185.A05 Advanced Functional Programming SS 25

Tuesday, 6 May 2025

Assignment 4

Model Checking Project: MiniCheck

All Chapters

Topic: Building a CTL Model Checker plus Implementing a Modelling extension

Submission deadline: Friday, 13th June 2025, noon (no continuous update)

Contact: prof. Jens Knoop, jens.knoop@tuwien.ac.at,
Francesco Pontiggia, francesco.pontiggia@tuwien.ac.at
Lou Elah Süsslin, lou.suesslin@student.tuwien.ac.at

(In case of need, this deadline will be extended (details posted via TUWEL))

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 an extension.

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).

Part II - The MiniCheck Extension. The extension of the MiniCheck core concerns software model checking (cf. Section 3).

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 the extension has to be implemented, too. However, it is not 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 4).

In Section 5, the submission artefacts are detailed. In contrast to previous assignments, you are not asked to implement functions with (often) 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 300 points. To gain more points on top of the aforementioned, you will have to implement the modelling extension of the MiniCheck tool (see Section 3) or the bonus task (cf. Section 4). At most 400 points are awarded for the project implementation. The project presentation (Demo) will be worth up to 200 points.

The total points for the project are calculated by adding up the points on the core, the selected extensions and the presentation (for a maximum of 600 points). For a positive evaluation of this project at least 300 points are required. A positive grade on the project is necessary in order to receive a positive grade in this course. A positive grade on the core itself is *not* required.

In contrast to previous exercises, there is no possibility for a continuous update until the project demo.

1.1 Note

We do not assume an in-depth knowledge of model checking and its purpose, nor logics. Some previous knowledge in related fields helps, but it's not mandatory, as all used formalisms are introduced. This includes also:

- Automata Theory
- Graph Algorithms
- Propositional Logics

See, e.g., [1] for deeper background details.

2 MiniCheck Core (300 P)

Model Checking consists of formally and exhaustively verifying a formula expressing some property on all the possible executions of a particular type of automata (the *model*). The model might be an abstract representation of a hardware component, or even a piece of software. 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.

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. 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*,
- $\rightarrow \subseteq S \times Act \times S$ 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*.

The labelling function maps every state to the set of atomic propositions holding in that state. Informally, atomic propositions are just labels that carry information about states. The initial state is chosen nondeterministically between all states $\in I$. The transition system operates as follows: after having chosen an initial state $s \in I$, it chooses nondeterministically a transition $s \xrightarrow{\alpha} s'$, executes the action α and moves on to $s' \in S$. Note that a state may have outgoing transitions going to different states, but also 1) outgoing transitions to different states but associated with the same action, and 2) outgoing transitions to the same state but with different actions. A *run* of the TS is a valid **infinite** sequence of interleaved states and actions, where valid just means that for each triple $s\alpha s'$ in the run the TS has a transition $s \xrightarrow{\alpha} s'$.

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 add a transition to a "sink" state s_{sink}, and then define a self transition from s_{sink} to itself.

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. Then, in the following, we will consider runs as sequence of states (*paths*), and forget about actions.

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. Atomic Propositions (or atomic formulae) are boolean variables that represent either a description of the state (e.g., OFF and ON 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, see the example in Section 2.5.).

Implement:

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

The parsing function should abort when it encounters a TS non complying with the syntax you defined. Besides syntactic checks, we require you to implement some additional checks to detect non well-formed TS. Make sure that 1) there are no terminal states (or add the sink state), 2) there is at least an initial state, and 3) that your formalisation of the set of atomic propositions is respected.

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 consider 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 (CTL)

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, to reason about the temporal ordering of events.

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 that can be visited in a run from an initial state. They are operators part of the so called Computation Tree Logic (CTL). Note here the difference between temporal ordering and timing: in temporal logics time is seen as a discrete sequence of time instances (or snapshots, if you like), and not as a continuous variable. With these formalism, you do not specify something like "A for two seconds, then B for one second", but rather "A, and in the next time instance B".

In particular, CTL (differently with respect to some other logics, e.g. LTL) is based on a *branching* notion of time. Branching time refers to the fact that every step in a path of the TS is associated with a choice between different successor states, which is a split between several different futures. All the possible paths that arise from a state can then be represented as a tree of states of infinite depth, where each choice of a successor in a state is between all the subtrees generated by all the successor states. By fixing a successor for every state in the tree, we get one of all possible futures, i.e., a single path, a traversal of the TS.

Computation Tree Logic (CTL) thus allows to express properties that must hold for *some* or *all* paths starting 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 starting in that state satisfying φ. Dually, $\forall\varphi$ holds in a state if *all* paths starting in that state satisfy φ.

CTL Syntax

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

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

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.

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 $\models$ (read "satisfies").

For state s and CTL (state) formula φ, $s \models \Phi$ if and only if Φ holds in s.

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

Given a path π (a sequence of states), $\models$ is defined for path formulae 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 all initial states of the TS satisfy Φ.

2.2.1 Derived Operators

In addition, we introduce the following well-known *derived* Boolean operators from propositional logic for state formulae.

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

We derive also some more temporal modalities for path formulae:

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

Intuitively, $\Diamond$ means *"at some point in the path"*, and $\Box$ means *"always in the path"*.

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

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

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

We give an informal notion of some of these formulae. The left hand side of the first equivalence means *"there exists a path such that at some point* Φ *holds"*. The right hand side of the second equivalence means *"on all paths, anything holds until* Φ *holds"*.

You are required to handle also derived operators.

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).

Task 2

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. Note that they should match the APs of the transition system.

Note

It is up to you to decide how to combine the two plain text representations of the inputs (the TS and the formula) (e.g., both in the same file, in two separate files …), and how to provide them to the Model Checker via some command-line arguments (e.g., the folder path, two paths, …). 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/logging you may find useful. (e.g. "Parsing formula from file …")

Resources

For a complete definition of CTL, [1, Paragraph 6.2 (in particular 6.1.1 and 6.1.2)].

2.3 The Model Checking procedure

Given a TS $\mathcal{A}$ and a CTL formula Φ we want to verify whether $\mathcal{A} \models \Phi$. In order to do so, we have to establish whether Φ is satisfied in each initial state s of $\mathcal{A}$. The model checking procedure is roughly composed of two steps:

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

In other words, $\mathcal{A} \models \Phi$ if and only if $I_{\mathcal{A}} \subseteq Sat(\Phi)$. The model checking procedure is essentially a bottom-up traversal of the parse tree of Φ - we compute for each subformula its satisfaction set, and then compose them bottom up to compute $Sat(\Phi)$.

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). The set of formulae in ENF is given by:

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

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 translate the remaining formulae in EFN (which is the easier way). The translation rules are ([1, Paragraph 6.2.4]):

$$\begin{aligned} \forall \bigcirc \Phi &\equiv \neg\exists \bigcirc \neg\Phi \\ \forall\big(\Phi \,\mathcal{U}\, \Psi\big) &\equiv \neg\exists\big(\neg\Psi \,\mathcal{U}\, (\neg\Phi \wedge \neg\Psi)\big) \wedge \neg\exists \Box \neg\Psi \end{aligned}$$

2.3.2 The algorithm

The following algorithm sketches the procedure.

Algorithm 1 CTL Model Checking

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

```
function SatFun(φ)
    if φ contains state subformulae then
        // for children nodes ψ1, ψ2 of the parse tree of φ
        Sat(ψ1) = SatFun(ψ1); Sat(ψ2) = SatFun(ψ2)
        combine Sat(ψ1), Sat(ψ2) depending on the operator of φ (∧, ∃○, ∃𝒰, ∃□)
        // see the characterization below
    else
        compute directly the set Sat(φ).
    end if
    return Sat(φ).
end function

SatSet := SatFun(Φ)
return I_A ⊆ SatSet.
```

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

We now formally define the set $Sat(\cdot)$ for every formula. 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:

$$
\begin{aligned}
Sat(\text{true}) &:= S \\
Sat(a) &:= \{s \in S \mid a \in L(s)\}, \text{ for any a } \in AP \\
Sat(\Phi \wedge \Psi) &:= Sat(\Phi) \cap Sat(\Psi) \\
Sat(\neg\Phi) &:= S \setminus Sat(\Phi) \\
Sat(\exists \bigcirc \Phi) &:= \{s \in S \mid Post(s) \cap Sat(\Phi) \neq \varnothing\} \\
Sat\big(\exists(\Phi\, \mathcal{U}\, \Psi)\big) &\text{ is the smallest subset T of S such that:} \\
&\quad (1)\ Sat(\Psi) \subseteq T \text{ and} \\
&\quad (2)\ s \in Sat(\Phi) \text{ and } Post(s) \cap T \neq \varnothing \text{ implies } s \in T \\
Sat(\exists \Box \Phi) &\text{ is the largest subset T of S such that:} \\
&\quad (3)\ T \subseteq Sat(\Phi) \text{ and} \\
&\quad (4)\ s \in T \text{ implies } Post(s) \cap T \neq \varnothing
\end{aligned}
$$

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 satisfy the respective conditions. 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 $\Phi_1 := \exists(\Phi \,\mathcal{U}\, \Psi)$.
Output: $Sat(\Phi_1) = \{s \in S \mid s \vDash \Phi_1\}$.

```
1: T := Sat(Ψ); (* compute the smallest fixpoint *)
2: while {s ∈ Sat(Φ) \ T | Post(s) ∩ T ≠ ∅} ≠ ∅ do
3:     T := T ∪ {s ∈ Sat(Φ) \ T | Post(s) ∩ T ≠ ∅};
4: end while
5: return T;
```

Intuitively speaking, at iteration i of the while loop, the set T_i contains all states that can reach a Ψ-state in at most i steps via a path passing only through Φ states.

Algorithm 3 Computation of the satisfaction set for Existential Always formulae

Input: finite transition system TS with state set S and CTL formula $\Phi_1 := \exists(\Box\, \Phi)$.
Output: $Sat(\Phi_1) = \{s \in S \mid s \vDash \Phi_1\}$.

```
1: T := Sat(Φ); (* compute the greatest fixpoint *)
2: while {s ∈ T | Post(s) ∩ T = ∅} ≠ ∅ do
3:     T := T \ {s ∈ T | Post(s) ∩ T = ∅};
4: end while
5: return T;
```

2.3.4 Detailed Algorithms for Until and Existential Always formulae

The following algorithm is a potential implementation in a "backward" manner of the computation of $Sat(\cdot)$ for Until formulae, assuming your representation of the Transition System allows to quickly access the predecessors of a state, defined as

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

The basic idea is to compute $Sat\big(\exists\,(\Phi \,\mathcal{U}\, \Psi)\big)$ by means of the iteration:

$$T_0 = Sat(\Psi) \text{ and } T_{i+1} = T_i \cup \{s \in Sat(\Phi) \mid Post(s) \cup T_i \neq \varnothing\}$$

until a fixpoint is reached, i.e. until $T_i = T_{i+1}$.

Algorithm 4 Enumerative backward search for computing $Sat\big(\exists\,(\Phi\,\mathcal{U}\,\Psi)\big)$

Input: finite transition system TS with state set S and CTL formula $\Phi_1 := \exists(\Phi\,\mathcal{U}\,\Psi)$
Output: $Sat(\Phi_1) = \{s \in S \mid s \vDash \Phi_1\}$

```
1:  E := Sat(Ψ);
2:  T := E;
3:  while E ≠ ∅ do
4:      let s' ∈ E;
5:      E := E\{s'};
6:      for s ∈ Pre(s') do
7:          if s ∈ Sat(Φ)\T then
8:              E := E ∪ {s};
9:              T := T ∪ {s};
10:         end if
11:     end for
12: end while
13: return T;
```

Likewise, the following algorithm is a potential implementation in a "backward" manner of the computation of $Sat(\cdot)$ for Existential Always formulae. The basic idea is to compute $Sat(\exists\,\Box\,\Phi)$ by means of the iteration:

$$T_0 = Sat(\Phi) \text{ and } T_{i+1} = T_i \cap \{s \in Sat(\Phi) \mid Post(s) \cup T_i \neq \varnothing\}$$

until a fixpoint is reached. The algorithm makes use of an additional counter vector *count*.

Algorithm 5 Enumerative backward search for computing $Sat(\exists\Box\,\Phi)$

Input: finite transition system TS with state set S and CTL formula $\Phi_1 := \exists\Box\,\Phi$
Output: $Sat(\Phi_1) = \{s \in S \mid s \vDash \Phi_1\}$

```
1:  E := S \ Sat(Φ);
2:  T := Sat(Φ);
3:  for s ∈ Sat(Φ) do
4:      count[s] := |Post(s)|;
5:  end for
6:  while E ≠ ∅ do
7:      let s' ∈ E;
8:      E := E \ {s'};
9:      for s ∈ Pre(s') do
10:         if s ∈ T then
11:             count[s] := count[s] − 1;
12:             if count[s] == 0 then
13:                 T := T \ {s};
14:                 E := E ∪ {s};
15:             end if
16:         end if
17:     end for
18: end while
19: return T;
```

2.3.5 Resources

[1, Paragraph 6.4]

Task 3

- Define a suitable data type reperesentation of CTL formulae in Existential Normal Form.
- Implement a function converting CTL formulae not in ENF into ones in ENF.
- Implement the CTL model checking Algorithm 1 presented here by using, if you like, Algorithms 4 and 5.

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:

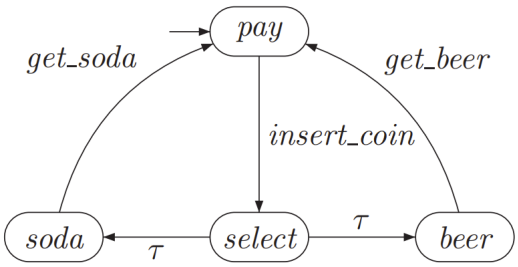
- `hspec`, a tutorial can be found `here`
- `tasty`
- `HTF`
- `HUnit`

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:

$$\forall \Diamond \text{pay} := \textit{“every path will eventually return to the “pay” state.”}$$
$$\exists \Diamond \text{soda} := \textit{“there is a path which eventually reaches the granting of a soda.”}$$
$$\exists \Diamond \text{soda} := \textit{“there is a path which eventually reaches the granting of a soda.”}$$
$$\forall \Box(\text{select} \implies (\forall \bigcirc \neg\text{select})) := \textit{“on every path, a “select” cannot be followed by another “select”.”}$$

The following formula does not hold for the TS:

$$\forall(\Diamond \text{ soda}) := \textit{“every path with eventually reach the granting of a soda.”}$$

This is not true, because the path which always selects the beer is feasible, and violates the specification. Also

$$\exists \Box \big(\text{select} \implies (\forall \bigcirc \text{ soda })\big) := \textit{“there is a path where all “select” states can be followed by “soda” states only.”}$$

is not true, because the only “select” state in the model can be followed by both a “soda” and a “beer” state.

3 Extension - 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 programs written in a toy programming language, MINI^{--}, into TS.

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. This means that the type of the variables is not explicitly specified in MINI^{--}. Variables are not required to be declared before definition. 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.

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. Regarding the variables in the argument lists, you need to consider all possible combinations of assignments of values for them.

The last statement of the procedure must be a `return` statement and no other `return` statement can occur in the procedure. The `if` statement is defined as in C. It executes either the first or optionally the second block.

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. 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 Transition Systems we are using.

3.2 Syntax

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

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 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 γ. Formally, a state is given by $\langle \texttt{P}, \gamma \rangle$, where program `P` is just viewed as a sequence of statements.

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 $2^{arg(\mathtt{P})}$, where arg is the set of arguments of P. 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:

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

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 statement to evaluate.

$$\frac{\varphi \notin \gamma}{\langle\ \texttt{if(}\varphi\texttt{)\{B\};P},\ \gamma\rangle \rightarrow \lightning} \qquad \frac{\varphi \notin \gamma}{\langle\ \texttt{if(}\varphi\texttt{)\{B\}else\{E\};P},\ \gamma\rangle \rightarrow \lightning}$$

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

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

$$\frac{\gamma \vDash (bool\_expr \mapsto v)}{\langle var\texttt{=}bool\_expr\texttt{;P},\ \gamma\rangle \rightarrow \langle \texttt{P},\ \gamma[var \mapsto v]\rangle} \qquad \frac{bool\_expr \notin \gamma}{\langle var\texttt{=}bool\_expr\texttt{;P},\ \gamma\rangle \rightarrow \lightning}$$

$$\frac{bool\_expr \in \gamma}{\langle\texttt{print\_bool(}bool\_expr\texttt{);P},\ \gamma\rangle \rightarrow \langle \texttt{P}, \gamma\rangle} \qquad \frac{bool\_expr \notin \gamma}{\langle\texttt{print\_bool(}bool\_expr\texttt{);P},\ \gamma\rangle \rightarrow \lightning}$$

$$\frac{var \in \gamma}{\langle var\texttt{=read\_bool();P},\ \gamma\rangle \rightarrow \langle \texttt{P},\ \gamma[var \mapsto T]\rangle} \qquad \frac{var \in \gamma}{\langle var\texttt{=read\_bool();P},\ \gamma\rangle \rightarrow \langle \texttt{P},\ \gamma[var \mapsto F]\rangle}$$

$$\frac{bool\_expr \in \gamma}{\langle\texttt{return}\ bool\_expr\texttt{;},\ \gamma\rangle \rightarrow \langle\downarrow,\ \gamma\rangle} \qquad \frac{bool\_expr \notin \gamma}{\langle\texttt{return}\ bool\_expr\texttt{;},\ \gamma\rangle \rightarrow \lightning}$$

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

We use the notation $\gamma[var \mapsto v]$ to denote the state that is obtained when updating the value of the variable var to v.

A statement is invalid if it uses a variable var that has not been defined before. We use notation $var \notin \gamma$ to indicate that variable var has no binding in the current

state. With a slight abuse of notation, we use $bool\_expr \notin \gamma$ or $\varphi \notin \gamma$ to indicate that a variable with no binding occurs in $bool\_expr$ or φ. An invalid statement makes the whole program invalid. This is represented in our semantics with a transition to an error state indicated with $\lightning$.

Note about terminal states

The last two rules have been added due to a technical problem. While MINI^{--} programs are guaranteed to terminate for any input or to reach the error state, the Transition Systems we have considered don't have 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 states of the program, which induces the required infinite runs in the Transition System.

Note about model checking

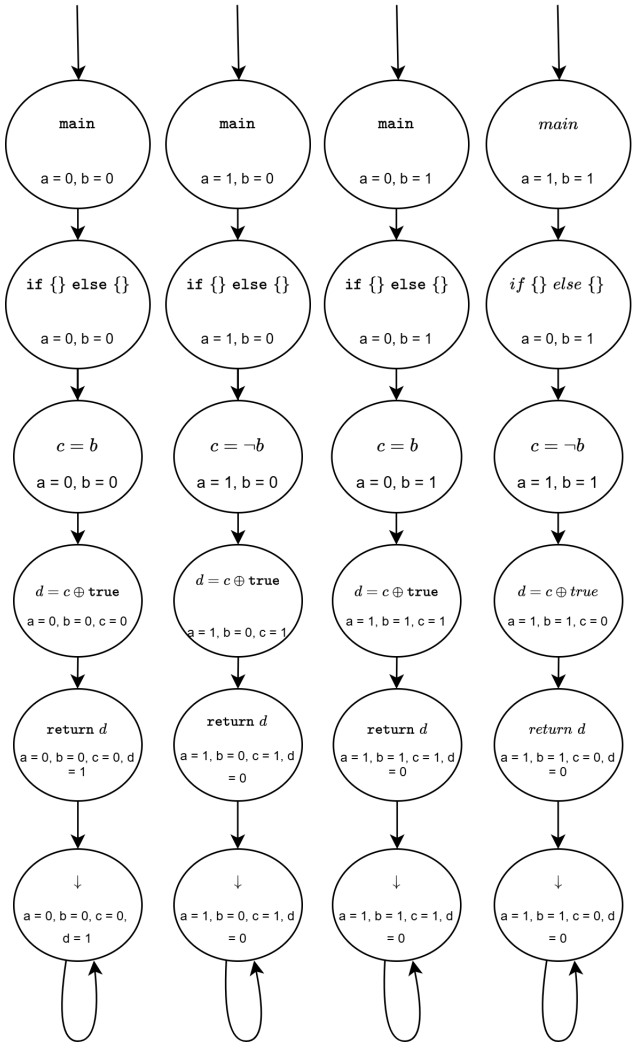
When verifying properties like *"Variable* $\mathtt{x}$ *must be false after one step for all paths"*, we encounter the problem that variable `x` may not be initialized after one step (i.e., $\mathtt{x} \notin \gamma$ at that point of the execution). In such cases, MiniCheck assumes `x` to be false. In other words, a variable is considered by the model checking procedure to be false until it is set to be true. There is not need to distinguish in model checking between an unitialized variable and a false one. You need however to do it in the semantics of MINI^{--}, where it makes a difference.

3.4 Example

A procedure over two inputs:

```
procedure main(a, b) {
    if (a) { c =  ¬ (b);
    } else { c = b; }
    d = c  ⊕  true;
    return d;
}
```

The corresponding Transition System, where initial states are at the top. An example formula on this program is $\exists \Diamond(return \wedge d)$, which simply means *"there is a path returning value 1"*. This formula holds for this TS because of the leftmost path. Also $\forall \Diamond return$ holds.



4 Bonus Task: Project Management (50 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.

4.1 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)

4.2 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.

4.3 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`

4.4 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?

5 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.

5.1 Project Implementation

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

5.2 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?
- Is the extension 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.