial states I, LTL formula φ (both over AP), and bound k

Output: "yes" if $TS \vDash_k \Phi$; otherwise, "no"

$\Pi := \{\pi := s_0 \ldots s_{k-1} \mid s_0 \in I \text{ and } \varpi \text{ is a valid path in TS}\};$
$\Sigma := \{\sigma(\pi) \mid \pi \in \Pi;$
if $\forall \sigma \in \Sigma (\sigma \vDash_k \varphi)$ **then**
    return "yes"
else
    return "no"
end if

Resources

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

5 Bonus Task: Project Management (25 P)

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.

5.1 Command-Line Interface

The program so far only has very rudimentary argument parsing, allowing a single or two filepath(s). 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 should be parsed for correctness of the syntax of the input file, but no formula is verified on it.

- On `--extensions`, a list of supported MINI extensions is printed.

- Other flags as you see appropriate. (optional, not graded)

5.2 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.

5.3 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`

5.4 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?

6 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.

6.1 Project Implementation

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

6.2 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.

- 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.