

185.A05 Advanced Functional Programming SS 24

Tuesday, 30/04/2024

Assignment 5

Model Checking Project: MiniCheck

All Chapters

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

Submission deadline: Monday, 10th June 2023, noon (no continuous update)

Contact: prof. Jens Knoop, jens.knoop@tuwien.ac.at,
Francesco Pontiggia, francesco.pontiggia@tuwien.ac.at
Dominik Leko, dominik.leko@tuwien.ac.at

Tim Dirr, tim.dirr@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 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).

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)

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. However, 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, 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 275 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](#)). 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 a 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 (a PDF version is available on the [Github repository](#)).

2 MiniCheck Core (275 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*). 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 $s\alpha s'$ 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. In the following, we will consider runs as sequence of states (*paths*).

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). We refer to 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 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 (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 \odot (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 part of the so called Computation Tree Logic (CTL).

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, where each choice in a state is between all the subtrees generated by all the successor states.

By fixing a choice 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 for *some* or *all* paths 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 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:

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

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.

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

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

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

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

You are required to handle all and only these 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).

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 (pronounced “satisfies”).

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

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

Given a path π (a sequence of states), \models is defined for path formulae as follows:

$\pi \models \bigcirc \Phi$	iff $\pi[1] \models \Phi$
$\pi \models \Phi \mathcal{U} \Psi$	iff $\exists j \geq 0. (\pi[j] \models \Psi \wedge (\forall 0 \leq k < j. \pi[k] \models \Phi))$

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. 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 you find useful that may result from parsing (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:

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

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

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

For derived temporal operators, one can use:

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

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”

```
1: function  $SatFun(\phi)$ 
2:   if  $\phi$  contains state subformulae then
3:     // for children nodes  $\psi_1, \psi_2$  of the parse tree of  $\phi$ 
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  $\phi$  ( $\wedge, \exists \bigcirc, \exists \mathcal{U}, \exists \square$ )
6:     // see the characterization below
7:   else
8:     compute directly the set  $Sat(\phi)$ .
9:   end if
10:  return  $Sat(\phi)$ .
11: end function
12:  $SatSet := SatFun(\Phi)$ 
13: return  $I_{\mathcal{A}} \subseteq SatSet$ .
```

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:

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

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 $Sat(\exists(\Phi \mathcal{U} \Psi))$ and $Sat(\exists \square \Phi)$, 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: $Sat(\Phi_1) = \{s \in S \mid s \models \Phi_1\}$

```

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

```

Algorithm 3 Computation of the satisfaction set for Existential Always formulae

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

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

```

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

```

2.3.4 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:

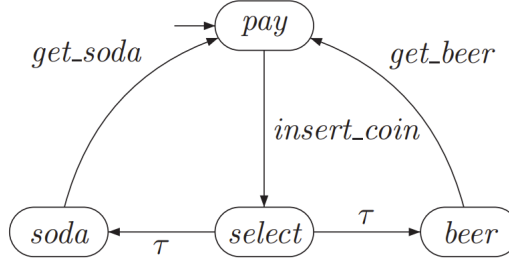
- [hspec](#), a tutorial can be found [here](#)
- [tasty](#)
- [HTF](#)
- [HUnit](#)

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;

The following formula do 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.
- $\{\exists \Box (\text{select} \implies (\forall \bigcirc \text{soda}))\} :=$ it exists a path such that, every time a selection is encountered, all paths starting from it release a soda. This does not hold.

3 Extension 1 - Parsing MINI⁻⁻ Programs (125 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. 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.

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 TS we are using.

3.2 Syntax

$$\begin{aligned}
\text{program} &:= \text{procedure main}(\text{argument_vars})\{\text{procedure_body}\} \\
\text{argument_vars} &:= \text{var} \mid \text{var}, \text{argument_vars} \\
\text{procedure_body} &:= \text{statements} \quad \text{return_stmt} \\
\\
\text{statements} &:= \varepsilon \mid \text{statement} \quad \text{statements} \\
\text{statement} &:= \text{if_stmt} \mid \text{assign_stmt} \mid \text{print_bool_stmt} \mid \text{read_bool_stmt} \\
\text{if_stmt} &:= \text{if}(\text{bool_expr})\{\text{statements}\} \\
&\quad \mid \text{if}(\text{bool_expr})\{\text{statements}\}\text{else}\{\text{statements}\} \\
\text{assign_stmt} &:= \text{var} = \text{bool_expr}; \\
\text{print_bool_stmt} &:= \text{print_bool}(\text{bool_expr}); \\
\text{read_bool_stmt} &:= \text{var} = \text{read_bool}(); \\
\text{bool_expr} &:= \text{bool_expr_nested} \mid \neg \text{bool_expr_nested} \\
&\quad \mid \text{bool_expr_nested} \quad \text{relator} \quad \text{bool_expr_nested} \\
\text{bool_expr_nested} &:= \text{bool} \mid \text{var} \mid (\text{bool_expr}) \\
\text{relator} &:= \wedge \mid \vee \mid \implies \mid \iff \mid \oplus \\
\text{return_stmt} &:= \text{return} \text{ var}; \\
\text{bool} &:= \text{true} \mid \text{false} \\
\text{var} &:= \text{ident} \\
\text{ident} &:= \text{char} \quad \text{ident_rest} \\
\text{ident_rest} &:= \varepsilon \mid \text{ident_char} \quad \text{ident_rest} \\
\text{ident_char} &:= \text{char} \mid \text{digit} \mid - \\
\text{char} &:= \text{a} \mid \dots \mid \text{z} \\
\text{digit} &:= 0 \mid \dots \mid 9
\end{aligned}$$

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 $\langle P, \gamma \rangle$, 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 $2^{arg(P)}$, 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:

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

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 evaluate.

$$\begin{array}{c}
\frac{\varphi \notin \gamma}{\langle \text{if } (\varphi)\{B\}; P, \gamma \rangle \rightarrow \downarrow} \qquad \frac{\varphi \notin \gamma}{\langle \text{if } (\varphi)\{B\} \text{ else } \{E\}; P, \gamma \rangle \rightarrow \downarrow} \\
\\
\frac{\gamma \models \varphi}{\langle \text{if } (\varphi)\{B\}; P, \gamma \rangle \rightarrow \langle B; P, \gamma \rangle} \qquad \frac{\gamma \not\models \varphi}{\langle \text{if } (\varphi)\{B\}; P, \gamma \rangle \rightarrow \langle P, \gamma \rangle} \\
\\
\frac{\gamma \models \varphi}{\langle \text{if } (\varphi)\{B\} \text{ else } \{E\}; P, \gamma \rangle \rightarrow \langle B; P, \gamma \rangle} \qquad \frac{\gamma \not\models \varphi}{\langle \text{if } (\varphi)\{B\} \text{ else } \{E\}; P, \gamma \rangle \rightarrow \langle E; P, \gamma \rangle} \\
\\
\frac{\gamma \models (Expr \mapsto v)}{\langle x := Expr; P, \gamma \rangle \rightarrow \langle P, \gamma[x \mapsto v] \rangle} \qquad \frac{Expr \notin \gamma}{\langle x := Expr; P, \gamma \rangle \rightarrow \downarrow} \\
\\
\frac{Expr \in \gamma}{\langle \text{print_bool}(expr); P, \gamma \rangle \rightarrow \langle P, \gamma \rangle} \qquad \frac{Expr \notin \gamma}{\langle \text{print_bool}(Expr); P, \gamma \rangle \rightarrow \downarrow} \\
\\
\frac{x \notin \gamma}{\langle x = \text{read_bool}; P, \gamma \rangle \rightarrow \downarrow} \\
\\
\frac{x \in \gamma}{\langle x = \text{read_bool}; P, \gamma \rangle \rightarrow \langle P, \gamma[x \mapsto T] \rangle} \qquad \frac{x \in \gamma}{\langle x = \text{read_bool}; P, \gamma \rangle \rightarrow \langle P, \gamma[x \mapsto F] \rangle} \\
\\
\frac{x \in \gamma}{\langle \text{return } x; \gamma \rangle \rightarrow \langle \downarrow, \gamma \rangle} \qquad \frac{x \notin \gamma}{\langle \text{return } x; \gamma \rangle \rightarrow \downarrow} \\
\\
\frac{}{\langle \downarrow, \gamma \rangle \rightarrow \langle \downarrow, \gamma \rangle} \qquad \frac{}{\langle \downarrow, \gamma \rangle \rightarrow \langle \downarrow, \gamma \rangle}
\end{array}$$

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 .

A statement is invalid if it uses a variable x that has not been defined before. We use notation $x \notin \gamma$ to indicate that variable x has no binding in the current state. With a slight abuse of notation, we use $Expr \notin \gamma$ or $\varphi \notin \gamma$ to indicate that a variable with no binding occurs in $Expr$ 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 \perp .

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 have 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 states of the program, which induces the required infinite runs in the Transition System.

3.4 Example

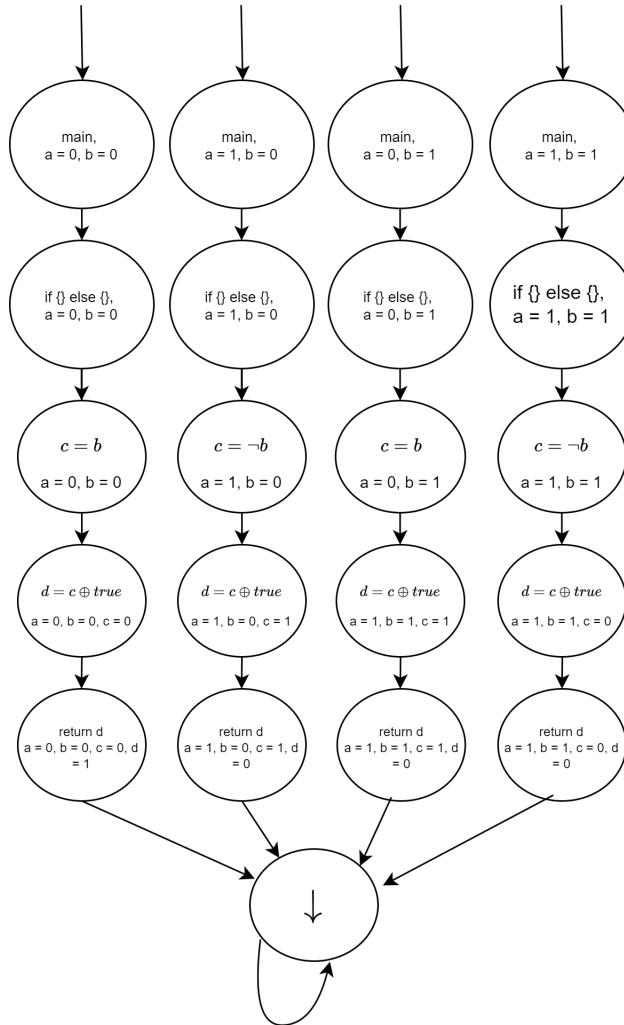
A procedure over two inputs:

```

procedure main(a, b) {
  if (a) { c =  $\neg$  (b);
  } else { c = b; }
  d = c  $\oplus$  true;
  return d;
}

```

The corresponding Transition System, where initial states are at the top:



4 Extension 2 - Bounded LTL Model Checking (125 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 is path-based, i.e. LTL formula are interpreted only in terms of sequences of states, i.e. paths. Differently with respect to CTL, LTL embraces a *linear* view of the time. 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 . There are no path quantifiers \exists or \forall .

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:

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

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

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

You are required to handle all and only these operators.

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 $s_0 s_1 s_2 \dots$ (i.e. a path) of the TS, its trace is just the induced infinite sequence of atomic propositions $\sigma = L(s_0)L(s_1)\dots$. Let $\sigma = A_0 A_1 A_2 \dots$ be an infinite sequence of labels. We use notation $\sigma[i \dots j]$ to indicate the subsequence $A_i \dots A_j$. Then \models is defined by:

$$\begin{aligned}
 \sigma &\models \text{true} \\
 \sigma &\models a \text{ iff } a \in A_0 \\
 \sigma &\models \varphi_1 \wedge \varphi_2 \text{ iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
 \sigma &\models \neg \varphi \text{ iff } \sigma \not\models \varphi \\
 \sigma &\models \bigcirc \varphi \text{ iff } \sigma[1 \dots](= A_1 \dots) \models \varphi \\
 \sigma &\models \varphi_1 \mathcal{U} \varphi_2 \text{ iff } \exists j \geq 0. \sigma[j \dots] \models \varphi_2 \text{ and } \sigma[i \dots] \models \Phi_1, \text{ for all } 0 \leq i < j
 \end{aligned}$$

An LTL formula φ holds in a path π if it holds for $\text{trace}(\pi)$. 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 $\sigma[j \dots] \models \varphi_2$ 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 within at most k steps. This leads us to:

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

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

$$\begin{aligned}
 \sigma &\models_k \text{true} \\
 \sigma &\models_k a \text{ iff } a \in A_0 \\
 \sigma &\models_k \varphi_1 \wedge \varphi_2 \text{ iff } \sigma \models_k \varphi_1 \text{ and } \sigma \models_k \varphi_2 \\
 \sigma &\models_k \neg \varphi \text{ iff } \sigma \not\models_k \varphi \\
 \sigma &\models_k \bigcirc \varphi \text{ iff } \sigma[1 \dots (k-1)](= A_1 \dots A_{k-1}) \models_k \varphi
 \end{aligned}$$

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 $\pi := s_0 \dots s_k$, we define its corresponding trace $\sigma := L(s_0) \dots L(s_k)$ 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 $TS \models_k \Phi$; otherwise, “no”

$\Pi := \{\pi := s_0 \dots s_{k-1} \mid s_0 \in I \text{ and } \pi \text{ is a valid path in TS}\};$

$\Sigma := \{\sigma(\pi) \mid \pi \in \Pi\};$

if $\forall \sigma \in \Sigma (\sigma \models_k \varphi)$ **then**
 return “yes”

else

 return “no”

end if

Resources

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

A note for those interested

CTL and LTL in general are *uncomparable*, which means that there are some CTL formulae that cannot be expressed in LTL, and viceversa. However, after having fixed a finite bound k , our bounded version LTL_k could be entirely expressed in CTL, and hence this extension is not adding any expressivity to the tool.

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.