MiniCheck Project Documentation

June 13, 2024

Florian Freitag
11908096

Johannes Blaha
11930322

Contents

1. Introduction

MiniCheck is a command line program Model checker which can verify computational tree models.

2. Building MiniCheck

MiniCheck is built with [cabal version 3.10.3]() and [GHC 9.4.8]().

Before building the program we first need to install the dependencies, all further chapters will assume that these two commands were run at least once so that all dependencies are already installed.

```
cabal update
cabal build miniCheck
```

The binary is now in an obscure subdirectory, however the build command should have printed the output location. Otherwise you can also locate the binary with `cabal list-bin miniCheck`.

3. Usage

In general the program can be invoked by locating the binary an executing it with the required arguments. However, it is more convenient to let cabal figure out the location of the binary with the `cabal run` command. This requires us to separate the cabal arguments from the arguments passed on to MiniCheck which is what the double dash `--` does in the following examples.

```
# Check a model
cabal run miniCheck -- "./examples/ts.txt" "./examples/ctl.txt"

# Verify if the transition system is correct
cabal run miniCheck -- "./examples/ts.txt" "./examples/ctl.txt" --ts

# List available extensions
cabal run miniCheck -- extensionmode --extensions

# Evaluate a Mini-- program
cabal run miniCheck -- minimmmode "examples/minimm.txt" "examples/
ctl_minimm.txt"

# Verify a bounded LTL with a depth of 5
cabal run miniCheck -- ltlmode "examples/ts.txt" "examples/ltl.txt" 5

# List more options
cabal run miniCheck -- --help
```

Our implementation features both extensions, Mini`--` and bounded *Linear Temporal Logic* (LTL) model checking .

3.1. Examples

There are a few examples provided in the `examples` folder which is located in the root directory of the project. They can be used to test the program and also provide a convenient starting point to create new custom examples. The provided examples are:

- `ts.txt:` A simple example of a transition system
- `ctl.txt:` A short computation tree logic formula
- `minimm.txt:` A simple Mini-- program
- `clt_minimm.txt` A simple ctl forumlar that makes sense on the Mini-- example
- `ltl.txt:` A minimal linear temporal logic model

A set of more complex inputs can be found out by reading the tests.

4. Dependencies and Haskell language extensions

MiniCheck depends on the the following packages from hackage:

Package	Version	Description
`base`	4.17.2.1	Core data structures and operations.
`parsec`	3.1.17.0	A monadic parser framework.
`mtl`	2.3.1	A package with monadic utility functions.
`containers`	0.7	A collection on containers like Hashsets and Hashmaps.
`cmdargs`	0.10.22	A command line argument parsing library.

Additionally, the following packages are required for testing:

Package	Version	Description
`raw-strings-qq`	1.1	Allows to use multiline string literals.
`hspec`	2.11.8	A testing framework for haskell.

MiniCheck also uses the following Haskell language extensions:

- DeriveDataTypeable
- BlockArguments
- QuasiQuotes (only in the tests for multiline string literals)

5. Structure

The project is structured into multiple modules, where each major component is split into a different module. In detail this means that the core is split into three major modules and the Mini-- and LTL extensions are split into two modules each:

- **Core:** The main program that allows the verify a Computation Tree Logic formula of a Transition System
  - *Transition System:* This module is responsible for parsing the plain text representation of the transition system
  - *Computational Tree Logic:* This module is responsible for parsing the plain text representation of the computational tree logic
  - *Model Checking:* This module does the actual verification process and outputs True/False given a TS and a CTL
- **Extension 1 - Mini--** Parse and verify program that are written in the Mini-- programming language
  - *Mini--:* The main Mini-- module that is responsible for parsing the Mini-- language into an *Abstract Syntax Tree* (AST).
  - *Mini--Compiler:* This module compiles a Mini-- AST into a Transition System.
- **Extension 2 - Bounded LTL Model Checking** Parse and verify bounded LTL Models.
  - *Linear Temporal Logic:* The module specifying the abstract representation of the logic and a parser generate it from a textual representation.
  - *Model Checking LTL:* Since LTL is a different kind of logic it needs it's own algorithms to verify models which are implemented here.

6. Testing

All tests can be executed with:

```
cabal test --test-show-details=direct
```

Most tests are closer to integration tests than unit tests. For example we test the parsers for the TS, CTL, Mini-- and LTL on their own but only verify if the output produced is correct and not internal details of the parser.

However, by having a large set of inputs for each of those parsers we eventually verify all aspects of each parser. Moreover, this approach allowed us to build up the test suite while we were developing the parser even though large structural parts of the parser weren't yet final.

Besides the parsers we also test the model checking and Mini-- compilation to TS. In the model checker we create a TS and CTL in each test. We specify the expected (correct) answer and verify that the checker also reaches the same conclusion. The tests for the specialised LTL Model checker are similar but of course use a LTL instead of the CTL.

The Mini-- to TS compiler is quite a bit more difficult to test, mainly because the resulting TS are so incredibly large. For example the following code creates a TS that in code form would be **312KiB** large and not only would it exceed any reasonable limit for the tests but also be so large that no human can ever verify that it is correct.

```
procedure main(a) {
     a = read_bool();
     b = read_bool();
     c = read_bool();
     d = read_bool();
     e = read_bool();
     return e;
 }
```

While testing against expected output is not quite possible we did find some ways to ensure that the output is probably correct.

Since compilation can fail, if an undefined variable is read a group of tests only checks that the compilation fails in those cases but not in cases where the input is correct.

The next group of tests checks if the number of nodes and transition is correct but not if all nodes are correctly connected.

The compiler also evaluates all Mini-- expressions. To ensure that those results are correct we create programs with many expressions which do not depend on the input and check if in all return nodes in the compiled TS the variable has the expected value.

None of the compilation tests check whether the nodes are correctly connected, however there are quite a few model checking tests which invoke the compiler and would fail if the compiler wouldn't connect the nodes correctly.

7. Known Limitations

7.1. Command Line Arguments

Because of some limitations on the chosen command line parser library, we couldn't quite match the API from the assignment. For example the assignment states that `MiniCheck --extensions` should list all implemented extensions but `cmdargs` would still expect the file paths for the input files which in this case of course doesn't make any sense.

We solved that by introducing sub commands, which we call modes and therefore the same command becomes `MiniCheck extensionmode --extensions` instead. In Chapter 3 we already documented all our flags which slightly differ from the assignment.

7.2. Mini-- undefined variables

While Mini-- has the concept of undefined variables in the Transition System doesn't have a comparable concept and each atomic property must be true or false.

Therefore it is not possible to write computational tree logic queries that probe the *definedness* of variables and all undefined variables will be compiled into atomic properties with the value false in the Transition System.