CHAPTER 1

User Manual

In this appendix we provide a brief guide on how to download, compile and run *PMModelChecker* on a Linux environment. The guide will terminate with some practical examples.

1.1 Dependencies

In order to use PMModelChecker you will need the following tools:

* `g++`, from the GNU Compiler Collection (https://gcc.gnu.org/);
* GNU Make (`make`) (https://www.gnu.org/software/make/);
* Flex (`flex`) (the Fast Lexical Analyzer, https://github.com/westes/flex);
* GNU Bison (`bison`, https://www.gnu.org/software/bison/), a general-purpose parser generator;
* Python >2.7 (the Python interpreter, https://www.python.org/).

Moreover, you can utilise the following tools for automatically generating transition systems and parity games, which subsequently will serve as input to PMModelChecker:

* mCRL2, obtainable at https://www.mcrl2.org/web/user_manual/index.html (for transition systems);
* PGSolver, obtainable at https://github.com/tcsprojects/pgsolver (for parity games).

1.2 Obtaining the sources

You can obtain the source code for PMModelChecker from

https://github.com/Cyofanni/PMModelChecker

Download the latest sources with the following command, from any location on your system:

```
git clone https://github.com/Cyofanni/PMModelChecker
```

The `git clone` command will create a directory named `PMModelChecker`. Then launch

```
cd PMModelChecker
```

The root folder contains the following subdirectories:

* `src/`, which includes the C++ source files;
* `include/`, which includes the header files;
* `scripts/`, which contains some Python scripts written for preprocessing tasks;
* `tests/`, which contains some instances of equational systems, syntax files, parity games and transition systems.

1.3 Customising the operators

Stay in the project's root directory and define the syntax of the language you will use to write your equational systems, using the format described in Chapter 5. So, let `customsyntax` be the name of the file that includes your language's syntax. Then run

```
python scripts/syntax_generator.py customsyntax
```

After running this command, the couple of files `src/mylex.l` and `src/myparser.yacc` will contain the input needed by Flex and Bison to generate a scanner and a parser for your language.

1.4 The Preprocessing Phase

The equational system you want to solve must undergo a *preprocessing* phase, so as to speed up the tool's parser. Your system can be preprocessed in the following way:

```
python scripts/preprocessor.py [basis_file] [original_system] \
                        [preprocessed_system]
```

The first command line argument (`[basis_file]`) should be a file containing the names you want to give to the basis elements. Each name should be separated by a newline character. Thus, a possible `basis_file` for a basis of size three must have this content:

```
name_of_item_0
name_of_item_1
name_of_item_2
```

1.5 Compiling *PMModelChecker*

The root folder of the project contains a *Makefile*, which includes all the instructions needed to compile the final tool. Now run

1. `make parser`
2. `make`
3. `make clean`

1.6 Running *PMModelChecker*

The `make` command will generate a single executable (in the project's root directory), named `pm_model_checker`, which is to be invoked from the command line in the way explained in the following subsections.

1.6.1 The *General* Mode

In this mode, the symbolic $\exists$-move for each simple operator should be provided explicitly by the user.

```
./pm_model_checker -ge [equational_system] [basis_size] \
             [lattice_height] [basis_file]
```

1.6.2 The μ-*calculus* Mode

When the tool is run in this mode, the symbolic $\exists$-moves for μ-calculus' $\Box$ and $\Diamond$ operators are computed automatically from a file representing the LTS in the *Aldebaran* format.

```
./pm_model_checker -mu [equational_system] [basis_size] \
         [lts_file] [lattice_height] [basis_file]
```

1.6.3 The *Normaliser* Mode

PMModelChecker includes a *normaliser*, that is to say, a component that can translate a system of equations containing composite operators into an equivalent (larger) system of normalised equations. The normaliser can be invoked in the following way:

```
./pm_model_checker -normalize [equational_system] [output_file]
```

1.7 Examples

1.7.1 Example: a boolean system

A very basic example can be found under the directory `tests/example1`. The file `system_and_moves` defines the system

$$\begin{cases} x_0 =_\mu x_0 \wedge x_1 \\ x_1 =_\nu x_0 \vee x_1 \end{cases}$$

which is a boolean system of two equations with alternating fixpoints, on the lattice `{false, true}`, with basis equal to `{true}`. Evidently its solution is the tuple $[u_0 = false, u_1 = true]$. As you already know, the solver implemented by PMModelChecker relies on Progress Measures defined in terms of symbolic $\exists$-moves. Therefore, any input should contain a move for each operator appearing in the equations, which in our case are

$$\phi^{and}_{true} = [true, 0] \wedge [true, 1]$$

$$\phi^{or}_{true} = [true, 0] \vee [true, 1]$$

So, let us try to run our tool on the system contained in `system_and_moves`, written in the language defined in the file `syntax`. First and foremost, we should generate the specification files needed by Flex and Bison:

```
python scripts/syntax_generator.py tests/example1/syntax
```

Then, from the root directory, launch:

1. `make parser`
2. `make`
3. `make clean`

Now, you need to preprocess the input by means of the following command (from the root directory):

```
python scripts/preprocessor.py tests/example1/basis \
          tests/example1/system_and_moves \
        tests/example1/prep_system_and_moves
```

Now we are ready to solve the system. Observe that in this case we should opt for the *general* mode, because of the fact that we are dealing with a boolean system, and moves for boolean operators have already been supplied. Thus, launch

```
./pm_model_checker -ge tests/example1/prep_system_and_moves \
                  1 2 tests/example1/basis
```

Some helpful notes on the command line arguments:

1. `tests/example1/prep_system_and_moves` contains the preprocessed equational system, i.e. the output generated previously thanks to the script named `preprocessor.py`;
2. `1` is the basis size of the boolean lattice `{false, true}`, which has only one item, `true`. For theoretical reasons, the bottom element of a lattice (which is `false` in our case) cannot be part of the basis;
3. `2` represents the height of the boolean lattice: it is needed for halting the fixpoint computation after a certain amount of iterations;
4. `tests/example1/basis` contains the names of the basis items: it is needed for printing the final result in a readable way.

Some notes on the final output: at the end of the computation, you should see the following lines on the screen:

```
PROGRESS MEASURE MATRIX IS:
[DON'T KNOW (*)][true <= u_1]
```

The so-called "Progress Measure Matrix" (also referred to as `R`) contains all the clues we need in order to infer the structure of the solution. Since in this example we are dealing with a system of two equations over a lattice with a basis of size one, the theory tells us that Progress Measure values will be distributed on a 1×2 matrix. Rows will be denoted by the corresponding basis items, whereas columns by the zero-based equation indices. Clearly, the matrix output in the current example has one row, corresponding to the only basis item (`true`), and two columns. Now consider each matrix cell:

* `R(true)(0) = [DON'T KNOW (*)]` : we know that whenever $R(b)(i) = *$ (* represents the top of the Progress Measure lattice), we cannot derive any

information on (b, i), i.e. if $\mathbf{u}$ is the solution of the system, we cannot conclude that $b \not\sqsubseteq u_i$. In this case, $b = true$, $i = 0$, and we know from the beginning that $u_0 = false$. Obviously, there is no way to conclude that $true \sqsubseteq false$;

* `R(true)(1) = [true <= u_1]` : here the computation did not reach the top value. Then we can safely infer that $b \sqsubseteq u_1$, where $b = true$ and $u_1 = true$. This is definitely congruent with the fact that $true \sqsubseteq true$.

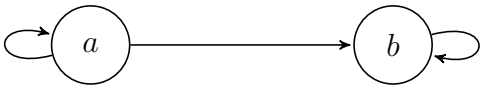
Notice that we could have shown the actual values for each matrix cell (which would have been arrays of naturals), but there was no point doing this, since a matrix of *YES/NO*-like values is fit for our purpose of understanding the solution.

1.7.2 Example: μ-calculus equations

The directory `tests/example2` contains an example based on μ-calculus. The file `system_and_moves` defines the system

$$\begin{cases} x_0 =_\mu p_0 \vee \Diamond x_0 \\ x_1 =_\nu x_0 \wedge \Box x_1 \end{cases}$$

which is equivalent to the μ-calculus formula $\phi = \nu x_1.((\mu x_0.(p_0 \vee \Diamond x_0)) \wedge x_1)$. We want to verify such formula with respect to the unlabelled transition system depicted below, where the propositional variable p_0 is true on the state named b and false otherwise:



The initial system of equations must be solved on the lattice $2^{\mathbb{S}}$, where $\mathbb{S} = \{a, b\}$. The set of singletons $\{\{a\}, \{b\}\}$ constitutes the basis $B_{2^{\mathbb{S}}}$. A specification for this transition system can be found in `tests/example2/transsystem.aut`, which is written in the *Aldebaran* format and can be easily generated with the mCRL2 toolset.
Clearly, the solution to our system is $[u_0 = \mathbb{S}, u_1 = \mathbb{S}]$ (corresponding to the fact that ϕ holds in every state).
Now, open the file `tests/example2/system_and_moves` and take a look at the set of moves, which is:

$$\begin{gathered} \phi^{and}_{\{a\}} = [\{a\}, 0] \wedge [\{a\}, 1] \\ \phi^{or}_{\{a\}} = [\{a\}, 0] \vee [\{a\}, 1] \\ \phi^{and}_{\{b\}} = [\{b\}, 0] \wedge [\{b\}, 1] \\ \phi^{or}_{\{b\}} = [\{b\}, 0] \vee [\{b\}, 1] \\ \phi^{p_0}_{\{a\}} = false \end{gathered}$$

$$\phi^{p_0}_{\{b\}} = true$$

Observe that we deliberately omitted the basic moves for the remaining operators, namely, $\Diamond$ and $\Box$, since they will be computed automatically from the transition system when running the tool in the *μ-calculus* mode. Please notice also that propositional variables (p_0 in this example) can be considered zero-arity operators with associated trivial basic moves (*true* or *false* atoms, according to the basis item).
Now you can carry out the usual steps.
First, generate the specification files for Flex and Bison:

```
python scripts/syntax_generator.py tests/example2/syntax
```

Then launch the usual build commands:

1. `make parser`
2. `make`
3. `make clean`

Preprocess the input:

```
python scripts/preprocessor.py tests/example2/basis \
          tests/example2/system_and_moves \
        tests/example2/prep_system_and_moves
```

So far, even the preprocessed input (`tests/example2/prep_system_and_moves`) lacks the $\Diamond$ and $\Box$ moves. Now we are ready to launch the tool in the *μ-calculus* mode:

```
./pm_model_checker -mu tests/example2/prep_system_and_moves 2 \
     tests/example2/transsystem.aut 2 tests/example2/basis
```

When run with the `-mu` flag, PMModelChecker reads the transition system specification (`tests/example2/transsystem.aut`), generates the $\Diamond$ and $\Box$ moves and appends them to the equational system file (`tests/example2/prep_system_and_moves`); notice that nothing changes in the non-preprocessed system. We recommend you to preprocess the system again whenever you want to solve it with the *μ-calculus* mode. Let us recap the meaning of each argument:

1. `tests/example2/prep_system_and_moves`, contains the preprocessed equational system, i.e. the output generated previously thanks to `preprocessor.py`;
2. `2` is the basis size of the powerset lattice $2^{\mathbb{S}}$, which has two items, $\{a\}$ and $\{b\}$;

3. `tests/example2/transsystem.aut` is the transition system in the Aldebaran format;
4. `2` is the height of the powerset lattice $2^{\mathbb{S}}$;
5. `tests/example2/basis` contains the names of the basis items.

A quick look at the output (`<=` represents the $\subseteq$ operator):

```
PROGRESS MEASURE MATRIX IS:
    [a <= u_0][a <= u_1]
    [b <= u_0][b <= u_1]
```

Everything is consistent with the known solution, indeed:

* $\{a\} \subseteq u_0 = \{a, b\}$;
* $\{a\} \subseteq u_1 = \{a, b\}$;
* $\{b\} \subseteq u_0 = \{a, b\}$;
* $\{b\} \subseteq u_1 = \{a, b\}$.

1.7.3 Example: running the normaliser

Under `tests/example3` you can find the file `complex_system`, where, as the name suggests, equations contain composite operators. You may wish to normalise it. Launch the command

```
./pm_model_checker -normalize tests/example3/complex_system \
                tests/example3/normalized_system
```

The file `tests/example3/normalized_system` contains the normalised system, where the right-hand side of each equation has just one operator.

1.7.4 Example: generating a system from a parity game

Now we show you how to produce a system of fixpoint boolean equations starting from a parity game written in the format used by PGSolver. The game we used to run this example can be found under `tests/example4`.
Firstly, sort the file according to the second column, which stores priorities:

```
sort -k 2 tests/example4/parity_game_1.gm > \
    tests/example4/parity_game_1_sorted.gm
```

Then, run the converter script (`game_to_system.py`) on the sorted game:

```
python scripts/game_to_system.py tests/example4/parity_game_1_sorted.gm \
                    tests/example4/boolean_system 0
```

The last command line argument could be equal either to 0 or to 1, and has the following meaning:

* 0: we are interested in player 0's win;
* 1: we are interested in player 1's win.