Manual

MARKOV REWARD MODEL CHECKER

Version 1.5

November 9, 2010

Authors:
Ivan S. Zapreev
Christina Jansen


Contents

1 Introduction

Model checking is an automated technique that establishes whether certain qualitative properties such as deadlock-freedom or request-response requirements ("does a request always lead to a response?") hold in a model of the system under consideration. Such models are typically transition systems that specify how the system can evolve during execution. Properties are usually expressed in temporal extensions of propositional logic, such as CTL [CES86].

In the last years adapting model checking to probabilistic systems has been a rather active research field. This has resulted in efficient algorithms for model-checking DTMCs and CTMCs, as well as Markov decision processes (MDPs), that are supported by several tools nowadays such as E $\vdash$MC^2 [HKMKS00], PRISM [HKNP06], GreatSPN [BDH00], VESTA [SVA05], Ymer [You05b], and the APNN Toolbox [BFKT03]. Various case studies have proven the usefulness of these model checkers. Popular logics are Probabilistic CTL (PCTL) [HJ94] and Continuous Stochastic Logic (CSL) [BHHK03].

Although these model checkers are able to handle a large set of measures of interest, the reward-based measures have received scant attention so far. Markov Reward Model Checker (MRMC) allowes for verification of Markov *reward* models (MRMs), in particular DMRMs and CMRMs. These are the underlying semantic models of various high-level performance modelling formalisms, such as reward extensions of stochastic process algebras, stochastic reward nets, and so on.

MRMC [KZH$^+$09], see also [JKO$^+$07, KZ09], supports the following types of probabilistic models:

- Discrete time Markov chains (DTMCs)
- Continuous time Markov chains (CTMCs)
- Discrete time Markov Reward models (DMRMs)
- Continuous time Markov Reward models (CMRMs)
- Continuous time Markov decision processes (CTMDPIs[1])

Hence, MRMC supports *Probabilistic Computation Tree Logic (PCTL)* and *Continuous Stochastic Logic (CSL)* for property specification as well as their reward extensions *Probabilistic Reward Computation Tree Logic (PRCTL)* and *Continuous Stochastic Reward Logic (CSRL)*. Table 1.1 provides correspondence between the before-mentioned logics and the supported models.

For PCTL the realized algorithms are mostly discussed by Hansson and Jonsson in [HJ94]. The exception is a long-run operator which is handled similar to the steady-state operator of

[1] Here, I stands for the internal non-determinism.

	DTMC	*CTMC*	*DMRM*	*CMRM*	*CTMDPI*
PCTL	+				
CSL		+			+ [a]
PRCTL			+		
CSRL				+	

[a] There is currently no support for the steady-state and unbounded-time reachability properties.

Table 1.1: The supported models and the corresponding logics

CSL. The supported algorithms for PRCTL have been described by Andova *et al.* [AHK03]. Model-checking techniques for CSL (on CTMCs) are derived from [BHHK03] and for its reward extension CSRL from [CKKP05] (see also [BHHK00, HCH+02]). For the latter one we have implemented two algorithms for time- and reward- bounded until formulae. One is based on discretization [TV00] and another on uniformization and path truncation [QS96]. The algorithms for PRCTL and CSRL support both state and impulse rewards. Model-checking of CSL (on CTMDPIs) implements procedures described in [BHKH05, BHH+06, BFK+09].

It is important to note that the model-checking procedures integrated in MRMC were complemented with the following extensions that are aimed at improving the tool's performance and accuracy:

Steady-state (long-run) operator of CSL (PCTL). For the operator $\mathrm{S}_{\bowtie b}(\Psi)$ the algorithmic improvement lies with searching only for BSCCs that can contain Ψ states, as opposed to searching for all BSCCs. The modification that was done to the model-checking algorithms is straightforward and therefore we do not explain it in further details.

Unbounded-until operator of CSL (PCTL). For model checking $\mathrm{P}_{\bowtie b}(\Phi\ \mathrm{U}\ \Psi)$, we first exclude states, using graph reachability analysis, from which Ψ states are always or never reachable. Then the model checking procedure for the remaining states is carried out as usual. All techniques required for this improvement are described in [CG04].

Time-bounded until operator of CSL. We have implemented a uniformization procedure [BHHK03] with a precise on-the-fly steady-state detection which is discussed in [KZ05, KZ06]. Similar to unbounded-until operator, the technique of [CG04] is employed to detect and remove states from which the Ψ states are never reached. Also we employ ideas, described in [KKNP01], that allow to compute the reachability probabilities for all initial states at once.

Bisimulation minimization. The bisimulation minimization algorithms have been realized for PCTL, CSL, PRCTL and CSRL, in the latter two cases without impulse rewards. For more details consider [KKZJ07].

Model checking by discrete event simulation. We developed and implemented algorithms for model-checking CSL properties by simulation of finite-state CTMCs. Our

approach is based on Monte Carlo simulation and derivation of confidence intervals. We provide statistical algorithms for model checking the most interesting CSL operators, such as steady-state, unbounded-reachability, and time-interval reachability operators. For more details we refer to [Zap08, KZ09].

The remainder of the manual is organized as follows. In Chapter 2 we discuss platforms supported by MRMC, the implementation language and licensing. Further, we illustrate the tool usage and introduce a snapshot of MRMC architecture via simple examples. The next chapter, Chapter 3, explains the installing process of the tool. The input-file formats of MRMC are discussed in Chapter 4. Chapter 5 is devoted to command-line options provided by the tool, while in Chapter 6 a list of all available MRMC commands and run-time options is given. The semantics of all supported logics are introduced in Chapter 7 and afterwords information about model checking by means of simulation is given in Chapter 8. Chapter 9 speaks about MRMC's test suite, while Chapter 10 concludes with the list of groups involved in the MRMC development and the corresponding contact information.

2 MRMC tool description

MRMC [KZH+09] is a command-line tool that supports an easy input format and is realized in the C programming language. The latter allows the tool to be small and fast due to compiler-based optimisations and smart memory management within the implementation [JKO+07]. Also, MRMC uses simple but high-performance data structures, such as: a slightly modified version of the well-known compressed-row, compressed-column representation of probability (rate) matrices, and bit vectors for representing sets of states.

Since MRMC v1.2.2 the tool supports all major platforms, namely Microsoft Windows, Linux and Mac OS X. The tool is distributed under the GNU General Public License (GPL) [PtFSF07a] and is available for free download at:

http://www.mrmc-tool.org/

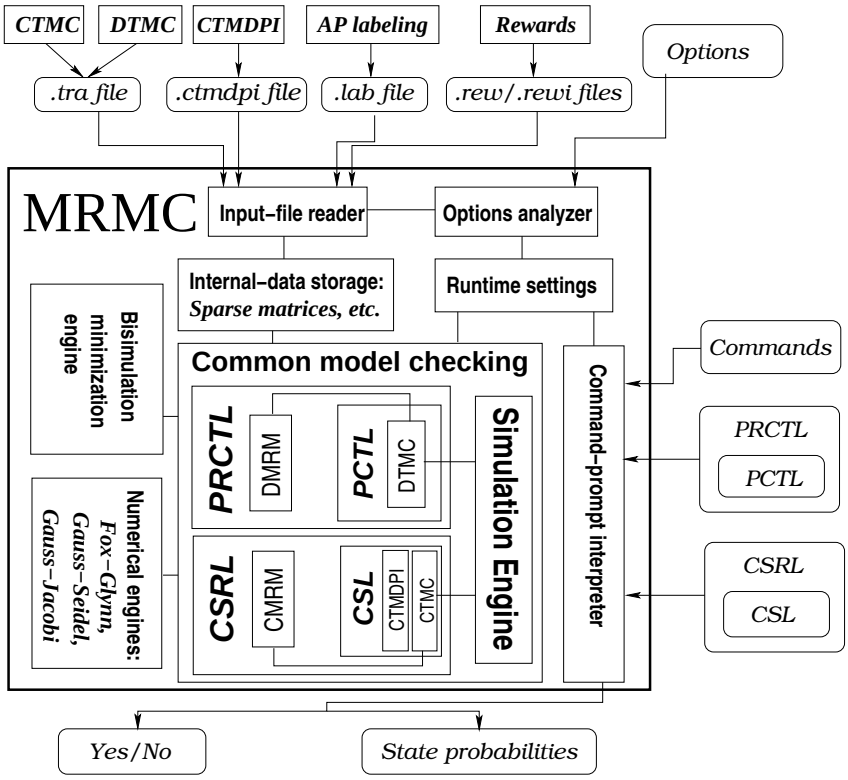


Figure 2.1: Tool architecture of MRMC

A sketch of the MRMC tool architecture is provided in Figure 2.1. Below we refer to it for illustration purposes when giving examples of MRMC inputs, outputs and functionality.

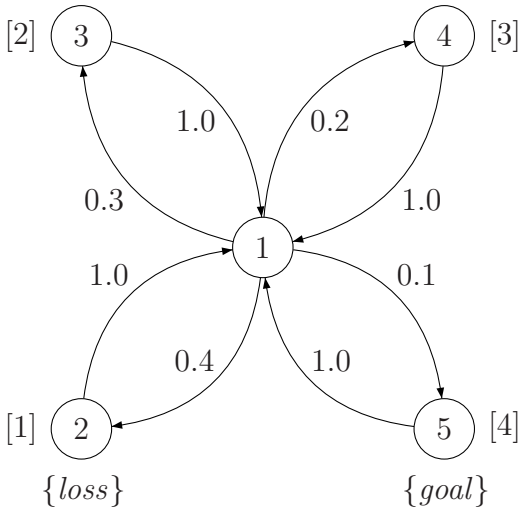


Figure 2.2: The die game: DMRM model

Example 1 *Consider a die with only four wedges that have numbers* 1, 2, 3 *and* 4 *imprinted on them. Let the die be biased in such a way that we get the before-mentioned outcomes with probabilities* 0.4 0.3, 0.2 *and* 0.1, *respectively. One can now play a simple game where the game round consists of continuously tossing the die until winning, if the outcome is* 4 *and the accumulated outcome is from* 5 *to* 50, *or losing, if the outcome is* 1.

A natural question rises: Is the probability to win this game, e.g. within 100 *tosses, larger than* 0.5? *The answer to such a question can be given if we represent this game as a DMRM model and reformulate the question in terms of the PRCTL logic.*

The required DMRM is provided in Figure 2.2. Here we have five states where state 1 *represents the moment at which the die is tossed and states from* 2 *to* 5 *correspond to the die outcomes from* 1 *to* 4. *These outcomes are transformed into state rewards and placed next to the states in the square braces. The* loss *and* goal *states are marked by labels enclosed in the curly braces. The* goal *label corresponds to the outcome* 4 *and in order to win, by reaching this state, the accumulated outcome has to be within* 5 *and* 50.

The measure-of-interest can be formulated as: $\mathrm{P}_{>0.5}\left(\neg loss\ \mathrm{U}^{[0,199]}_{[5,50]}\ goal\right)$. *The given property asserts that the probability to reach the* $goal$ *state, without visiting the* $loss$ *state within* 199 *time steps, and the accumulated reward being from* 5 *to* 50, *is larger than* 0.5. *Notice that we have the upper time bound* 199 *that in the model corresponds to* 100 *die tosses.*

On the start up, MRMC accepts several command-line options, e.g., that specify the model (CTMC, DTMC, etc.), and expects five input files: a `.tra` – file describing the probability or rate matrix of a DTMC, CTMC or an MRM, a `.lab` – file indicating the state labelling with atomic propositions, a `.ctmdpi` – file describing the rate matrix and the transition labelling of a CTMDPI, a `.rew` – file specifying the state-reward structure of an MRM, and a `.rewi` – file specifying the impulse-reward structure of an MRM. For all supported model types either the `.tra` or `.ctmdpi` and `.lab` files are compulsory, whereas `.rew` and `.rewi` files are used only for specifying reward models.

Example 2 *The DMRM model of Example 1 can be seen as a superposition of three parts:* (i) *the DTMC given by state-transitions and corresponding distributions,* (ii) *the labelling*

function that maps sets of labels to the DTMC states, and (iii) *the state-reward function that maps reward values to the DTMC states. In order to be used with MRMC, all these three parts have to be transformed into the MRMC input files. Such a translation is given in Table 2.1.*

The `game.tra` *file contains an intuitive text-based representation of the DTMC, i.e. its state transitions and corresponding probabilities. The* `game.lab` *file contains label declarations and maps sets of labels to the states of DTMC. Similarly the* `game.rew` *file contains mapping of the state rewards to the model states.*

In order to start MRMC with the given input files the following command should be executed in a shell environment such as csh, bash *on Linux (Mac OS X), or* Dos command prompt *on Microsoft Windows:*

```
MRMC/bin> mrmc dmrm game.tra game.lab game.rewi
```

When executed, this command starts MRMC by triggering several of its components, see Figure 2.1. First "Options analyzer" parses the command-line arguments, setting up the DMRM *model as the current one in the "Runtime settings" component and invoking "Input-file reader" for processing the files* game.tra, game.lab *and* game.rewi. *At this stage necessary data structures for storing the probability matrix are provided by "Internal-data storage", labelling and state rewards, which then become accessible through "Runtime settings". Once MRMC is started it produces the following output:*

```
 ------------------------------------------------------------------
|                    Markov Reward Model Checker                   |
|                        MRMC version 1.4.1                        |
|               Copyright (C) RWTH-Aachen, 2006-2009.              |
|         Copyright (C) The University of Twente, 2004-2008.       |
|                            Authors:                              |
|    Ivan S. Zapreev (since 2004), Christina Jansen (2007-2008),   |
|     David N. Jansen (since 2007), E. Moritz Hahn (2007-2008),    |
|           Sven Johr (2006-2007), Tim Kemna (2005-2006),          |
|                    Maneesh Khattri (2004-2005)                   |
|           MRMC is distributed under the GPL conditions           |
|           (GPL stands for GNU General Public License)            |
|          The product comes with ABSOLUTELY NO WARRANTY.          |
|  This is a free software, and you are welcome to redistribute it. |
 ------------------------------------------------------------------
Logic                   = PRCTL
Loading the 'simple_dmrm_dice.tra' file, please wait.
States=5, Transitions=8
Loading the 'simple_dmrm_dice.lab' file, please wait.
Loading the 'simple_dmrm_dice.rew' file, please wait.
The Occupied Space is 992 Bytes.
Type 'help' to get help.
>>
```

where, first the MRMC logo is printed, then some general information about the accepted model and finally the MRMC shell invitation sign >>. *After that the tool is up and running, ready to accept user commands.*

Once started, MRMC provides a shell-like environment (*a command prompt*) where the user can specify the tool run-time options, such as a use of certain algorithms, and the properties that have to be verified. For every verification problem the tool outputs a set of states

that satisfy the given property and, if applicable, the list of probabilities. Note that the complete list of MRMC command-line options and command-prompt commands can be found in Chapter 6.

game.tra	game.lab	game.rew
STATES 5	#DECLARATION	2 1
TRANSITIONS 8	loss goal	3 2
1 2 0.4	#END	4 3
1 3 0.3	2 loss	5 4
1 4 0.2	5 goal	
1 5 0.1		
2 1 1.0		
3 1 1.0		
4 1 1.0		
5 1 1.0		

Table 2.1: The die game: MRMC input files

Example 3 *Extending Example 2, we can answer to the model checking problem of Example 1, by executing the following command in the MRMC command prompt:*

```
>>P{>0.5}[ !loss U[0,199][5,50] goal]
$RESULT: ( 0.0647999, 0.0000000, 0.0959998, 0.1199998, 0.1199997 )
$STATE: {  }
The Total Elapsed Model-Checking Time is 45 milli sec(s).
>>
```

By doing so we invoke the "Command-prompt interpreter" component, cf. Figure 2.1, that processes all commands of the MRMC shell. This component, using "Runtime settings" determines which model-checking engine is needed, in this case it is "PRCTL model checking", and then invokes it. As a result, we get two outputs: a probability vector $RESULT, *and a set of states* $STATE. *The former corresponds to the list of probabilities to satisfy the formula* $\neg loss\ \mathrm{U}^{[0,199]}_{[5,50]}\ goal$ *when starting in the first, second, etc. states. The latter one is the set of states in which the formula* $\mathrm{P}_{>0.5}\left(\neg loss\ \mathrm{U}^{[0,199]}_{[5,50]}\ goal\right)$ *is satisfied.*

Since, when playing the die game, we always start in state 1, *i.e. we first toss the die, from the vector* $RESULT *we can see that the probability to win the game within* 100 *die tosses is just* `0.0647999` *and thus indeed* 1 *is not in the set* $STATE.

Since we already have a good idea of how MRMC works, we proceed with concrete information on the tool installation process. The die example from above will be referenced in the upcoming chapters to illustrate the tool functionality.

3 Building MRMC

This chapter is devoted to the installing process of MRMC and all related components. MRMC can be freely downloaded from:

http://www.mrmc-tool.org/

Further, we first explain how to build MRMC on the supported platforms. After that we proceed with a section on getting and configuring the optional MRMC test suite, which is useful for internal, functional and performance testing of the tool.

3.1 Building MRMC from source code

To compile MRMC from sources GNU Make as well as GCC is needed. Additionally, compilation under Windows requires Cygwin.

- http://gcc.gnu.org/
- http://www.cygwin.com/

3.1.1 Getting & Installing GSL

Since MRMC v1.3, the tool requires the GNU Scientific Library (GSL), a collection of numerical routines for scientific computing. The current version of GSL is available at:

ftp://ftp.gnu.org/pub/gnu/gsl

GSL follows the standard GNU installation procedure. Brief installing instructions can be found here, for further information on this topic see [PtFSF07b].

Note that, in order to install GSL on Windows you are first required to install Cygwin and then to perform GSL installation procedure using the Cygwin shell. For more details see Section 3.1.3.

First, unpack the GSL distribution file into the location of your choice, enter that directory and prepare the Makefiles by using the *configure* command. Afterwords run *make* to compile and *make install* to install the library. On most systems the latter will require root privileges.

```
$ tar -xf gsl-1.9.tar.gz
$ cd gsl-1.9
$ ./configure
$ make
$ sudo make install
```

Further we assume that GSL is properly installed on your system.

3.1.2 Linux

To build MRMC on Linux unpack the distribution into the location of your choice. We define `MRMC_HOME_DIR` to be the absolute name of the MRMC distribution folder. After MRMC is unpacked, enter this directory and run *make all*.

```
$ unzip mrmc_src_v1.3.zip
$ cd MRMC_HOME_DIR
$ make all
```

After that you will find the MRMC executable in the folder `MRMC_HOME_DIR/bin`. Note that you might have to modify Makefile.def, depending on your system's configuration. Please check the settings there if `make all` fails.

In order to clean up distribution, i.e. to remove all object files and pre-compiled binaries run *make clean*.

3.1.3 Windows

To build MRMC on Windows first download and install Cygwin

http://www.cygwin.com

Make sure that 'gcc', 'make', 'yacc' ('bison') and 'lex' ('flex') modules are included. Ensure that the absolute name of the MRMC distribution folder does not contain spaces.

In the next step install the GNU Scientific Library (GSL) as described in Section 3.1.1 and then proceed with the installation steps specified in Section 3.1.2. Ensure that all commands are executed within the Cygwin shell.

3.1.4 Mac OS X

To build MRMC on Mac OS use the instructions of Section 3.1.2.

3.1.5 Getting & Using Splint

Some source files are annotated for the static checker splint (see http://www.splint.org). Splint checks a. o. for null pointer assignments, memory leaks, and safety of `#define` macros. Splint can be downloaded from its homepage and installed according to the instructions found there.

Splint can be used as follows:

Check a single source file: To check, e. g., bitset.c (currently the only annotated file):

```
$ cd MRMC_HOME_DIR/obj
$ make lint-bitset
```

One has to run `make lint-`*filename* in the directory `MRMC_HOME_DIR/obj` independently from the directory where the source file is located.

Check all source files: Currently, this will incur a lot of error messages, as not yet all sources have been annotated.

```
$ cd MRMC_HOME_DIR
$ make lint
```

Check the test suite sources: Internal tests (see below) also have source files. As the test suite is not packaged together with the source installation, there is an independent way to check the test suite sources:

```
$ cd MRMC_HOME_DIR/test
$ ./test_all.sh -lint -internal
```

will run splint on all annotated internal tests, compile the sources and run the internal tests.

3.2 Getting & Installing Test Suite

The test-suite allows to perform internal, functional and performance testing of MRMC. It is not distributed with the MRMC sources, but it can be freely downloaded from:

http://www.mrmc-tool.org/

After downloading the `MRMC_test_v1.3.zip` file, unpack it in the MRMC folder. As a result a directory `MRMC_HOME_DIR/MRMC_test_v1.3/` will be created. Further, for brevity, we assume that you rename it into `MRMC_HOME_DIR/test/`.

3.2.1 Configuring tests

The main configuration parameters of the MRMC test-suite can be set in the

`MRMC_HOME_DIR/test/settings.cfg`

configuration script. These parameters are subdivided into two groups:

General settings

- `MRMC_HOME_DIR` - The absolute name of the MRMC distribution directory.
- `MRMC` - The location of the MRMC binary. This setting does not need to be changed if `MRMC_HOME_DIR` is set correctly. Note that, when running MRMC on Windows, the binary name should be set to `mrmc.exe`.
- `VALGRING_HOME` - The absolute path to the `valgrind` executable [ABFH+08]. It is only required if tests are run under the `-valgrind` option. Note that in this case MRMC should be first recompiled with the `-O0 -ggdb -g` options, which are available in `MRMC_HOME_DIR/makefile.def`.
- `VALGRIND_LOG_FILES_DIR` - The absolute name of the folder for storing *log* filed produced by `valgrind`.
- `EXTRA_VALGRIND_PARAM` - Extra options for `valgrind`.

Performance-test settings The performance part of the test suite was developed for Linux platform only. It is not proven to work under Windows or Mac OS X.

- `PRISM` - The absolute path of the PRISM [KNP02] command line executable. This setting is required for generating performance-test models.
- `TMPDIR` - This setting should point to a local directory, which will be used for storing generated models.
- `YMER` - The absolute path of the Ymer [You05b] command line executable[2].
- `VASTA_JAR` - The absolute path of the VESTA [SVA04] jar file[2].
- `NUMBER_OF_PERFORMANCE_REPETITIONS` - The number of times every performance test is going to be repeated. If set to zero, no "elapsed-time" statistics is collected. At the same time the functional testing and the memory-usage statistics are collected only for the `lumping` sub suite.
- `MILLISECONDS` - The time units of the "elapsed-time" plots.
- `KILOBYTES` - The data units of the "memory-usage" plots.
- `CONFUNIT`- The data units of the "confidence" plots[2].
- `PERFORMANCE_TEST_TIMEOUT_SECS` - The timeout (in seconds) for each performance test invocation.

For more information on the MRMC test suite, we refer to Chapter 9 and also to the test-suite manual: `MRMC_HOME_DIR/test/TS_Manual.pdf`.

[2]This setting is required only for the `simulation` sub suite.

4 MRMC's Input Files

As already mentioned in Chapter 2 MRMC expects five input files: a `.tra` – file describing the probability or rate matrix of a DTMC, CTMC or an MRM, a `.lab` – file indicating the state labeling with atomic propositions, a `.ctmdpi` – file describing the rate matrix and the transition labeling of a CTMDPI, a `.rew` – file specifying the state-reward structure, and a `.rewi` – file specifying the impulse-reward structure. For all supported model types either the `.tra` or the `.ctmdpi` and `.lab` files are compulsory, whereas `.rew` and `.rewi` files are used only for specifying reward models.

Here we would like to give a formal definition of the structure the input files should meet. Please note, that **MRMC does not check if the input is in a proper format and thus may show malicious behavior in case of a wrong input**. For examples of MRMC's input files see Table 2.1 of Chapter 2. Additionally, examples for CTMDPIs can be found in Appendix A.

4.1 The `.tra` File Format

The `.tra` file contains the rate (probability) matrix:

```
File structure:

Tra_File =   Header Body
Header   =   'STATES' <number of states> \n
             'TRANSITIONS' <number of transitions> \n
Body     =   <from state> <to state> <rate/probability> \n
              Body
           | <from state> <to state> <rate/probability> \n
```

The header defines the number of states and transitions in the system. The body contains transitions in the format:

```
<from state> <to state> <rate/probability>
```

Note that, "from state" and "to state" should be given as natural numbers, the rates/probabilities as real numbers. State indexes start with 1 and transitions must be given in ascending order of first row and then column index.

4.2 The `.lab` File Format

The `.lab` file contains the labeling of states with atomic propositions.

```
File structure:

Lab_File         =   Declaration Body
Declaration      =   '#DECLARATION' \n
                     Atomic_Prop_List \n
                     '#END' \n
Body             =   <state> Atomic_Prop_List \n Body
                   | <state> Atomic_Prop_List \n
Atomic_Prop_List =   <atomic proposition> Atomic_Prop_List
                   | <atomic proposition>
```

In the declaration section all needed atomic propositions must be defined. We allow quite complicated atomic propositions, namely the ones that fit the following regular expression:

```
<atomic proposition> =  {let}{alnum}*
let                  =  [_a-zA-Z]
alnum                =  [_a-zA-Z0-9<>_^*+-=]
```

The propositions are assigned to states in the following manner:

```
<state> Atomic_Prop_List
```

4.3 The `.ctmdpi` File Format

The `.ctmdpi` file contains the rate matrix and additionally the transition labeling to distinguish between different non-deterministic choices. The file format for the transition descriptions are given below.

```
File structure:

Ctmdpi_File       =   Header Body_Int_Nondet
Header            =   'STATES' <number of states> \n
                      '#DECLARATION' \n
                       Atomic_Prop_List \n
                      '#END' \n
Body_Int_Nondet   =   <from state> <label> \n
                      * <to state> <rate> \n
                      { * <to state> <rate> } \n
                      Body_Int_Nondet
                    | <from state> <label> \n
                      * <to state> <rate> \n
                      { * <to state> <rate> } \n
```

The header defines the number of states the MDP contains as well as all needed transition labels, which are used to label the non-deterministic decisions.

The body contains the transitions and transition labels, where “from state” is the state the the selection starts from and “label” is the external choice that was made. After this line, a number of lines follow, which list the states “to state” one can go to with rate “rate”.

Note that, “from state” and “to state” should be given as natural numbers, the rates/probabilities as real numbers. State indexes start with 1 and transitions must be given in ascending order of first row and then column index.

4.4 The `.rew` File Format

The `.rew` file contains the state-reward definitions.

```
File structure:

Rew_File =   Body
Body     =   <state> <reward> \n  Body
           | <state> <reward> \n
```

Note that, only natural reward values are allowed, therefore any rational rewards must (and can) be transferred into natural numbers first.

4.5 The `.rewi` File Format

The `.rewi` file contains the impulse-reward definitions.

```
File structure:

Rewi_File =   Header Body
Header    =   'TRANSITIONS' <number of transitions> \n
Body      =   <from state> <to state> <reward> \n  Body
            | <from state> <to state> <reward> \n
```

In the header the number of transitions is given, the body contains reward to transition assignments in the format:

```
<from state> <to state> <reward>
```

Note that, “from state” and “to state” should be given as natural numbers. Furthermore, like for the `.rew` file only natural reward values are allowed.

4.6 Getting MRMC models

Specifying a whole model in the formats explained above is not very intuitive especially for large systems. Therefore in this section we introduce two tools – namely PRISM and PEPA – that offer a clearly defined language for designing models. Both of them feature the automatic generation of MRMC input files.

4.6.1 PRISM

PRISM [KNP08b] stands for Probabilistic Symbolic Model Checker and is being developed at the University of Birmingham, United Kingdom, for the analysis of probabilistic systems.

MRMC models can be generated from PRISM models starting from the tool version 3.0. PRISM can be downloaded from:

http://www.prismmodelchecker.org/download.php

The model-generation options of PRISM are listed here and can also be obtained by running `prism -help`:

- –exportmrmc - Use MRMC format when exporting matrices/vectors/labels.
- –exportlabels <file> - Export the list of labels and satisfying states to a `.lab`-file.
- –exporttrans <file> - Export the transition matrix to a `.tra`-file.
- –exportstaterewards <file> - Export the state rewards vector to a `.rew`-file.
- –exporttransrewards <file> - Export the transition rewards matrix to a `.rewi`-file.

Example 4 *Consider Example 2 of Chapter 2. The DMRM model given in Figure 2.2 can be specified as the following PRISM model:*

```
--------------------------- File: game.pm -----------------------------

probabilistic

module Dice

dice_state : [1..5] init 1;

[] dice_state=1 -> 0.4:(dice_state'=2) + 0.3:(dice_state'=3)
+ 0.2:(dice_state'=4) + 0.1:(dice_state'=5);
[] dice_state=2 -> 1.0:(dice_state'=1);
[] dice_state=3 -> 1.0:(dice_state'=1);
[] dice_state=4 -> 1.0:(dice_state'=1);
[] dice_state=5 -> 1.0:(dice_state'=1);

endmodule

rewards
dice_state=2 : 1;
dice_state=3 : 2;
dice_state=4 : 3;
dice_state=5 : 4;
endrewards

-----------------------------------------------------------------------
```

```
-------------------------- File: game.pctl ----------------------------

label "loss" = dice_state=2;
label "goal" = dice_state=5;

-----------------------------------------------------------------------
```

In the file `game.pm` *the DMRM model is specified, whereas the file* `game.pctl` *contains only the state labellings.*

To generate the MRMC model with PRISM, run the following command,

```
$ prism game.pm game.pctl -exportmrmc -exportlabels
game.lab -exporttrans game.tra -exportstaterewards game.rew
```

which produces the `.tra`, `.lab` *and* `.rew` *input files shown in Table 2.1 of Chapter 2. These files can be immediately consumed by MRMC.*

For more information on generating MRMC models using PRISM see [KNP08b].

4.6.2 Performance Evaluation Process Algebra (PEPA)

Performance Evaluation Process Algebra (PEPA) [Hil96] is an algebraic process-oriented language for modeling concurrent systems. The process algebra is being mainly developed in Laboratory for Foundations of Computer Science, University of Edinburgh, United Kingdom. Performance of a PEPA model can be evaluated by deriving and analyzing the underlying CTMC. PEPA modelers are provided with the PEPA Workbench [TG06],

http://www.dcs.ed.ac.uk/pepa/tools/

an Eclipse-platform [Fou07] application for managing the models. One of the PEPA Workbench features is an Eclipse wizard for exporting PEPA models into the MRMC input-file formats.

5 Running MRMC

In order to start MRMC open a shell environment such as csh or bash on Linux and Mac OS X, or Dos command prompt on Microsoft Windows and switch to `MRMC_HOME_DIR`.

5.1 Command line options

Starting MRMC without parameters

- for Linux/Max OS: `$ ./bin/mrmc`
- for Windows: `$ ./bin/mrmc.exe`

will yield the following output:

```
ERROR: The <model> parameter is undefined.
Usage: mrmc <model> <options> <.tra file> <.ctmdpi file> <.lab file> <.rew file>
<.rewi file>
        <model>         - could be one of {ctmc, dtmc, dmrm, cmrm, ctmdpi}.
        <options>       - could be one of {-ilump, -flump}, optional.
        <.tra file>     - is the file with the matrix of transitions
                          (for DMRM/CMRM, DTMC/CTMC).
        <.ctmdpi file>  - is the file with the transition matrix and transition labels
                          (for CTMDPI).
        <.lab file>     - contains labeling.
        <.rew file>     - contains state rewards (for DMRM/CMRM).
        <.rewi file>    - contains impulse rewards (for CMRM, optional).

Note: In the '.tra' and '.ctmdpi' file transitions should be ordered by rows and columns!
```

The `model`-parameter should be set to one of the supported models, namely CTMC, DTMC, CMRM, DMRM and CTMDPI. Remember that the latter model is a CTMDP with internal non-determinism, see Appendix A.

Options `-ilump` and `-flump` enable formula- independent and dependent lumping correspondingly. For more information on lumping, please consider reading [KKZJ07].

We expect users to provide MRMC with the input files that meet the formats specified in Chapter 4, for illustration see Example 2 on page 6. Note that, the order of input files, options and other parameters does not have to be strict.

A complete list of all MRMC runtime commands, sorted by their affiliation to different model checking aspects, can be found in the next chapter.

6 MRMC run-time Commands

Once started, MRMC provides a shell-like environment (*a command prompt*) where the user can use the tool run-time commands to set for example the use of certain algorithms, or specify the properties that have to be verified. Further we will list and discuss MRMC's command-prompt commands sorted by their affiliation to the different aspects of model checking.

6.1 Basic Commands

6.1.1 `help`

When typing `help` in MRMC's command prompt, information on general commands is displayed:

```
quit    - exit the program.
help HT - display a help info on a given topic.
print   - print run-time settings.
print tree - print the formula tree with the results and supplementary
             information.
$RESULT[N] - access the computed results of U, X, L, S, E, C, Y operators
             by a state index.
$STATE[N]  - access the state-formula satisfiability set by a state
             index.
set *      - Where * is one of the following:
print L          - Turn on/off most of the resulting output, see
                   '$RESULT[I]' and '$STATE[I]' commands.
simulation L     - Turn on/off the simulation engine.

Here:
        HT is one of {logic, simulation, rewards, common}.
        L is one of {on, off}.
        N is a natural number.
```

First we are going to explain the basic commands listed in this help output, the more involved ones are covered in the subsequent sections.

`quit` – Exits the program.

`help HT` – For some terms a specialized help is available. See the description provided for `help logic`, `help simulation`, `help rewards` and `help common` below.

`print tree` – Prints the tree of the last model-checked formula with all intermediate results.

Note: The next two commands provide different output in case of using the discrete event simulation engine. For more details we refer to Section 8.

`$RESULT[N]` – Allows to access the probability of satisfying the model-checked formula in state `N`.

`$STATE[N]` – Displays whether state `N` satisfies the model-checked formula, i. e. for a state fulfilling the formula the result is `TRUE`, otherwise `FALSE`.

6.1.2 `help logic`

The command `help logic` prints the formal syntax, given in *Extended Backus-Naur Form (EBNF)*, of the logic formulae accepted by MRMC. The output depends on the value of the `logic` parameter with which MRMC was invoked. Figures 6.1 through 6.4 show outputs for all available logics. These logics allow to specify model-checking properties, as it is done in Example 3 on page 8. Additional information on the logic semantics and examples are provided in Chapter 7.

6.1.3 `help simulation`

The `help simulation` command provides the user with all options related to MRMC's simulation engine:

```
set *    - Where * is one of the following:
         sim_type ST          - Sets the simulation type, \ie{} either
                                do simulation for all initial states
                                or just one.
         initial_state N      - Sets the initial state for the simulation
                                type ST == one.
         sim_method_steady MS - Sets the simulation mode for the
                                steady-state (long-run) operator.
         reg_method_steady RM - Sets the mode for the regeneration method when
                                model checking the steady-state operator.
         gen_conf R           - The confidence level for simulation.
         indiff_width R       - The indifference-region width.
         max_sample_size N    - The maximum sample size.
         min_sample_size N    - The minimum sample size.
         sample_size_step N   - The sample-size increase step.
         sample_size_step_type SS - Sets the sample-size step type.
         sim_method_disc RNG  - The random-number generator for a
                                discrete distribution.
         sim_method_exp RNG   - The random-number generator for an
                                exponential distribution
                                (time-interval until, CSL).
For the simulation of unbounded until and the pure simulation of
steady-state (long-run) operator:
         max_sim_depth N  - The maximum simulation depth.
```

```
CONST =   ff | tt
SFL   =   CONST
        | LABEL
        | ! SFL
        | SFL && SFL
        | SFL || SFL
        | ( SFL )
        | P{ OP R }[ PFL ]
        | L{ OP R }[ SFL ]
PFL   =   X SFL
        | SFL U SFL
        | SFL U[ N, N ] SFL
```

Figure 6.1: PCTL formulae

```
CONST =   ff | tt
SFL   =   CONST
        | LABEL
        | ! SFL
        | SFL && SFL
        | SFL || SFL
        | ( SFL )
        | P{ OP R }[ PFL ]
        | E [ R, R] [ SFL ]
        | E [N][ R, R] [ SFL ]
        | C [N][ R, R] [ SFL ]
        | Y [N][ R, R] [ SFL ]
PFL   =   X SFL
        | SFL U SFL
        | SFL U[ N, N][ R, R]
          SFL
```

Figure 6.2: PRCTL formulae

```
CONST =   ff | tt
SFL   =   CONST
        | LABEL
        | ! SFL
        | SFL && SFL
        | SFL || SFL
        | ( SFL )
        | P{ OP R }[ PFL ]
        | S{ OP R }[ SFL ]
PFL   =   X SFL
        | SFL U SFL
        | X[ R, R ] SFL
        | SFL U[ R, R ] SFL
```

Figure 6.3: CSL formulae

```
CONST =   ff | tt
SFL   =   CONST
        | LABEL
        | ! SFL
        | SFL && SFL
        | SFL || SFL
        | ( SFL )
        | P{ OP R }[ PFL ]
        | S{ OP R }[ SFL ]
PFL   =   X SFL
        | SFL U SFL
        | X[ R, R ] SFL
        | SFL U[ R, R ] SFL
        | X [R, R][R, R] SFL
        | SFL U[ R, R][ R, R]
          SFL
```

Figure 6.4: CSRL formulae

```
        min_sim_depth N  - The minimum simulation depth.
        sim_depth_step N - The simulation-depth increase step.
        bscc_dim_multiplier N - The BSCC dimension multiplier for the
                                sample-based regeneration state choice.
Here:
        RNG is one of {app_crypt, ciardo, prism, ymer, gsl_ranlux,
                       gsl_lfg, gsl_taus}.
        ST is one of {one, all}.
        SS is one of {auto, manual}.
        MS is one of {pure, hybrid}.
        RM is one of {pure_reg, heuristic}.
        R is a real value.
        N is a natural number.
```

For more information on the simulation options read Section 6.2.3. For details on the available *Random Number Generators (RNGs)* read Chapter 8.

6.1.4 `help rewards`

The command `help rewards` yields the following output:

```
set *   - Where * is one of the following:
        method_until_rewards MU - Method for time-reward-bounded until
                                  formula.
        w R                     - The probability threshold for
                                  uniformization
                                  Qureshi-Sanders.
        d R                     - The discretization factor for
                                  discretization Tijms-Veldman.
Here:
        MU is one of { uniformization_sericola,
                       uniformization_qureshi_sanders,
                       discretization_tijms_veldman }.
        R is a real value.
```

For more information on reward options listed above, we refer to Section 6.2.4.

6.1.5 `help common`

The `help common` command provides the user with information concerning options, related to all model-checking procedures and numerical methods. For detailed information on these options, see Sections 6.2.1 and 6.2.2.

```
set *   - Where * is one of the following:
        ssd L           - Turn on/off the steady-state detection for time
                          bounded until (CTMC model).
        error_bound R   - Error Bound for all iterative methods.
        max_iter N      - Number of Max Iterations for all iterative
                          methods.
        overflow R      - Overflow for the Fox-Glynn algorithm.
        underflow R     - Underflow for the Fox-Glynn algorithm.
        method_path M   - Method for path formulas.
        method_steady M - Method for steady state formulas.
        method_bscc MB  - Method for BSCC search.
```

```
Here:
        L is one of {on, off}.
        R is a real value.
        M is one of {gauss_jacobi, gauss_seidel}.
        MB is one of {recursive, non_recursive}.
```

6.1.6 `print`

The `print` command displays the current status of all relevant run-time settings. A sample output may look as follows:

```
---General settings:
Logic                   = PCTL
Formula ind. lumping    = OFF
Formula dep. lumping    = OFF
M. C. simulation        = OFF
Method Path             = Gauss-Seidel
Method Steady           = Gauss-Seidel
Method BSCC             = Recursive
Results printing        = ON

---Numerical methods:
-Iterative solvers:
  Error Bound           = 1.000000e-06
  Max Iterations        = 1000000
```

A complete list of all runtime options and their correspondence to the `print` command output can be found in Section 6.2. Below we describe the parameters listed in the output above:

- `General settings`
  - `Logic` – Corresponds to the `logic` parameter MRMC was invoked with (cf. Chapter 5).
  - `Formula ind. lumping` – Is related to the option `-ilump` MRMC was invoked with (cf. Chapter 5).
  - `Formula dep. lumping` – Is related to the option `-flump` MRMC was invoked with (cf. Chapter 5).
  - `M. C. simulation` – Corresponds to the command `set simulation L` (cf. Section 6.2.3). With simulation enabled, the output of the `print` command is extended.
  - `Method Path` – Corresponds to the command `set method_path M` (cf. Section 6.2.1).
  - `Method Steady` – Corresponds to the command `set method_steady M` (cf. Section 6.2.1).
  - `Method BSCC` – Corresponds to the command `set method_bscc MB` (cf. Section 6.2.1).

  - `Results printing` – Reports whether model checking results are printed. In order to manage this option, use `set print L` (cf. Section 6.2.1).

- `Numerical methods`
  - `Error Bound` – Corresponds to the command `set error_bound R` (cf. Section 6.2.2).
  - `Max Iterations` – Corresponds to the command `set max_iter N` (cf. Section 6.2.2).

Note that, depending on specific run-time settings, the output of the `print` command may be extended with additional information. For example, when the simulation engine is turned on, the user is provided with information about its parameters as well.

6.2 Advanced Commands

In this section, we list the remaining MRMC commands that allow to influence its run-time behavior. Every command will be given in the following format:

<command name> (related `print` output) – short description.

6.2.1 Common

Let us consider the MRMC commands responsible for managing the general behavior of the tool. When displaying the current settings with the `print` command, all commands described here can be found in the section `General settings`. Below we have `L` $\in$ {`on, off`} and `M` $\in$ {`gauss_jacobi, gauss_seidel`}.

`set print L` **(Results printing)** – Turns on/off printing of model-checking results that follows the formula verification procedure.

`set ssd L` **(Steady-state detection)** – Turns on/off the steady-state detection for the time-bounded until operator (CTMC/CMRM).

`set method_path M` **(Method Path)** – Sets the iterative method for solving a system of linear equations when computing reachability probabilities for model checking of an unbounded-until formula (DTMC/DMRM and CTMC/CMRM).

`set method_steady M` **(Method Steady)** – Sets the iterative method for solving a system of linear equations when computing steady-state probabilities of BSCCs[1]. The latter happens when model checking the steady-state, long-run and unbounded-until formulas (DTMC/DMRM and CTMC/CMRM).

[1] Bottom Strongly Connected Components

`set method_bscc MB` **(Method BSCC)** – Sets the method used when searching for bottom strongly connected components. Here `MB` defines the BSCCs search implementation based on:

- `recursive` – Recursive functions.
- `non_recursive` – Cycle iterations.

Generally, the `recursive` method is faster, but can run into a segmentation fault caused by an insufficient stack size (it is likely to happen for large models). The `non_recursive` method does not use recursive function calls, and thus avoids the stack exhaustion.

`set method_ctmdpi_transient CB` **(CTMDPI Transient Method)** – Sets the method which is used to compute transient reachability for CTMDPIs. By default, this value is set to `hd_auto`. This means that for uniform CTMDPIs the method of [BHKH05] is used, but for non-uniform CTMDPIs the method of [BFK+09] is taken. With `hd_non_uni` the method of [BFK+09] can be enforced. Setting the value to `hd_uni` leads to usage of [BHKH05] for uniform CTMDPIs, but to failure for non-uniform models. This can be used as an additional check, in case generated CTMDPIs are expected to be uniform.

6.2.2 Numerical Methods

In this section we list commands that allow to manage the numerical engine of MRMC. The list of corresponding parameters can be found in the `Numerical Methods` section of the `print` command output.

`set error_bound R` **(Iterative solvers/Error Bound)** – Sets the error bound for all iterative methods.

`set max_iter N` **(Iterative solvers/Max Iterations)** – Sets the maximum number of iterations for all iterative methods.

`set overflow R` **(Fox-Glynn algorithm/Overflow)** – Sets the overflow threshold for the Fox-Glynn algorithm [FG88].

`set underflow R` **(Fox-Glynn algorithm/Underflow)** – Sets the underflow threshold for the Fox-Glynn algorithm.

6.2.3 Simulation

In this section we list commands related to MRMC's discrete-event simulation engine. At present simulation can be used when model checking unbounded-until, time-bounded until, and steady-state operators on CTMC/CMRM models. We do not support nested simulation. Therefore, given a formula we only apply simulation to the (appropriate) sub formulas that have the closest location to the formula-tree root. The sub-formulas that are located below are verified using numerical methods.

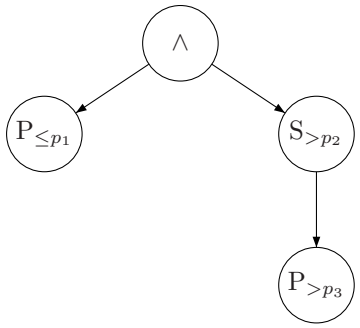


Figure 6.5: Formula tree: $\mathrm{P}_{\leq p_1}(\Psi \ \mathrm{U} \ \Phi) \ \wedge \ \mathrm{S}_{>p_2}(\mathrm{P}_{>p_3}(\Psi' \ \mathrm{U} \ \Phi'))$

Example 5 *Consider the formula* $\mathrm{P}_{\leq p_1}(\Psi \ \mathrm{U} \ \Phi) \ \wedge \ \mathrm{S}_{>p_2}(\mathrm{P}_{>p_3}(\Psi' \ \mathrm{U} \ \Phi'))$ *with the corresponding formula tree depicted in Figure 6.5. The formula is a conjunction of the unbounded-until formula* $\mathrm{P}_{\leq p_1}(\Psi \ \mathrm{U} \ \Phi)$ *and the steady-state formula* $\mathrm{S}_{>p_2}(\mathrm{P}_{>p_3}(\Psi' \ \mathrm{U} \ \Phi'))$*. The latter one has an unbounded-until sub formula. In the given situation MRMC applies numerical methods to verify sub formula* $\mathrm{P}_{>p_3}(\Psi' \ \mathrm{U} \ \Phi')$*. Then the unbounded-until sub formula* $\mathrm{P}_{\leq p_1}(\Psi \ \mathrm{U} \ \Phi)$ *and steady-state sub formula* $\mathrm{S}_{>p_2}(\mathrm{P}_{>p_3}(\Psi' \ \mathrm{U} \ \Phi'))$ *are model checked using simulations.*

With simulation on, the `print` command output is extended with parameters of the simulation engine, cf. Example 6 of Chapter 8. These options are displayed in the `Monte Carlo Simulation` section. Below, we assume that `L` $\in \{$`on, off`$\}$ and `N` $\in \mathbb{N}$.

`set simulation L` **(M. C. Simulation)** – Turns MRMC's simulation engine on/off. The status of simulation engine is reported under the `General settings` sub point of the `print` command output.

`set sim_type ST` **(Simulation type)** – Sets the simulation type `ST` $\in \{$`one, all`$\}$. Unlike in numerical model checking, where verification is done for all initial sates at once, in model checking via simulation we can either do verification for one initial state or all initial states. The former can be set by using the `set initial_state N` command, described below.

`set initial_state N` **(Sim. initial state)** – Sets the state for which the validity of the formula is going to be verified.

`set sim_method_steady MS` **(Sim. steady state)** – Sets the simulation mode for the steady-state/long-run operator. Here, `MS` is one of

- `pure` – Model checking only by discrete simulation.
- `hybrid` – Probabilities of reaching BSCCs are computed numerically.

`set reg_method_steady RM` **(Reg. method steady)** – Sets the regeneration method for the steady-state operator. Here, `RM` is one of

- `pure_reg` – Random choice of the regeneration state.
- `heuristic` – Use heuristic to choose a frequently visited regeneration state.

`set gen_conf R` **(Confidence level)** – Sets the confidence level (probability) with which we can trust the model-checking results. Here, `R` $\in [0.25, 1.0]$. Note that, this confidence level is guaranteed only under a specific condition that is explained in Chapter 8.

`set indiff_width R` **(Indiff. reg. width)** – Sets the width of the indifference region, i. e. the maximum width of the confidence intervals that will be considered. For more details see Chapter 8.

`set max_sample_size N` **(Max sample size)** – Sets the maximum sample size, i. e. the maximum number of independent traces to be considered.

`set min_sample_size N` **(Min sample size)** – Sets the minimum sample size, i. e. the minimum number of independent traces to be considered.

`set sample_size_step_type SS` **(Sample-size step type)** – Sets the type to determine the sample-size increment. Here, `SS` is one of

- `auto` – The sample size step is computed and dynamically set based on relevant factors.
- `manual` – The sample size step is static and manually set.

`set sample_size_step N` **(Sample-size step)** – Sets the increment for the sample-size, i. e. the number by which the number of observations in the samples will be increased, for sequential confidence intervals.

`set sim_method_disc RNG` **(RNG discrete dist.)** – Sets the method of generating values for discrete random variables (cf. Chapter 8). This method is used for simulating state transitions of embedded DTMCs. The Random Number Generator `RNG` can be one of the following:

- `app_crypt` – Combined linear congruential generator.
- `ciardo` – Improved linear congruential generator.
- `prism` – Linear congruential generator, similar to the RNG used in PRISM.
- `ymer` – Mersenne Twister, similar to the RNG used in Ymer.
- `gsl_ranlux` – Ranlux generator, GSL Library.
- `gsl_lfg` – Lagged Fibonacci generator, GSL Library.
- `gsl_taus` – Tausworthe generator, GSL Library.

`set sim_method_exp RNG` **(RNG exponential dist.)** – Sets the RNG for generating exponentially distributed random variables. This method is used for simulating exponentially distributed state-exit times. Here `RNG` $\in \{$ `app_crypt, ciardo, prism, ymer, gsl_ranlux, gsl_lfg, gsl_taus` $\}$.

Note: The following commands are used for managing options specific for the unbounded-until operator.

`set max_sim_depth N` **(Max simulation depth)** – Sets the max. simulation depth, i. e. the maximum number of steps in every simulated path.

`set min_sim_depth N` **(Min simulation depth)** – Sets the min. simulation depth, i. e. the minimum number of steps in every simulated path.

`set sim_depth_step N` **(Simulation-depth step)** – Sets the increment for the simulation-depth, i. e. the number of steps by which the simulation depth will be increased.

`set bscc_dim_multiplier N` **(BSCC dim. multiplier)** – Sets the multiplier for the heuristic regeneration method. As the multiplier increases the heuristic regeneration state choice is more likely to produce better results at the expense of runtime.

6.2.4 Rewards

`set method_until_rewards MU` **(Method Until Rewards)** – Defines the method, that will be used for CSRL model checking of time- and reward-bounded until formulae. Here, `MU` is one of:

- `uniformization_qureshi_sanders` - Uniformization Qureshi-Sanders [QS96]
- `discretization_tijms_veldman` - Discretization Tijms-Veldman [TV00]
- `uniformization_sericola` - Not supported

`set w R` **(Probability threshold)** – Sets the path probability bound for Qureshi & Sanders uniformization algorithm, i. e. only paths with path probability greater or equal to the bound are considered significant relative to the solution.

`set d R` **(Discretization factor)** – Sets the discretization factor for time interval and accumulated rewards in the discretization algorithm by Tijms & Veldman.

7 Property Specification with Temporal Logics

Model checking is the process of checking whether a given model satisfies a given logical formula. As MRMC is a probabilistic model checker, it supports the common logics for specification of probabilistic properties, namely PCTL, PRCTL, CSL and CSRL. In this chapter all the formulae accepted by MRMC will be introduced on the basis of EBNF. For a property specification example, see Example 3 on page 8 or Examples 7 and 8 of Section 8.

PCTL and PRCTL as well as CSL and CSRL (cf. Section 6.1.2) share a set of common formulae. Every logic only extends the set of these formulae. Note that in most cases MRMC performs global model checking, i. e. properties are verified in every model state and the states satisfying the given formula are reported. The exception is model-checking by discrete event simulation, there it is possible to check the validity of the formula in just one given state.

7.1 Common-logic subset

The common formulae are the following:

```
Common Semantics:

CONST =   ff | tt
SFL   =   CONST
        | LABEL
        | ! SFL
        | SFL && SFL
        | SFL || SFL
        | ( SFL )
        | P{ OP R }[ PFL ]
PFL   =   X SFL
        | SFL U SFL
```

We distinguish between two types of formulae: state and path formulae. A state formula `SFL` is interpreted over the states of the considered system and therefore results in a set of states satisfied by the formula. A path formula `PFL` is interpreted over system paths and thus for every given initial state results in a set of paths, starting in this state, that satisfy the formula.

7.1.1 State formulae (SFL)

`tt` **(True)** – Is a constant satisfied in every state of a model.

`ff` **(False)** – Is a constant satisfied in none of model states.

`LABEL` **(Atomic proposition)** – Is satisfied in the states assigned with the given atomic proposition (label).

`!SFL` **(Negation)** – Is satisfied in states, which do not fulfill `SFL`.

$\texttt{SFL}_1$ `&&` $\texttt{SFL}_2$ **(Conjunction)** – Is satisfied in states fulfilling both $\texttt{SFL}_1$ and $\texttt{SFL}_2$.

$\texttt{SFL}_1$ `||` $\texttt{SFL}_2$ **(Disjunction)** – Is satisfied in states fulfilling $\texttt{SFL}_1$ or $\texttt{SFL}_2$.

`P{ OP R}[ PFL]` **(Probability measure)** – For every state, it asserts that the probability measure of paths starting in the given state and satisfying `PFL` meets the probability constraint `OP R`. Here `OP` $\in \{>, <, \leq, \geq\}$ and `R` $\in \mathbb{R}_{[0,1]}$.

7.1.2 Path formulae (PFL)

`X SFL` **(Next)** – Asserts that on a path, starting in some state s, the immediate successor state of s satisfies the formula `SFL`.

$\texttt{SFL}_1$ `U` $\texttt{SFL}_2$ **(Unbounded until)** – Asserts that on a path there is a state satisfying $\texttt{SFL}_2$ and all preceding states satisfy $\texttt{SFL}_1$.

7.2 PCTL

PCTL [HJ94] is an extension of CTL, which allows for probabilistic quantification of properties. PCTL extends the set of common formulae by one state and one path formula.

```
SFL   =  ...
       | L{ OP R }[ SFL ]
PFL   =  ...
       | SFL U[ N, N ] SFL
```

`L{ OP R }[ SFL ]` **(Long-run)** – Checks if the long-run probability for being in states that fulfill `SFL` meet the probability constraint `OP R`.

$\texttt{SFL}_1$ `U[ 0, N ]` $\texttt{SFL}_2$ **(Time-bounded until)** – Asserts that on a path there is a state satisfying $\texttt{SFL}_2$, such that this state is reached within `N` time steps (transitions) and all preceding states on the path satisfy $\texttt{SFL}_1$.

7.3 PRCTL

PRCTL [AHK03] is the rewards extension of PCTL and therefore extends PCTL with the following formulae:

```
SFL   =  ...
       | E[ R, R] [ SFL ]
       | E[N][ R, R] [ SFL ]
       | C[N][ R, R] [ SFL ]
       | Y[N][ R, R] [ SFL ]
PFL   =  ...
       | SFL U[ N, N ][ R, R ] SFL
```

`E[ R1, R2 ] [ SFL ]` – Asserts that the long-run expected reward rate per time-unit for `SFL` states lies within the interval `[R1, R2]`.

`E[N][ R1, R2 ] [ SFL ]` – Asserts that the expected reward rate in `SFL`-states up to n transitions reached at the `N`-th epoch lies within the interval `[ R1, R2 ]`.

`C[N][ R1, R2 ] [ SFL ]` – Asserts that the instantaneous reward in `SFL` states at the `N`-th epoch lies within the interval `[ R1, R2 ]`.

`Y[N][ R1, R2 ] [ SFL ]` – Asserts that the expected accumulated reward rate in `SFL` states until the `N`-th transition lies within the interval `[ R1, R2 ]`.

`SFL1 U[ N1, N2 ][ R1, R2 ] SFL2` **(Time- & reward-interval until)** – Asserts that `SFL2` will be satisfied within $j \in$ `[ N1, N2 ]` steps, that all preceding states satisfy `SFL1`, and that the accumulated reward until reaching the `SFL2`-state lies in the interval `[ R1, R2 ]`.

7.4 CSL

CSL [BHHK03] extends PCTL, but it works with the continuous time domain. Here the long-run operator `L{ OP R }` is substituted with the steady-state operator `S{ OP R }` and the time-bounded next operator is added:

```
SFL   =  ...
       | S{ OP R }[ SFL ]
PFL   =  ...
       | X[ R, R ] SFL
```

`S{ OP R }[ SFL ]` **(Steady-state)** – Is similar to the long-run operator of PCTL, cf. Section 7.2.

`X[ R1, R2 ] SFL` **(Time-bounded next)** – Asserts that a transition is made to a `SFL` state at some time point $t \in$ `[ R1, R2 ]`.

For CTMDPIs, the probability measure operator is interpreted in the way that is has to hold for all possible resolutions of non-determinism in the model. Because of this, in the formula `P{OP R} [ U[ N, N ] ]` , if `OP` $\in \{<, \leq\}$ then the maximum over all (time-abstract, history-dependent) schedulers is computed ([BHKH05, BFK+09]), but if `OP` $\in \{>, \geq\}$ the minimum is taken. For uniform CTMDPIs, the algorithm of [BHKH05] will be used, which is more efficient than the one for general CTMDPIs described in [BFK+09].

7.5 CSRL

CSRL [CKKP05] extends CSL with the following formulae:

```
PFL   =  ...
       | X[R, R][R, R] SFL
       | SFL U[ R, R ][ R, R ] SFL
```

`X[ R1, R1' ][ R2, R2' ] SFL` **(Time- & reward-interval next)** – Asserts the a transition can be made to a `SFL` state at some time point $t \in$ `[ R1, R1' ]` such that the accumulated reward until time point t lies in the interval `[ R2, R2' ]`.

`SFL1 U[ 0, R1 ][ 0, R2 ] SFL2` **(Time- & reward-bounded until)** – Asserts that `SFL2` is satisfied at some time instant $t \in$ `[ 0, R1 ]` such that the accumulated reward until t lies in the interval `[ 0, R2 ]`, and that at all preceding time instants `SFL1` holds.

8 Model Checking by Discrete Event Simulations

Since MRMC v1.3, we support model-checking by means of discrete event simulation. Being statistical in nature, such an approach cannot guarantee that the verification result is 100% correct. Yet, it allows to bound the probability of generating an incorrect answer to a verification problem, and, unlike the numerical approaches[1], model checking using simulations does not suffer from the state-space explosion. Note that, in the current implementation MRMC operates on the pre-generated Markov chain which is completely loaded into the computer's RAM[2], therefore the state-space explosion is not eliminated.

Techniques for model checking CSL (PCTL) properties using simulations have already been developed. For example in [YS02], later extended by [YS06], an algorithm based on Monte Carlo simulation and hypothesis testing for non-explosive stochastic discrete-event systems is suggested. In [SVA04], the algorithms of [YS02] are extended to statistically verify black-box, deployed systems with a passive observer. Both statistical approaches [YS02, SVA04] considered a sub-logic of CSL that excludes steady-state and unbounded-reachability properties. In [You04], the algorithm is extended to deal with a subclass of unbounded-reachability problems. In [SVA05] the statistical verification method of [YS02] is extended to verify unbounded-reachability properties of CSL (or PCTL) on finite-state CTMCs (DTMCs), and SMCs. All these approaches presume an "on-the-fly" model generation.

Contrary to the above mentioned techniques, our approach is based on Monte Carlo simulation and derivation of confidence intervals. We provide statistical algorithms for model checking the most interesting CSL operators, such as steady-state, unbounded-reachability, and time-interval reachability operators. In addition, when model checking unbounded-reachability or steady-state properties of CSL, we do simulations on the embedded DTMC. The latter simplifies simulation runs and also lets the corresponding techniques for model checking of PCTL properties on DTMCs to be easily derived. We do not consider nested simulation, see Section 6.2.3 on page 25, and working with finite-state systems, we assume that we can deduce the structure of the Markov chain. For instance we can detect Bottom Strongly Connected Components (BSCCs) of the Markov chain. For more details on the implemented algorithms, as well as comparison to the previously existing simulation techniques, consider reading Part 2 of [Zap08].

Of course, the quality and speed of simulations heavily depends on the quality and speed of the underlying *random number generator (RNG)*. For this reason seven different RNGs, which vary in many aspects, are available in MRMC. The ones with the best performance

[1] Numerical model checking is carried out by symbolic and numerical methods.
[2] Random access memory.

and reliability results are set to be used by default. For an extended experimental comparison of available RNG's, consider reading Appendix B.

The rest of this chapter is organized as follows. In Section 8.1 we introduce the main concepts of using confidence intervals in model checking. Further, in Section 8.2 we discuss the simulation engine of MRMC on the basis of several examples.

8.1 Confidence intervals and model checking

Let us consider the verification of the three most important operators of CSL: the unbounded-until operator $\mathrm{P}_{\bowtie b}\left(\mathcal{A}\ \mathrm{U}\ \mathcal{G}\right)$, the steady-state operator $\mathrm{S}_{\bowtie b}\left(\mathcal{G}\right)$, and the time-interval until operator $\mathrm{P}_{\bowtie b}\left(\mathcal{A}\ \mathrm{U}^{[t_1,t_2]}\ \mathcal{G}\right)$, with $t_1, t_2 \in \mathbb{R}_{\geq 0}$ and $t_1 \leq t_2$. We assume that $\bowtie \in \{<, \leq, >, \geq\}$ and, since we do not consider nested simulation, both $\mathcal{A}$ and $\mathcal{G}$ are treated as sets of states.

In order to verify the formulas $\mathrm{P}_{\bowtie b}\left(\mathcal{A}\ \mathrm{U}\ \mathcal{G}\right)$, $\mathrm{P}_{\bowtie b}\left(\mathcal{A}\ \mathrm{U}^{[t_1,t_2]}\ \mathcal{G}\right)$ or $\mathrm{S}_{\bowtie b}\left(\mathcal{G}\right)$, we apply the following procedure. First, for an initial state s_0 the probability $\widetilde{p}$ ($= \mathit{Prob}\left(s_0,\ \mathcal{A}\ \mathrm{U}\ \mathcal{G}\right)$, $= \mathit{Prob}\left(s_0,\ \mathcal{A}\ \mathrm{U}^{[t_1,t_2]}\ \mathcal{G}\right)$ or $= \mathit{Prob}^{\infty}\left(s_0,\ \mathcal{G}\right)$) is estimated in a form of the *c. i.* Second, the *c. i.* of $\widetilde{p}$ is checked against the probability constraint $\bowtie\ b$, to assess whether s_0 satisfies the given formula or not.

Leaving the task of computing the *c. i.* of $\widetilde{p}$ out of scope, further we concentrate on the second step of the outlined approach. There are two important reasons for that. First, this procedure is universal for all considered operators. Second, because of the probabilistic nature of the *c. i.*, the procedure should guarantee the correctness of the result with some (predefined) confidence.

Further, we split our discussion into three parts. First, we show how to decide on $\widetilde{p} \bowtie\ b$ when it is known that $\widetilde{p} \in [A_l,\ A_r]$. Then, we recall the notion of the *c. i.* of $\widetilde{p}$ and outline several problems related to the use of *c. i.* in validation of $\widetilde{p} \bowtie\ b$. Finally, we show how to overcome this problems, either by imposing some assumptions or by putting constraints on the width of the used *c. i.*

8.1.1 Simple problem

Let the value of $\widetilde{p}$ be unknown, but let us also know two bounds $A_l,\ A_r \in \mathbb{R}_{\geq 0}$ such that $A_l \leq \widetilde{p} \leq A_r$. In this setting, assessing whether $\widetilde{p} \bowtie\ b$ holds can be done based on the bounds A_l and A_r in a straightforward manner. Clearly, such an assessment, for all allowed $\bowtie$, is possible only if $b \notin [A_l,\ A_r]$ and thus the check yields three possible answers: positive ($TRUE$), negative ($FALSE$), or "Don't know" (NN).

8.1.2 Using confidence intervals

For a given confidence ξ and sample size $M \in \mathbb{N}_{\geq 2}$, the *c. i.* of $\widetilde{p}$ can be represented in the following form:

$$\mathit{Prob}\left(A_l\left(\overrightarrow{\mathbf{X}}\right) \leq \widetilde{p} \leq A_r\left(\overrightarrow{\mathbf{X}}\right)\right) \approx \xi, \tag{8.1}$$

where $\overrightarrow{\mathbf{X}}$ is a sample obtained via simulations of the given Markov chain. Equation (8.1) indicates that sampled intervals $\left[A_l\left(\overrightarrow{\mathbf{X}}\right),\ A_r\left(\overrightarrow{\mathbf{X}}\right)\right]$ contain $\widetilde{p}$ in about $100 \cdot \xi\,\%$ cases. The latter implies that using the *c. i.* of $\widetilde{p}$, for deciding $\widetilde{p} \bowtie\ b$, brings us two problems:

- If $b = \widetilde{p}$ then the solution of the model-checking problem is generally unknown. I.e., similar to model checking by means of hypothesis testing [YS06, SVA04, SVA05], the analysis based on the *c. i.* will be inconclusive. Clearly, in this case with probability ξ we have $\widetilde{p}, b \in \left[A_l\left(\overrightarrow{\mathbf{X}}\right),\ A_r\left(\overrightarrow{\mathbf{X}}\right)\right]$.
- Due to the probabilistic nature of the *c. i.*, the result of the comparison between the *c. i.* and constraint $\bowtie\ b$ becomes probabilistic itself. This means that, in order to give a correct answer to $\widetilde{p} \bowtie\ b$, it is not enough to check the *c. i.* of $\widetilde{p}$ against $\bowtie\ b$. In addition, we have to provide a confidence with which the result of such comparison provides a correct answer to the original problem.

8.1.3 Solving the problems

The first problem is generally unsolvable. Thus we can only assume that $|b - \widetilde{p}| = \delta$ with $\delta \in \mathbb{R}_{>0}$. Under this assumption, the second problem can be solved as follows.

Let us choose $\delta' \in \mathbb{R}_{>0}$ such that $\delta' < \delta$ and consider only *c. i.* borders $A_l\left(\overrightarrow{\mathbf{X}}\right), A_r\left(\overrightarrow{\mathbf{X}}\right)$ such that $A_r\left(\overrightarrow{\mathbf{X}}\right) - A_l\left(\overrightarrow{\mathbf{X}}\right) \leq \delta'$. Clearly, using such *c. i.* for deciding on $\widetilde{p} \bowtie\ b$ will guarantee us that in at least[3] $100 \cdot \xi\ \%$ cases we will be given a correct answer.

In the solution above, δ' is defined using δ which is unknown. Yet, it is clear that an incorrectly chosen δ' can be recognized by the fact that in repetitive simulations the combined percentage of "incorrect" and "Don't know" answers exceeds $100 \cdot (1 - \xi)\ \%$.

Note that, producing a δ'-tight *c. i.* is a matter of computing a sequential confidence interval. In MRMC we implemented a naive procedure where we increase the sample size until the *c. i.* becomes narrow enough. We realize that using this improper procedure can cause the decrease of the confidence levels, although this was not observed in our experiments, see Chapter 7 of [Zap08]. The description of a proper sequential *c. i.* derivation can be found in [Fis96, CR65].

Let us summarize that for a given confidence ξ and a maximum *c. i.* width δ' the simulation engine of MRMC guarantees to provide the correct answer to the model-checking problem if the following conditions hold:

1. $|b - \widetilde{p}| = \delta \in \mathbb{R}_{>0}$
2. $\delta' \in \mathbb{R}_{>0}$ and $\delta' < \delta$

Note that, in MRMC δ' corresponds to the value of `Indiff. reg. width`, manageable by the `set indiff_width R` command, see Section 6.2.3.

8.2 Simulation engine

In this section we provide several examples that explain how the simulation engine of MRMC can be used.

Example 6 *Consider the dice model depicted in Figure 2.2 on page 6. Let us forget about its rewards and assume that this model is a CTMC. Then if we invoke MRMC on this model, turn the simulation engine on and use the* `print` *command, we get the following:*

[3] An incorrect *c. i.* of $\widetilde{p}$ can still result in the correct answer to $\widetilde{p} \bowtie\ b$.

```
$ mrmc ctmc game.lab game.tra
...
>> set simulation on
>> print
---General settings:
...
M. C. simulation        = ON
...

---Monte Carlo simulation:
Simulation type         = ALL
Sim. steady state       = HYBRID
Reg. method steady      = HEURISTIC
Confidence level        = 9.500000e-01
Indiff. reg. width      = 2.000000e-02
Max sample size         = 100000
Min sample size         = 10000
Sample-size step type   = AUTO
Sample-size step        = 100
RNG discrete dist.      = Appl. Crypt.
RNG exponential dist.   = GSL Taus
Max simulation depth    = 100000
Min simulation depth    = 10000
Simulation-depth step   = 1000
BSCC dim. multiplier    = 3

---Numerical methods:
...
```

Here, for brevity, we omitted uninteresting parts of the output. Notice that, the section called `General settings` *indicates that the simulation engine is activated, and the newly appeared section* `Monte Carlo simulation` *contains most of the options, manageable by the commands given in Section 6.2.3. Note that, more options are available in case of doing simulations for one initial state:*

```
>>set sim_type one
>>print
---General settings:
...

---Monte Carlo simulation:
Simulation type         = ONE
Sim. initial state      = 1
...
```

Above, the simulation mode is changed and the new option `Sim. initial state` *indicates that the default initial state is* 1.

In the following example we are going to consider the most typical case of model-checking using the simulation engine:

Example 7 *Extending Example 6, let us be interested in a simple question: Is the probability to reach the* goal *state without visiting the* loss *state greater than* 0.3*? The latter can be expressed as the following CSL formula:* $\mathrm{P}_{>0.3}\,(\neg \mathit{loss}\ \mathrm{U}\ \mathit{goal})$.

As we would like to check the above formulated question by means of simulation, we invoke MRMC's simulation engine by typing `set simulation on`*. Providing MRMC with the formula above will cause the tool to run its model checking procedure:*

```
>>set simulation on
>>P{> 0.3} [ !loss U goal]
$SIMULATED: YES
$MAX_NUM_USED_OBSERV: 101944
$CONFIDENCE: 9.500000e-01
```

```
$CI_LEFT_RESULT: ( 0.1919024, 0.0000000, 0.1893418, 0.1974192, 1.0000000 )
$CI_RIGHT_RESULT: ( 0.2097583, 0.0000000, 0.2051940, 0.2114822, 1.0000000 )
$YES_STATE: { 5 }
$NO_STATE: { 1, 2, 3, 4 }

The Total Elapsed Model-Checking Time is 115 milli sec(s).
>>
```

As a result, we get four relevant outputs: two probability vectors `$CI_LEFT_RESULT` *and* `$CI_RIGHT_RESULT`*, as well as the two state sets* `$YES_STATE` *and* `$NO_STATE`*. The probability vectors correspond to the left and right* c.i. *borders derived for the first, second, etc. state of the model. Note that, the trivial probabilities, i. e.* 0.0 *and* 1.0*, are most likely to be computed via graph analysis. The* `$YES_STATE` *set contains the states in which the formula is satisfied. The* `$NO_STATE` *set contains states in which the formula is not satisfied. If the for a given state the simulation result is inconclusive, then it does not appear in any of the sets.*

In the output above, `$MAX_NUM_USED_OBSERV` *indicates the maximum – over all initial states – number of states that were considered in order to provide the answer for the given model-checking problem. More specifically, we count states visited during the simulation procedure. Therefore, the same model state is counted as many times as it is visited. On the other hand, we do not take into account state visits that occur during the model-graph analysis or numerical computations (for the case of hybrid simulation).*

The `$CONFIDENCE` *output tells us, that the results are correct with the* 95% *confidence. In is important to note that in case of nested formulas, when we have to simulate more than one operator, the confidence levels for sub formulas are derived from the overall confidence level. Their values then can be viewed by using the* `print tree` *command, see Section 6.1.*

In the following example we are going to explain two important cases: the output of the simulation results for one initial state; and an insufficient number of observations.

Example 8 *Extending Example 7, let us assume that we are only interested in verifying* $\mathrm{P}_{>0.3}\left(\neg \mathit{loss}\ \mathrm{U}\ \mathit{goal}\right)$ *in state* 3*. Also, we can be afraid of spending too much time on simulation and thus want to reduce the maximum sample size and simulation depth. The latter is important only for model checking the unbounded-until, or the steady-state (by pure simulation) operators. Then our interaction with MRMC might look as follows:*

```
>>set simulation on
>>set sim_type one
>>set initial_state 3
>>set min_sample_size 10
>>set max_sample_size 30
>>set min_sim_depth 10
>>set max_sim_depth 30
>>P{> 0.3} [ !loss U goal]
$SIMULATED: YES
$INITIAL STATE: 3
$MAX_NUM_USED_OBSERV: 308
$CONFIDENCE: 9.500000e-01
$CI_LEFT_RESULT: ( 0.1154063 )
$CI_RIGHT_RESULT: ( 0.3023792 )
$YES_STATE: {  }
$NO_STATE: {  }
$INDIFF_ERR_STATE: { 3 }
WARNING: Increase max_sample_size for obtaining the conf. int. of the desired width.

The Total Elapsed Model-Checking Time is 0 milli sec(s).
```

Here, we first set simulation mode to `one` *and then set the initial state to be* 3*. Next, we reduce the minimum and the maximum sample sizes and simulation depths. After that we invoke the model checking procedure. In this case the* c.i.*-border arrays have size* 1*. This can be checked by the following:*

```
>>$RESULT[1]
$CI_LEFT_RESULT[1] = 0.1154063
$CI_RIGHT_RESULT[1] = 0.3023792
>>$RESULT[3]
$CI_LEFT_RESULT[3] = ??
WARNING: Invalid index 3, required to be in the [1, 1] interval.
$CI_RIGHT_RESULT[3] = ??
WARNING: Invalid index 3, required to be in the [1, 1] interval.
```

Here, unlike in the previous example, the sets `$YES_STATE` *and* `$NO_STATE` *are empty. This should indicate that the simulation provides inconclusive results. Moreover, and* **this is an important part**, *a new set* `$INDIFF_ERR_STATE` *is added to the output. This set contains our initial state, i.e.* 3*. If this set appears in the output, it means that the max. number of observations (the max. sample size) and/or the max. simulation depth are not large enough to produce the* c.i. *tighter than the (specified) value of* `Indiff. reg. width`*, see Section 8.1. If this happens, the simulation run should be discarded, and the max. sample size / simulation depth values have to be increased.*

9 MRMC Test Suite

In order to keep MRMC bug free and to compare its performance to other model-checking tools (such as PRISM [KNP02], Ymer [You05b] and VESTA [SVA04]) we have developed a fully automated test suite featuring: internal, functional and performance tests.

The internal tests are targeted on testing, e. g., MRMC data structures, such as: sparse matrices, bit sets, sample vectors, and etc. The functional tests are used to assess the user-level behavior of the tool. This includes tests for the command-line interface, model-checking algorithms, and etc. Last but not least, the performance tests allow to evaluate the efficiency of implemented algorithms, such as: probabilistic bisimulation minimization, and "discrete event simulation" based model checking. Here, we consider several efficiency aspects: verification time, memory usage and etc.

The test suite contains well-known case studies: Wireless Group Communication Protocol (WGC) [MNS99, BCD02, MKL04], Simpel Peer-To-Peer Protocol (PTP) [KNP06], Workstation Cluster (WC) [HHK00, BKKT03, YKNP04, KNP02, KNP08b], Cyclic Server Polling System (CSP) [IT90, You05b, You05a, HKMKS00, SVA04, YKNP06, YS06], Randomized Mutual exclusion (RME) [PZ86], Crowds Protocol (CP) [RR98, KNP08a] and Synchronous Leader Election Protocol (SLE) [IR90, LP02, GSB94, FP04].

The test suite is freely distributed and can be obtained from:

http://www.mrmc-tool.org/

Note that, the test suite is intended to be used on a Linux platform only and its performance sub suite is not proven to work correctly under "Windows + Cygwin" or "Mac OS X". For the test-suite installation instructions see Section 3.2 of Chapter 3. The test-suite structure is as follows:

- `./TS_Manual.pdf` – The test-suite manual.
- `./LICENSE` – A copy of the GPL license.
- `./README` – The "read me" file.
- `./RELEASENOTES` – The release notes.
- `./settings.cfg` – The configuration script.
- `./test_all.sh` – The test-suite invocation script.
- `./clean_all.sh` – The test-suite "clean-up" script.
- `./stop.sh` – The test-run termination script.
- `./internal_tests/` – Unit tests of the MRMC core.

- `./functional_tests/` – Functional tests of MRMC.
- `./performance_tests/` – Performance tests of MRMC.

10 Contact

The development of MRMC began in 2004 in the Formal Methods and Tools group (FMT) at the University of Twente (The Netherlands) under the supervision of Prof. Dr. Ir. Joost-Pieter Katoen. Later, the main development of the tool was moved to the Software Modeling and Verification group at the RWTH Aachen (Germany). At present there are several other groups involved into the tool development, namely the Informatics for Technical Applications group at the Radboud University Nijmegen (The Netherlands), the Dependable Systems and Software group at the University of Saarland (Germany), and the Scientific Computing and Control Theory group at the Centrum voor Wiskunde en Informatica (The Netherlands).

If you have any questions, comments or ideas, or if you want to participate in MRMC development, please consider the following contact information:

Name: Prof. Dr. Ir. Joost-Pieter Katoen
Relation: The MRMC team leader, 2004 – present
Affiliation: Software Modeling and Verification, RWTH Aachen, Germany

Name: Dr. Ivan S. Zapreev
Relation: MRMC development, 2004 – present
Affiliation: Scientific Computing and Control Theory, Centrum voor Wiskunde en Informatica, The Netherlands

Name: Dr. David N. Jansen
Relation: MRMC extension and optimization, 2007 – present
Affiliation: Model-Based System Development, Radboud University Nijmegen, The Netherlands

Name: Prof. Dr.-Ing. Holger Hermanns
Relation: CTMDPI model checking, 2007 – present
Affiliation: Dependable Systems and Software, University of Saarland, Germany

More contact information can be found on the MRMC web-page [ZJN+08].

Bibliography

[ABFH+08] Cerion Armour-Brown, Jeremy Fitzhardinge, Tom Hughes, Nicholas Nethercote, Paul Mackerras, Dirk Mueller, Julian Seward, Robert Walsh, and Josef Weidendorfer, *Valgrind*, http://www.valgrind.org/, 2008.

[AHK03] Suzana Andova, H. Hermanns, and Joost-Pieter Katoen, *Discrete-Time Rewards Model-Checked*, Formal Modeling and Analysis of Timed Systems (FORMATS) (K.G. Larsen and P. Niebert, eds.), vol. 2791, LNCS, Springer, 2003, pp. 88–104.

[BCD02] Andrea Bondavalli, Andrea Coccoli, and Felicita Di Giandomenico, *QoS Analysis of Group Communication Protocols in Wireless Environment*, Concurrency in Dependable Computing (Paul Ezhilchelvan and Alexander Romanovsky, eds.), Kluwer Academic Publishers, 2002, pp. 169–188.

[BDH00] S. Bernardi, S. Donatelli, and A. Horváth, *Compositionality in the GreatSPN Tool and Its Application to the Modelling of Industrial Applications*, Practical Use of High-level Petri Nets (K. Jensen, ed.), University of Aarhus, Department of Computer Science, 2000, pp. 127–146.

[BFK+09] Tomás Brázdil, Vojtech Forejt, Jan Krcal, Jan Kretínský, and Antonín Kucera, *Continuous-time stochastic games with time-bounded reachability*, Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Leibniz International Proceedings in Informatics (LIPIcs), vol. 4, 2009, pp. 61–72.

[BFKT03] P. Buchholz, M. Fischer, P. Kemper, and C. Tepper, *Model checking of CTMCs and discrete event simulation integrated in the APNN-Toolbox*, Measurement, Modelling, and Evaluation of Computer-Communication Systems (F. Bause, ed.), vol. 781, Fachbereich Informatik, Universität Dortmund, 2003, pp. 30–33.

[BHH+06] Eckard Bode, Marc Herbstritt, Holger Hermanns, Sven Johr, Thomas Peikenkamp, Reza Pulungan, Ralf Wimmer, and Bernd Becker, *Compositional Performability Evaluation for STATEMATE*, Quantitative Evaluation of Systems (QEST), IEEE Computer Society, 2006, pp. 167–178.

[BHHK00] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen, *On the Logical Characterisation of Performability Properties*, International Colloquium on Automata, Languages and Programming (ICALP) (Ugo Montanari, Jos D. P. Rolim, and Emo Welzl, eds.), LNCS, vol. 1853, Springer, 2000, pp. 780–792.

[BHHK03] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, *Model-Checking Algorithms for Continuous-Time Markov Chains*, IEEE Transactions on Software Engineering **29** (2003), no. 6, 524–541.

[BHKH05] Christel Baier, Holger Hermanns, Joost-Pieter Katoen, and Boudewijn R. Haverkort, *Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes*, Theoretical Computer Science **345** (2005), no. 1, 2–26.

[BKKT03] P. Buchholz, J.-P. Katoen, P. Kemper, and C. Tepper, *Model-checking large structured Markov chains*, Journal of Logic and Algebraic Programming **56** (2003), 69–96.

[BM93] Jon L. Bentley and M. Douglas McIlroy, *Engineering a sort function*, Software: practice and experience **23** (1993), no. 11, 1249–1265.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, AMC Transactions On Programming Languages And Systems **8** (1986), no. 2, 244–263.

[CG04] Frank Ciesinski and Marcus Größer, *On Probabilistic Computation Tree Logic*, Validation of Stochastic Systems (Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, eds.), LNCS, vol. 2925, Springer, 2004, pp. 147–188.

[CKKP05] L. Cloth, J.-P. Katoen, M. Khattri, and R. Pulungan, *Model-Checking Markov Reward Models with Impulse Rewards.*, Dependable Systems and Networks (DSN), IEEE Computer Society, 2005, pp. 722–731.

[CL77] A. A. Crane and J. Lemoine, *An introduction to the regenerative method for simulation analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1977.

[CR65] Y. S. Chow and H. Robbins, *On the asymptotic theory of fixed-width sequential confidence intervals for the mean*, Annals of Mathematical Statistics **36** (1965), no. 2, 456–462.

[DHS03] Salem Derisavi, Holger Hermanns, and William H. Sanders, *Optimal state-space lumping in Markov chains*, Information processing letters **87** (2003), 309–315.

[FG88] Bennett L. Fox and Peter W. Glynn, *Computing Poisson probabilities*, Communications of the ACM **31** (1988), no. 4, 440–445.

[Fis96] George S. Fishman, *Monte Carlo: Concepts, Algorithms and Applications*, Springer, New York, NY, USA, 1996.

[Fou07] Eclipse Foundation, *Eclipse*, http://www.eclipse.org, 2007.

[FP04] W. Fokkink and J. Pang, *Simplifying Itai-Rodeh leader election for anonymous rings*, Electronic Notes in Theoretical Computer Science **128** (2004), no. 6, 53–68.

[GSB94] Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar, *On randomization in sequential and distributed algorithms*, ACM Computing Surveys **26** (1994), no. 1, 7–86.

[HCH+02] B. Haverkort, L. Cloth, H. Hermanns, J.-P. Katoen, and C. Baier, *Model Checking Performability Properties*, Dependable Systems and Networks (DSN), IEEE Computer Society, 2002, pp. 103–112.

[Her] Holger Hermanns, *Homepage of the Dependable Systems group*, http://depend.cs.uni-sb.de.

[HHK00] B. Haverkort, H. Hermanns, and J.-P. Katoen, *On the Use of Model Checking Techniques for Dependability Evaluation*, Symposium on Reliable Distributed Systems (SRDS), IEEE Computer Society, 2000, pp. 228–237.

[Hil96] Jane Hillston, *A Compositional Approach to Performance Modelling*, Distinguished Dissertations Series, Cambridge University Press, New York, NY, USA, 1996.

[HJ94] N. Hansson and B. Jonsson, *A logic for reasoning about time and reliability*, Formal Aspects of Computing **6** (1994), no. 5, 512–535.

[HKMKS00] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle, *A Markov Chain Model Checker*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Susanne Graf and Michael Schwartzbach, eds.), LNCS, vol. 1785, Springer, 2000, pp. 347–362.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, *PRISM: A Tool for Automatic Verification of Probabilistic Systems*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (H. Hermanns and J. Palsberg, eds.), LNCS, vol. 3920, Springer, 2006, pp. 441–444.

[IR90] Alon Itai and Michael Rodeh, *Symmetry breaking in distributed networks*, Information and Computation **88** (1990), no. 1, 60–87.

[IT90] Oliver C. Ibe and Kishor S. Trivedi, *Stochastic Petri Net Models of Polling Systems*, Selected Areas in Communications **8** (1990), no. 9, 1649–1657.

[JKO+07] David N. Jansen, Joost-Pieter Katoen, Marcel Oldenkamp, Mariëlle Stoelinga, and Ivan S. Zapreev, *How Fast and Fat Is Your Probabilistic Model Checker?*, Haifa Verification Conference (HVC), LNCS, vol. 4899, Springer, 2007, pp. 65 – 79.

[KKNP01] J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker, *Faster and Symbolic CTMC Model Checking*, Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV) (Luca de Alfaro and Stephen Gilmore, eds.), LNCS, vol. 2165, Springer, 2001, pp. 23–38.

[KKZJ07] Joost-Pieter Katoen, Tim Kemna, Ivan S. Zapreev, and David N. Jansen, *Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Orna Grumberg and Michael Huth, eds.), LNCS, vol. 4424, Springer, 2007, pp. 87–101.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker, *PRISM: Probabilistic Symbolic Model Checker*, Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS) (T. Field, P. Harrison, J. Bradley, and U. Harder, eds.), LNCS, vol. 2324, Springer, 2002, pp. 200–204.

[KNP06] _____, *Symmetry Reduction for Probabilistic Model Checking*, Computer Aided Verification (CAV) (T. Ball and R. Jones, eds.), LNCS, vol. 4114, Springer, 2006, pp. 234–248.

[KNP08a] _____, *Prism case studies*, http://www.prismmodelchecker.org/casestudies/, 2008.

[KNP08b] _____, *Prism web-page, Workstation Cluster Example*, http://www.prismmodelchecker.org/casestudies/cluster.php, 2008.

[Knu01] Timo Knuutila, *Re-describing an algorithm by hopcroft*, Theoretical computer science **250** (2001), no. 1–2, 333–363.

[KZ05] J.-P. Katoen and Ivan S. Zapreev, *Safe On-The-Fly Steady-State Detection for Time-Bounded Reachability*, Tech. Report TR-CTIT-05-52, CTIT, University of Twente, 2005.

[KZ06] Joost-Pieter Katoen and Ivan S. Zapreev, *Safe On-The-Fly Steady-State Detection for Time-Bounded Reachability*, Quantitative Evaluation of Systems (QEST), IEEE Computer Society, 2006, pp. 301–310.

[KZ09] _____, *Simulation-Based CTMC Model Checking: An Empirical Evaluation*, Quantitative Evaluation of Systems (QEST), IEEE Computer Society, 2009, www.mrmc-tool.org, pp. 31–40.

[KZH+09] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen, *The Ins and Outs of The Probabilistic Model Checker MRMC*, Quantitative Evaluation of Systems (QEST) (Los Alamitos, Calif.), IEEE Computer Society, 2009, www.mrmc-tool.org, pp. 167–176.

[KZH+10] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen, *The ins and outs of the probabilistic model checker MRMC*, Performance evaluation (2010), DOI:10.1016/j.peva.2010.04.001.

[LP02] Richard Lassaigne and Sylvain Peyronnet, *Approximate verification of probabilistic systems*, Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV) (Holger Hermanns and Roberto Segala, eds.), Springer, 2002, pp. 213–214.

[MKL04] Mieke Massink, Joost-Pieter Katoen, and Diego Latella, *Model Checking Dependability Attributes of Wireless Group Communication*, Dependable Systems and Networks (DSN), IEEE Computer Society, 2004, pp. 711–720.

[MN98] Makoto Matsumoto and Takuji Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation **8** (1998), no. 1, 3–30.

[MNS99] Michael Mock, Edgar Nett, and Stefan Schemmer, *Efficient Reliable Real-Time Group Communication for Wireless Local Area Networks*, European Dependable Computing Conference (Jan Hlavicka, Erik Maehle, and Andrs Pataricza, eds.), LNCS, vol. 1667, Springer, 1999, pp. 380–400.

[MS76] Ian Munro and Philip M. Spira, *Sorting and searching in multisets*, SIAM journal of computation **5** (1976), no. 1, 1–8.

[PM88] Stephen K. Park and Keith W. Miller, *Random Number Generators: Good Ones Are Hard to Find*, Commun. ACM **31** (1988), no. 10, 1192–1201.

[PtFSF07a] GNU Project and the Free Software Foundation, *GNU General Public License (GPL)*, http://www.gnu.org/copyleft/gpl.html, 2007.

[PtFSF07b] ———, *GNU Scientific Library (GSL)*, http://www.gnu.org/software/gsl, 2007.

[PZ86] A. Pnueli and L. Zuck, *Verification of Multiprocess Probabilistic Protocols*, Distributed Computing **1** (1986), no. 1, 53–72.

[QS96] M. A. Qureshi and W. H. Sanders, *A New Methodology for Calculating Distributions of Reward Accumulated During a Finite Interval*, Fault-Tolerant Computing, IEEE Computer Society, 1996, pp. 116–125.

[RR98] M. K. Reiter and A. D. Rubin, *Crowds: Anonymity for Web Transactions*, ACM Transactions on Information and System Security, vol. 1, ACM Press, 1998, pp. 66–92.

[Sch95] Bruce Schneier, *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*, John Wiley & Sons, Inc., New York, NY, USA, 1995.

[SVA04] Koushik Sen, Mahesh Viswanathan, and Gul Agha, *Statistical Model Checking of Black-Box Probabilistic Systems*, Computer Aided Verification (CAV) (Rajeev Alur and Doron A. Peled, eds.), LNCS, vol. 3114, Springer, 2004, pp. 202–215.

[SVA05] ———, *On Statistical Model Checking of Stochastic Systems*, Computer Aided Verification (CAV) (Kousha Etessami and Sriram K. Rajamani, eds.), LNCS, vol. 3576, Springer, 2005, pp. 266–280.

[TG06] Mirco Tribastone and Stephen Gilmore, *A New Generation PEPA Workbench*, Process Algebra and Stochastically Timed Activities (PASTA), 2006, pp. 1820–1845.

[TV00] H. C. Tijms and R. Veldman, *A fast algorithm for the transient reward distribution in continuous-time Markov chains*, Operations Research Letters **26** (2000), no. 4, 155–158.

[VL08] Antti Valmari and Petri Lehtinen, *Efficient minimization of dfas with partial transition functions*, 25th international symposium on theoretical aspects of computer science (STACS 2008) (Dagstuhl, Germany) (Susanne Albers and Pascal Weil, eds.), Leibniz International Proceedings in Informatics (LIPIcs), vol. 1, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008, pp. 645–656.

[YKNP04] H. Younes, M. Kwiatkowska, G. Norman, and D. Parker, *Numerical vs. Statistical Probabilistic Model Checking: An Empirical Study*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (K. Jensen and A. Podelski, eds.), LNCS, vol. 2988, Springer, 2004, pp. 46–60.

[YKNP06] Håkan Younes, Marta Kwiatkowska, Gethin Norman, and David Parker, *Numerical vs. Statistical Probabilistic Model Checking*, Software Tools for Technology Transfer (STTT) **8** (2006), no. 3, 216–228.

[You04] H. Younes, *Black-box probabilistic verification*, Tech. Report CMU-CS-04-162, Carnegie Mellon University, 2004.

[You05a] _____, *Verification and Planning for Stochastic Processes with Asynchronous Events*, Ph.D. thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[You05b] _____, *Ymer: A Statistical Model Checker*, Computer Aided Verification (CAV) (Kousha Etessami and Sriram K. Rajamani, eds.), LNCS, vol. 3576, Springer, 2005, pp. 429–433.

[YS02] Håkan Younes and Reid Simmons, *Probabilistic Verification of Discrete Event Systems using Acceptance Sampling*, Computer Aided Verification (CAV) (Ed Brinksma and Kim Guldstrand Larsen, eds.), LNCS, vol. 2404, Springer, 2002, pp. 223–235.

[YS06] H. Younes and R. Simmons, *Statistical Probabilistic Model Checking with a Focus on Time-Bounded Properties*, Information and Computation **204** (2006), no. 9, 1368–1409.

[Zap08] I. S. Zapreev, *Model Checking Markov Chains: Techniques and Tools*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 2008.

[ZJN+08] Ivan S. Zapreev, Christina Jansen, Viet Yen Nguyen, David N. Jansen, et al., *MRMC homepage*, http://www.mrmc-tool.org/, 2008.

A CTMDPI: Model examples

This appendix describes CTMDPI models supported by MRMC. These are the input models for the CTMDPI model-checking component [BHKH05] of MRMC, developed by the Dependable Systems and Software group [Her] of the Saarland University.

A.1 Markov decision processes

In general, Markov decision processes (MDPs), and CTMDPIs in particular, are similar to Markov chains, except that in addition to the stochastic transitions they also allow for the non-deterministic ones. The non-determinism introduced by them is supposed to be resolved by some scheduler.

Typically, MDPs are expected to have an initial distribution. However, we will assume that there is just one initial state, namely the state 1 of any given CTMDPI model.

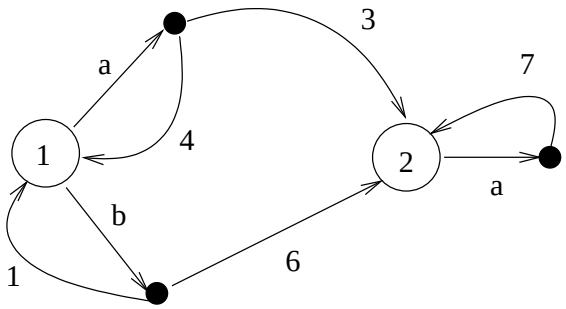


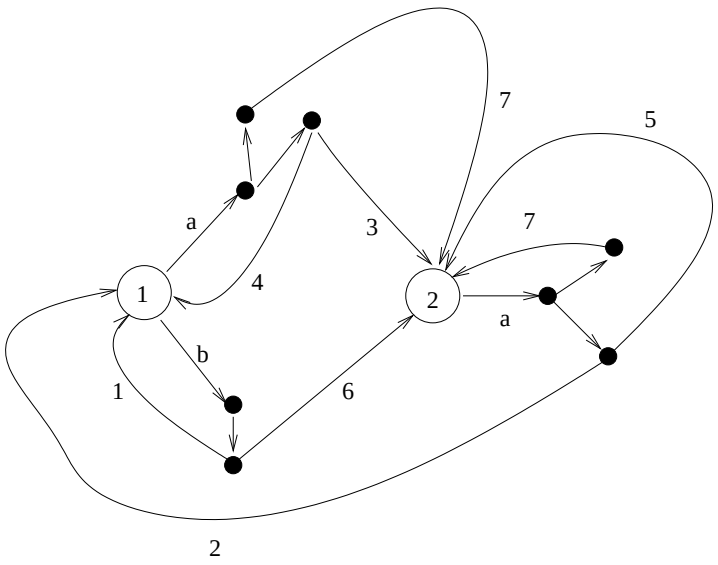
Figure A.1: A CTMDP example

Example 9 *An example CTMDP is depicted in Figure A.1. This model contains only two states:* 1 *and* 2*. For the first one, a scheduler can choose between two transitions, namely* a *and* b*. If the choice is done in favor of the first one, then further we have a probabilistic choice defined by the rate* 3 *of going to state* 2 *and the rate of* 4 *of going back to state* 1*. Alternative, if the scheduler chooses* b*, the rate of returning to state* 1 *is only* 1 *and to state* 2 *is* 6*. State* 2 *does not have a true non-deterministic choice, because there is only one non-deterministic transition present.*

It becomes clear now, that with MDP models, like with simple CTMCs, one can be interested in computing, e. g., reachability probabilities and etc. The only difference is that, since we can have any possible scheduler, we have to talk about minimal and maximal probabilities. All this implies that we can actually do model checking of CTMDPs.

A.1.1 Markov decision processes with internal non determinism

The CTMDP described before has only one level of non determinism. It is also possible to have CTMDPs with two layers of non-determinism, in this case we call them CTMDPIs. On the first layer, an external scheduler takes a decision, then an internal decision occurs, and after this the probabilistic decision takes place.




```
STATES 2
#DECLARATION
#END
1 a
* 2 7.0
1 a
* 1 4.0
* 2 3.0
1 b
* 1 1.0
* 2 6.0
2 a
* 2 7.0
2 a
* 2 5.0
* 1 2.0
```

Figure A.2: A CTMDPI example

Example 10 *Consider a CTMDPI model given in the left-hand side of Figure A.2. In this model, state* 1 *has two external non-determinism choices:* `a` *and* `b`. *If decision* `a` *is taken, then there is an internal non-deterministic choice. One branch of it leads to going to state* 2 *with the rate* 7. *The other one leads to going to state* 2 *with the rate* 3 *and to state* 1 *with the rate* 4. *Decision* `b` *leads to a trivial internal non-determinism. State* 2 *has an internal non-determinism as well.*

The CTMDP examples above are given by state-transitions, corresponding distributions, and labeling functions that map sets of labels to the transitions. Note that, for model checking we also need to provide state labeling functions (in our cases the set of state labels are empty).

In order to be used with MRMC, CTMDPIs have to be transformed into the MRMC input files that have an extension `.ctmdpi`. For example, the model given in the left-hand side of Figure A.2 results in a file given in right-hand side of the same figure. It is important to note, that CTMDPI model checking uses CSL for specifying properties. At present we only support time-bounded reachability properties. Similar to the CTMC model checking, these properties are based on state labels that have to be specified in a `.lab` file. The transition labels are needed only for the CTMDPI model-checking engine and should not be used in properties.

B RNG Investigations

We define random number generators (RNGs) as algorithms that allow to generate uniformly distributed random numbers for a prescribed real interval.

RNG implementations are commonly used in programming and especially in discrete-event simulation engines. In MRMC we use RNGs for simulating discrete or exponentially distributed random variables. The first ones are used to simulate the probabilistic choice between state-transitions and the second ones are employed to simulate exponentially distributed waiting times of CTMC states.

Nowadays, there exist many RNGs, but often these generators vary in various aspects. For example, they can differ in: time needed to calculate a random number or the quality of their output. The latter aspect can be split in (at least) two parts: one generator can calculate more equidistributional random numbers than the other; different generators can have different *periods*, i. e. the number of method invocations after which the generated random numbers start to repeat in a circular manner. In our experiments, though, we mainly concentrated on how good RNGs are for generating values of non-uniform discrete and exponentially distributed random variables. This was done by accessing the speed of random-number generation and the correspondence of the sampled distributions to the original ones.

To choose which generator is better and can be used as a default one in MRMC, we tested seven different RNGs. Some of them were taken because they already made it into probabilistic model checkers such as PRISM, Ymer or VESTA, the others are widely used in industry, and etc.

The rest of the appendix is organized as follows: Section B.1 presents the description of the considered RNGs. Further, in Section B.2, we explain how RNGs can be used for generating values of non-uniform discrete and exponentially distributed random variables, and present the experimental setup. Section B.3 provides the experimental results and comparison.

B.1 Random Number Generators

Here, we provide a short summary of the tested RNGs, and also indicate the MRMC option values corresponding to each of them.

B.1.1 Linear Congruential Generator (LCG) – `prism`

LCG is the oldest and mostly used random-number generator algorithm. A sequence of random numbers is calculated according to the formula $x_{n+1} = (a * x_n + c) \, mod \, m$, where x_0 denotes the seed (the initial value) and m is the RNG's period. The considered LCG is implemented as the random function *rand()* of the standard C library (gcc). The use of *rand()* was taken into account, because PRISM uses it in its simulation engine. However, it should

be noted that the C random function is known to suffer from a low period. Even the 32-bit version of it can only offer a period of $m = 2^{32}$.

B.1.2 Improved LCG [PM88] (ILCG) – `ciardo`

ILCG is a version of LCG, developed by Steve Park and Keith Miller. It works similar to the Standard C random function, but is known to generate more equidistributional random numbers. Therefore, it is often proposed to be used instead of *rand()*, although it also has a small period of $m = 2^{32}$.

B.1.3 Combined LCG [Sch95] (CLCG) – `app_crypt`

This RNG is another extension of the standard LCG. The main advantage of this method is that, by using two independent LCGs, it increases the period up to about $m = 2^{64}$. Note that, in most cases it is more efficient to combine two LCGs than taking one with a much larger modulus (period). CLCG is widely used in the field of Cryptography.

B.1.4 Mersenne Twister [MN98] (Twister) – `ymer`

The Mersenne Twister is a random-number generator developed by Makoto Matsumoto and Takuji Nishimura in 1997. Today, there exist several variants of this algorithm. We have chosen Mersenne Twister MT19937 (32-bit version), because it is the newest and most commonly used one. This algorithm is also employed by Ymer and comes with a large period of $m = 2^{19937} - 1$.

B.1.5 RNGs from GSL [PtFSF07b]

RNGs introduced in this section are a part of the GNU Scientific Library (GSL).

Ranlux Generator (Ranlux) – `gsl_ranlux`

According to the GSL documentation, the implemented RANLUX algorithm is a second-generation version of the RANLUX algorithm of Lüscher and has a period of about $m = 10^{171}$. GSL developers recommend this algorithm as the one with the best mathematically-proven quality at the expense of performance.

Lagged Fibonacci Generator (LFG) – `gsl_lfg`

According to the GSL documentation, LFG produces random numbers as xor'd sum of previously calculated values on the basis of the following formula:

$$r_n = r_{n-A} \text{ XOR } r_{n-B} \text{ XOR } r_{n-C} \text{ XOR } r_{n-D}$$

with $A = 471$, $B = 1586$, $C = 6988$, $D = 9689$. This RNG has a period of $m = 10^{2917}$ and is recommended by GSL developers as a fast simulation-quality generator.

Tausworthe Generator (Tausworthe) – `gsl_taus`

According to the GSL documentation this is a maximally equidistributed combined Tausworthe generator (or polynomial generator) by L'Ecuyer with a period of $m = 2^{88}$ (about 10^{26}). Like the lagged Fibonacci generator, the Tausworthe generator is recommended by GSL developers as a fast simulation-quality generator (which is faster than LFG).

B.2 Experimental setup

In this section, we describe the experimental setup used for the evaluation of the before mentioned RNGs, in application to generation of non-uniform discrete and exponentially distributed random variables.

In essence, our approach is based on taking a random variable with a particular distribution and sampling a set of its values (produced with the help of a particular RNG). These values are then used for computing the estimate of the underlying distribution. The latter one is compared to the original distribution of the random variable. The main values measured in our experiments (per distribution), are as follows:

1. The time needed for generating a random values when using a particular RNG.
2. The difference between the estimated and original distributions.

B.2.1 Non-Uniform Discrete Random Variables

Generation of non-uniformly distributed discrete random numbers, employing standard RNGs mentioned in Section B.1, is typically done in the following manner.

Let us have a discrete random variable x with a finite set of values $x_1, \ldots, x_n$. The value x_i is then produced with probability p_i for any $i \in 1, \ldots, n$ and $\sum_{i=1}^{n} p_i = 1.0$. Let us now have an RNG which generates us random numbers in the interval $[A,\ B]$ with $0 \leq A < B$. Then, to generate values of x we should perform the following steps:

1. Split the real interval $[0,\ 1]$ into n fixed non-overlapping intervals $I_1, \ldots, I_n$ such that the width of I_i equals to p_i for any $i \in 1, \ldots, n$.
2. Generate a uniformly-distributed random number C and scale it down using the formula $C/(B - A)$. This way we obtain the value in the interval $[0,\ 1]$.
3. Find $j \in 1, \ldots, n$ such that $C/(B - A) \in I_j$. This j exists because $\{I_i\}_{i=1}^{n}$ forms a coverage of $[0,\ 1]$.
4. Return x_j as the value of the random variable x.

Clearly, step $1.$ has to be performed only once and states $2.$ to $3.$ result in values of x that agree to its distribution.

Test Distributions

For our experiments we have chosen six different probability distributions. Each of these distributions had 100 values, most of which with non-zero probabilities.

1. The standard uniform distribution ("Unif").
2. A non-uniform distribution ("Diff"), where one value appears with a very high probability (0.899924), and all other values have very small or zero probabilities.
3. The "Lorentz" distribution[1] with the largest and smallest probabilities being equal to 0.013151 and 0.00685 respectively.
4. Three distributions: "Pow2", "Pow3" and "Pow4". For each $X \in \{2,\ 3,\ 4\}$, "Pow$X$" was generated as follows:
   a) Generate 100 random values using a uniform distribution on the interval $[0,\ 1]$.
   b) Take these values to the power X.
   c) Normalize the resulting values in such a way that they sum up to one.
   d) Take the new values as probabilities for the distribution on $1, \ldots, 100$.

Test Settings

For a given RNG R and a distribution D every distribution estimate was computed based on $1.000.000$ sampled values. Also, for every given D and R, we computed 50 distribution estimates.

The run time for R on D was calculated as a mean time needed for generating 50 distribution estimates. The quality of each R on D was estimated based on the following quantitative value:

$$\sum_{i=1}^{100} \frac{1}{50} \sum_{j=1}^{50} \frac{\left|p_i^j - p_i\right|}{p_i}, \tag{B.1}$$

where $\{p_i\}_{i=1}^{100}$ is the set of probability values of the original distribution and $\left\{ \{p_i^j\}_{i=1}^{100} \right\}_{j=1}^{50}$ are the probabilities of the 50 sampled distributions.

B.2.2 Exponentially Distributed Random Variables

The exponential distribution is a probability distribution over the set of positive real numbers. In order to generate values of an exponentially-distributed random variable, we use the commonly known inversion method: if u is a uniformly-distributed random variable then

$$x := -\frac{1}{\lambda} ln(1-u)$$

has exponential distribution with the rate λ. As an optimization, we use formula:

$$x := -\frac{1}{\lambda} ln(u),$$

[1] a well-known probability distribution in physics

since $1 - u$ is a uniformly-distributed random variable itself.

Test Distributions

We considered exponential distributions with $\lambda \in \{0.01,\ 0.1,\ 0.5,\ 1.0,\ 5.0,\ 10.0\}$.

Test Settings

For every given λ (distribution E_λ) and every RNG R we sampled $10.000.000$ random values.

The run time for R on E_λ was calculated as a total time needed for generating all of these values. Since exponential distribution is continuous, the quality of each R on E_λ was estimated using discretization:

1. Compute M – the maximum over all simulated values.
2. For $\delta = 0.3$, compute $N = M/\delta + 1$ – the number of δ intervals that form a partitioning of the simulated values: $\{I_i\}_{i=1}^{N}$ where $I_i = [(i-1) * \delta,\ i * \delta)$[2]
3. Define $P_i = \mathit{Prob}\,(X \in I_i)$ for $i \in 1, \ldots, N$ and X being a random variable with the distribution E_λ.
4. Define $P'_i = S_i/10^7$ for $i \in 1, \ldots, N$ and S_i being the number of simulated values that fall into the interval I_i.

This process gives us a discrete distribution: for any $i \in 1, \ldots, N$ we have I_i with probability P_i; and an estimate of this distribution: defined by the values of $\{P'_i\}_{i=1}^{N}$. The quality of R on D was then estimated based on the quality of the discritized exponential distribution and its discritized estimate. This was done by computing the following quantitative value:

$$\sum_{i=1}^{N} \frac{|P_i - P'_i|}{\delta}. \tag{B.2}$$

Note that, this formula is different from the one given by Equation B.1. Here we divide $|P_i - P'_i|$ by δ because we are interested in the quality with which we approximate the density function of the original (continuous) distribution.

B.3 RNG comparison - results

All experiments were done on a standard PC with an AMD® Athlon® CPU $3000+$ processor (64-bit) and and 2 GB of RAM. The used operating system was openSuSE 10.2.

B.3.1 Non-Uniformly Random Numbers

A brief summary of the obtained results can be found in Table B.1.

[2] In our experiments, we had probabilities over intervals: $[0.0,\ 0.3), \ldots, [2.7,\ 3.0)$ for $\lambda \in \{0.01,\ 0.1,\ 0.5,\ 1.0,\ 5.0\}$; and $[0.0,\ 0.3), \ldots, [1.2,\ 1.5)$ for $\lambda = 10.0$.

Position	Speed	Simulation Quality
1.	LFG Tausworthe CLCG	Ranlux
2.	Twister LCG ILCG	LFG Tausworthe CLCG Twister ILCG
3.	Ranlux	LCG

Table B.1: Non-uniform discrete random variables

Run time

The time needed for generating 1.000.000 random values for the considered RNGs on corresponding distributions is provided in Figure B.1. The quality of every RNG on every distribution is summarized in Table B.2.

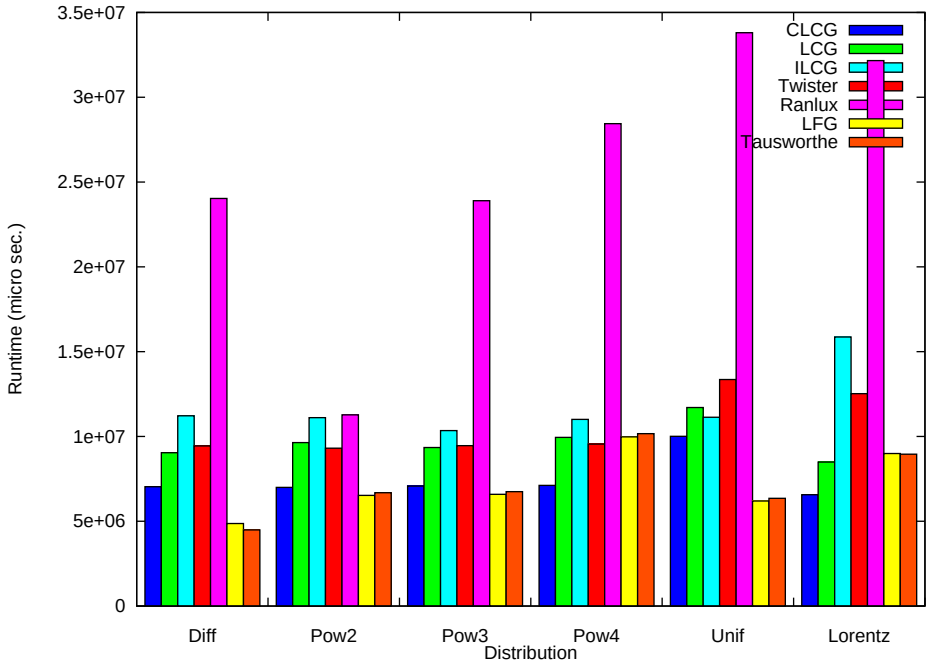


Figure B.1: Run time: Non-uniform discrete random variables

Having a closer look at Figure B.1 and Table B.2, the results can be formulated as follows. From the run-time point of view, the RNG with worst performance is clearly the Ranlux Generator, which positioned itself behind all other RNGs in every of the six test cases. The first three places are fought out between Tausworthe, LFG and CLCG in three out of six test cases, whereat they still gained leading positions in the remaining three cases. By looking at the plots in detail, one may notice that LFG and Tausworthe have similar results in every test case, whereas CLCG is remarkably faster in “Lorentz” and “Pow4”, remarkably slower in

Pos.	Distribution					
	Diff	**Pow2**	**Pow3**	**Pow4**	**Unif**	**Lorenz**
1	Tausworthe	LFG	LFG	CLCG	LFG	CLCG
2	LFG	Tausworthe	Tausworthe	Twister	Tausworthe	LCG
3	CLCG	CLCG	ILCG	LFG	CLFG	Tausworthe
4	LCG	Twister	LCG	LCG	ILCG	LFG
5	Twister	LCG	Twister	Tausworthe	LCG	Twister
6	ILCG	ILCG	ILFG	ILCG	Twister	ILCG
7	Ranlux	Ranlux	Ranlux	Ranlux	Ranlux	Ranlux

Table B.2: Run time: Non-uniform discrete random variables

"Diff" and "Unif" distribution. Summarized, LFG and Tausworthe positioned themselves in first place, closely followed by CLCG. The middle-ranked RNGs are then LCG, ILCG and Twister with similar results, with ILCG tending to be the slowest out of this three RNGs, except from the "Unif" test case, and with Twister and LCG swapping positions from case to case.

Sums of average errors

The sum of average errors for the considered RNGs on corresponding distributions is provided in Figure B.2. The quality of every RNG on every distribution is summarized in Table B.2.

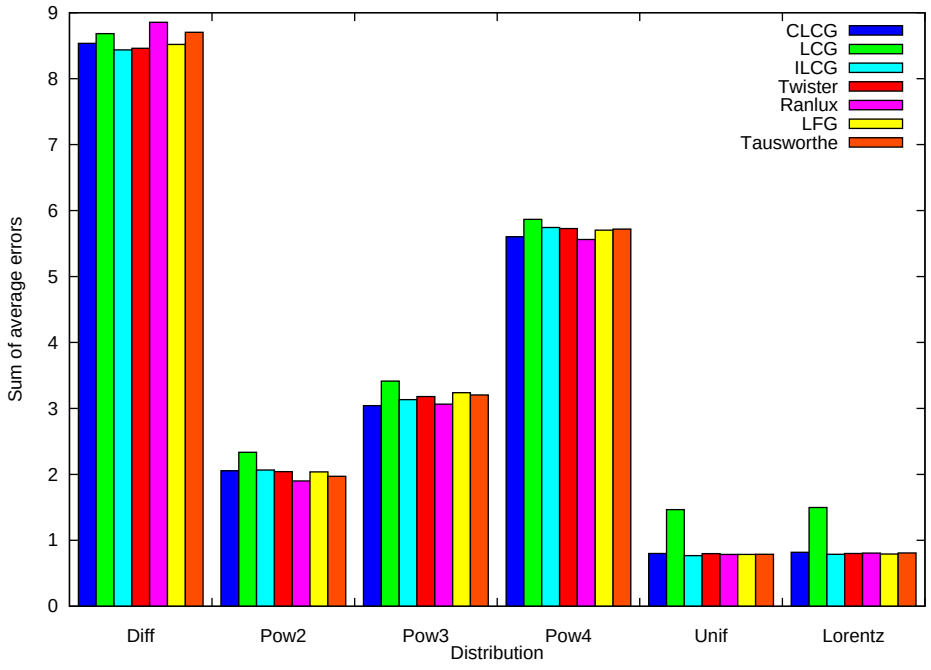


Figure B.2: Sums of average errors: Non-uniform discrete random variables

The results of Figure B.2 and Table B.3 can be formulated as follows. From the simulation-quality point of view, the Ranlux Generator positioned itself in leading position. It gained

Pos.	Distribution					
	Diff	**Pow2**	**Pow3**	**Pow4**	**Unif**	**Lorenz**
1	ILCG	Ranlux	CLCG	Ranlux	ILCG	ILCG
2	Twister	Tausworthe	Ranlux	CLCG	Ranlux	LFG
3	LFG	LFG	ILCG	LFG	LFG	Twister
4	CLCG	Twister	Twister	Tausworthe	Tausworthe	Ranlux
5	LCG	CLCG	Tausworthe	Twister	Twister	Tausworthe
6	Tausworthe	ILCG	LFG	ILCG	CLCG	CLCG
7	Ranlux	LCG	LCG	LCG	LCG	LCG

Table B.3: Sums of average errors: Non-uniform discrete random variables

first/second places in four out of six test cases. Only for the "Diff" distribution Ranlux Generator produces poor results compared to the remaining RNGs. Furthermore, as LCG produced worst results in five out of six test cases, it clearly can be seen as the RNG with the poorest simulation quality in our testing environment. Although the simulation quality middle-ranked RNGs aren't clearly distinguishable, one can detect some trends there. LFG obtains position three with most stability, whereas ILCG shows the biggest difference in positioning through the whole test cases. Twister can mostly be found around places four to five, CLCG and Tausworthe mostly show up on places four to six. Summarized, no clear ordering can be found for the middle-ranked RNGs.

B.3.2 Exponentially Distributed Random Numbers

A brief summary of the obtained results can be found in Table B.4.

Position	Speed	Simulation Quality
1.	CLCG	Ranlux
2.	LFG Twister Tausworthe LCG ILCG	LFG Twister CLCG Tausworthe ILCG LCG
3.	Ranlux	

Table B.4: Exponentially distributed random variables

Run time

The time needed for generating 10.000.000 random values for the considered RNGs on corresponding distributions is provided in Figure B.3. The quality of every RNG on every

distribution is summarized in Table B.5.

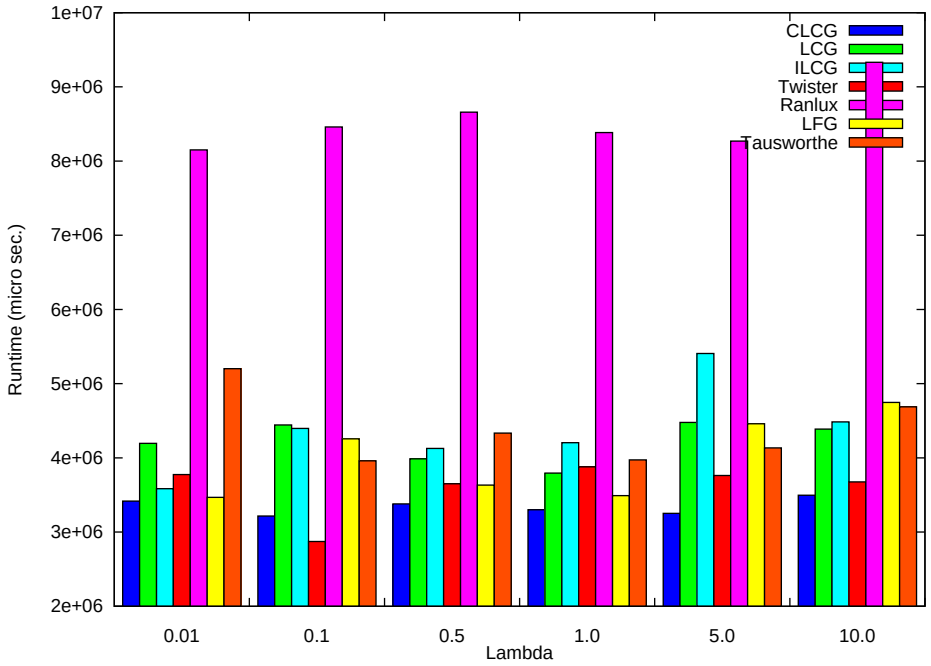


Figure B.3: Run time: Exponentially distributed random variables

Pos.	λ					
	0.01	**0.1**	**0.5**	**1.0**	**5.0**	**10.0**
1	CLCG	Twister	CLCG	CLCG	CLCG	CLCG
2	LFG	CLCG	LFG	LFG	LFG	Twister
3	ILCG	Tausworthe	Twister	LCG	LCG	LCG
4	Twister	LFG	LCG	Twister	Twister	ILCG
5	LCG	ILCG	ILCG	Tausworthe	Tausworthe	Tausworthe
6	Tausworthe	LCG	Tausworthe	ILCG	ILCG	LFG
7	Ranlux	Ranlux	Ranlux	Ranlux	Ranlux	Ranlux

Table B.5: Run time: Exponentially distributed random variables

Having a closer look at Figure B.3 and Table B.5, the results can be formulated as follows. From runtime point of view CLCG can be seen as the winner over all considered RNGs. In five out of six test cases it was placed first, for the sixth case CLCG was placed second. When looking at pure runtime, Ranlux Generator again showed the worst performance of all RNGs and all test cases examined. The middle-ranked RNGs can barely be ordered. ILCG and Tausworthe Generator shows poor performance relatively to the remaining RNGs, in most of the test cases, closely followed by CLG. Thus LFG and Twister position themselves at positions two and three, with LFG producing slightly better results.

Sums of errors

The sum of errors for the considered RNGs on corresponding distributions is provided in Figure B.4. The quality of every RNG on every distribution is summarized in Table B.6.

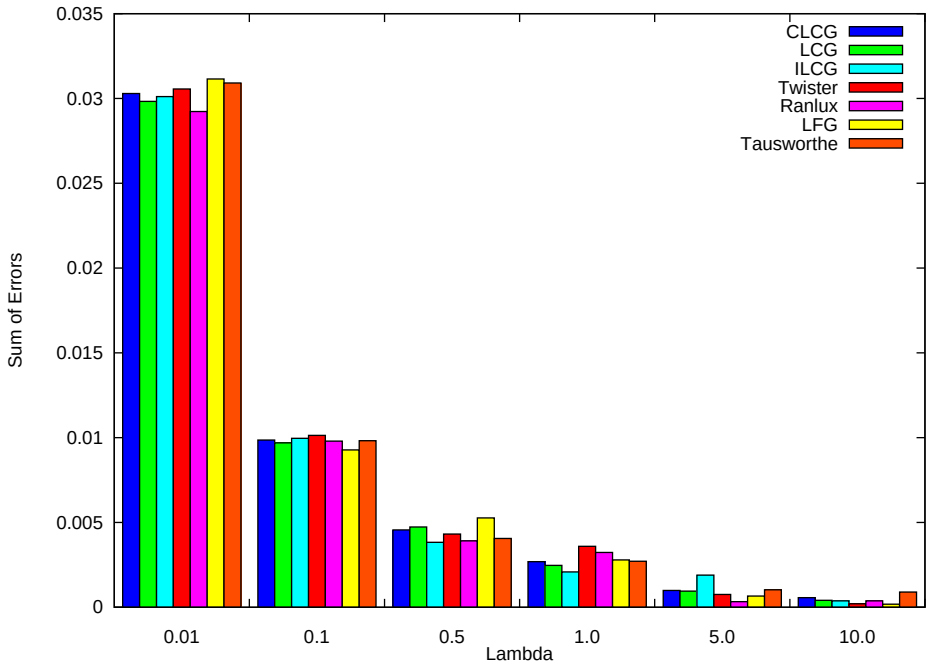


Figure B.4: Sums of errors: Exponentially distributed random variables

Pos.	λ					
	0.01	**0.1**	**0.5**	**1.0**	**5.0**	**10.0**
1	Ranlux	LFG	ILCG	ILCG	Ranlux	LFG
2	LCG	LCG	Ranlux	LCG	LFG	Twister
3	ILCG	Ranlux	Tausworthe	CLCG	Twister	ILCG
4	CLCG	Tausworthe	Twister	Tausworthe	LCG	Ranlux
5	Twister	CLCG	CLCG	LFG	CLCG	LCG
6	Tausworthe	ILCG	LCG	Ranlux	Tausworthe	CLCG
7	LFG	Twister	LFG	Twister	ILCG	Tausworthe

Table B.6: Sums of errors: Exponentially distributed random variables

The results of Figure B.4 and Table B.6 are hard to summarize by giving an exact ordering on the considered RNGs. Tausworthe Generator, as well as CLCG, show (at least for the middle- to low-ranked positions) some kind of stability on places four to six for Tausworthe Generator and places three to six for CLCG respectively. Concerning the leading positions, Ranlux is the only RNG showing durable behavior on position one to three in four out of the six test cases. The remaining RNGs – namely LFG, LCG, ILCG and Twister – permanently change positions with being in first place for one test case, but already in last place for

another. As no clear tendency could be observed here, we obtain a large field of middle-ranked RNGs for the simulation quality tests.

C CTMC Steady State Simulation

This appendix describes two heuristics applied to the steady-state simulation algorithms implemented in MRMC, see Section 8 and also Part II of [Zap08]:

1. Speeding up the regeneration method on larger models.
2. Optimising the frequency of computing confidence intervals on smaller models.

The positive effect of using both of these heuristics, in case of model checking steady-state properties on CTMCs, has solid experimental evidences, see e. g. [KZ09].

C.1 Heuristic Regeneration Point

Our experiments revealed that, in case of large Markov chains, steady-state simulations can take very long time. The main problem lies within the regeneration method [CL77] used for data collection and analysis: *On large ergodic models ($\geq$ 1.000 states) regeneration cycles typically need a lot of time to be complete*. The problem can be relaxed by finding a regeneration point that allows for shorten regeneration cycles.

In the present state of the art, finding an optimal regeneration point is an unsolved research problem [CL77]. Thus, we have chosen several available heuristics and performed an experimental evaluation of their performance using various case studies.

	Heuristic	Description
1.	pure regeneration method	the state with the lowest index in every BSCC
2.	highest incoming rate	the state with highest incoming rate
3.	lowest rate difference	the state with the lowest difference between incoming and outgoing rate
4.	**sample-based approach**	the most visited state after foregoing sampling
5.	dynamic approach	dynamic regeneration state, i.e. choose a new regeneration state (randomly/sorted by rate etc.) after every completed cycle

Table C.1: Regeneration state choice heuristics

According to our results, see Figure C.1, the **sample-based approach** provides the best results regarding the cycle length and the overall performance. On the figure, the plot corresponding to this technique is named: *static, sample-based*.

MRMC implements the **sample-based approach** by selecting the regeneration point for each BSCC B_i to be the most recurring state in a test simulation run of length $3 \times |B_i|$. At present, MRMC has this heuristic enabled by default.

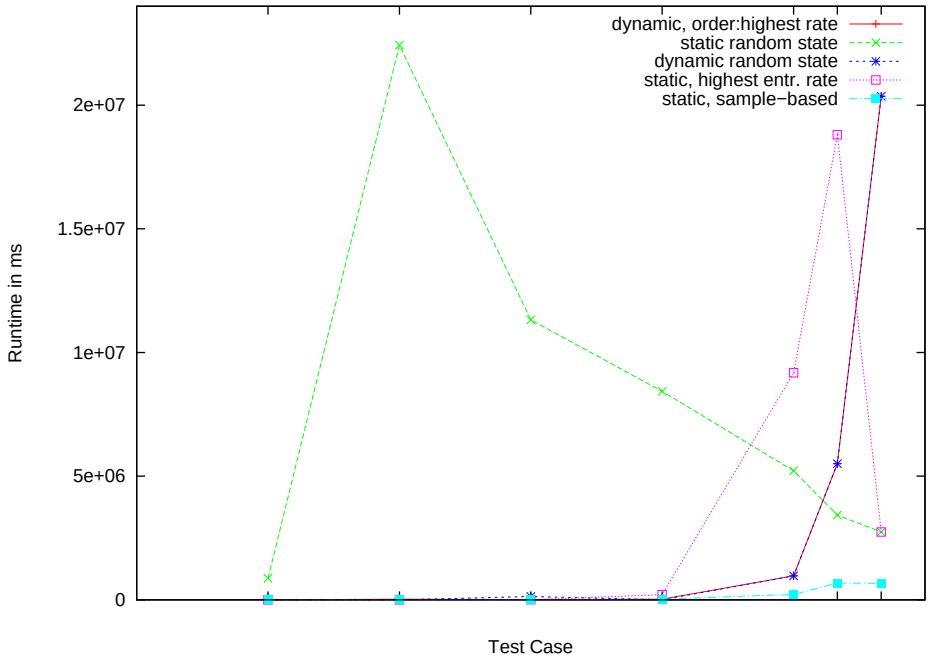


Figure C.1: Runtime: dynamic and static regeneration points

C.2 Heuristic Sample-size Steps

On small ergodic models, ≤ 1.000 states, regeneration cycles are pretty small. This leads to a very frequent re-computation of the confidence intervals that becomes a computational bottleneck and leads to long model-checking times.

The effect can be discounted if in regeneration simulations we use a "dynamic" sample-size increase. We (successfully) used the following formulae for the minimal sample size used in simulations:

$$N^s_{min} := \sqrt{\frac{T}{B}}$$

and the sample-size step:

$$\Delta N^s := T/\left(\frac{L}{B \times N^s}\right) + N^s$$

where N^s – the current sample size during the regeneration simulation; N^s_{min} – the minimal sample size to consider; ΔN^s – the delta to increase the sample size before recomputing the confidence interval; L – #states visited during regeneration simulation; B - #(simulated BSCCs), i. e. reachable non-trivial BSCCs with $\mathcal{G}$ states; T - #states in simulated BSCCs.

D Partition refinement and sparse matrices in MRMC 1.5

by David N. Jansen, Model-based system development, Radboud Universiteit, Nijmegen, The Netherlands

Partition refinement is the principle behind the algorithm used in MRMC for lumping, both formula-independent and formula-dependent.

In a partition refinement algorithm, one starts with a coarse partition of the state space. Each block in the partition is one equivalence class in the bisimulation relation—at least, that is the intention. In most cases, however, partition blocks are initially larger than equivalence classes and need to be split according to some rules until a fixpoint is reached. In this fixpoint, then, the blocks are exactly the equivalence classes.

The rule to decide which blocks need to be split for bisimilarity is: Maintain a set of potential *splitters*. A splitter Sp is a block in the current partition[1] that makes another block C split because $\mathbf{P}(\cdot, Sp)$ is not the same for each state in C. If such a block C has been found, one has to refine it into smaller subblocks $C_1, C_2, \ldots, C_k$, such that $\mathbf{P}(\cdot, Sp)$ is constant on every single C_i.

If C was a potential splitter itself, the $C_1, \ldots, C_k$ replace C as potential splitters. Otherwise, the $C_1, \ldots, C_k$ become potential splitters, except that one C_i is redundant; of course it is advantageous to pick a maximal C_i as the redundant one.

We analysed the lumping algorithm presented in [DHS03] and came to the following conclusions:

- The data structure to store a partition, the set of potential splitters and the set of predecessor blocks (sets of sets of states) are often constructed from several linked lists. However, as every state is in exactly one set of the partition and the set of splitters and the set of predecessor blocks are subsets of the partition, one can replace the data structure, as suggested in [Knu01, VL08], by an array containing all states in a peculiar order.

[1] Actually, [DHS03] maintain a set of potential splitters that may also contain elements of earlier versions of the partition: Contrary to what is written below, [DHS03] do not replace a block C in the set of potential splitters by its subblocks $C_1, \ldots, C_k$ if it is split by some other potential splitter. Instead, they always regard one of the C_i as redundant. (Unfortunately, this is written down in a manner that is rather unclear: the fine distinction, not visible from their article alone, is that the set L contains blocks, but the set L'' contains *references* to blocks.)

While [DHS03] can achieve the same time bound $\mathcal{O}(m \log n)$ as the variant described in this text, it seems less efficient to us, as some splitters are larger than required, which incurs more calculations. Additionally, some states may be a member of more than one splitter at the same time, which would disallow our data structure.

- Using splay trees is not required for the optimal time bound of $\mathcal{O}(m \log n)$ (where $m =$ number of transitions and $n =$ number of states). What is required, is a sorting algorithm that accounts for equal keys.

  We therefore replaced splay sort by a variant of quicksort for equal keys [BM93].

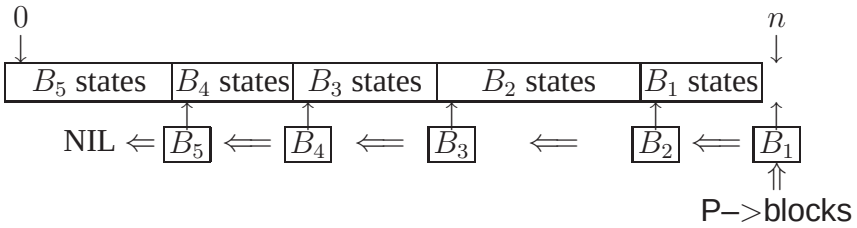
- Finding the incoming probabilities of a state is inefficient in MRMC 1.4.1, because the sparse matrix data structure does not provide that information readily. (It does only provide the list of predecessors of a state; to find the incoming transition probability, a [binary] search through the successors of every predecessor is required.)

  We have improved the data structure so that the incoming probabilities can be found more easily. This was possible without using extra memory. (However, as this improvement led to degraded performance in other parts of MRMC, we decided not to release it.)

We are going to give more details about each of these points below.

D.1 Partition data structure

It is known that every state is in exactly one block of the partition. We therefore place the state indices in an array P–>ib[·].id of size n such that states belonging to the same block are adjacent. The linked list of partition blocks then contains the index in the array where a block starts and ends (as usual in C, the end pointer actually points one past the subarray). If one keeps the block list in the reverse order of their position in the array, it is enough to maintain the end index: the start index can easily be found by looking at the end of the next block in the list. (for the last block in the list, the start is obviously 0; this is the reason why we use the *reverse* order.) A partition containing five blocks could be drawn as follows:



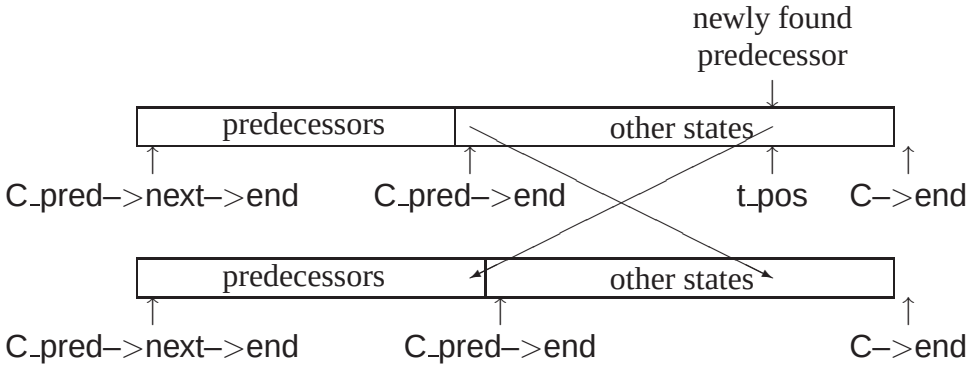
In this diagram, double arrows indicate pointers (for the linked list) and single arrows indicate array indices. P–>blocks is a field of the partition structure. In a separate array P–>ib[·].b, we store for every state a pointer to its block.

The list of potential splitters is maintained as a sublist of the block list: every block has a second pointer B–>u.next_Sp; if the block is a splitter, B–>u.next_Sp points to the next potential splitter. (This list may be unordered.) In addition, each block has a flag B–>flag to indicate whether it is really in the list; sometimes, we can see afterwards that a certain block is actually not a potential splitter.[2]

[2] This may perhaps be improved upon, even without using a double-linked list; a potential splitter is only deleted from the list if it is chosen as the current splitter, or if, after adding some new blocks to the partition, one inserts all but the largest one into the list of splitters.

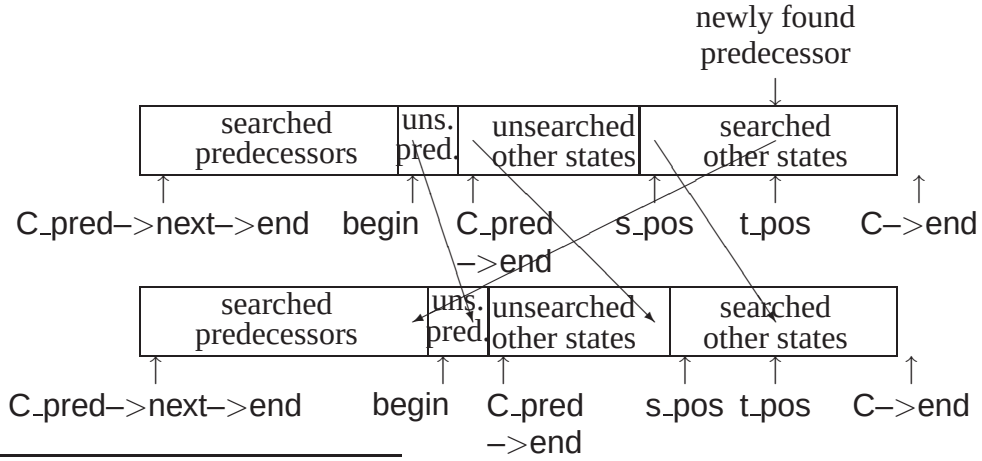
The list of predecessor blocks is, in principle, another sublist of the block list, requiring another pointer B–>u.next_PredCl. However, with a trick we can achieve that the list of splitters and the list of predecessor blocks become disjoint, so we can use a C union for the two sublist pointers. The trick is: If a predecessor block is found, it is split immediately into two subblocks: a new subblock C_pred containing the found predecessors of the splitter and the remaining block C containing the other states in the block. (Note that it is easy to insert a new block near the *beginning* of a given block: one just assigns C_pred–>next = C–>next, C–>next = C_pred. As the block would have to be split anyway into these two parts, this does not incur additional overhead.) The new subblock becomes element of the list of predecessor blocks and the old block stays in the list of potential splitters (if it was there).

In function find_predecessors of file src/lumping/lump.c, states that are found to be predecessors swap their position with other states to make every state go into the correct subblock. In the situation depicted below, the state at t_pos is swapped with the one at C_pred–>end, which is increased afterwards to reflect the new situation.



The function find_predecessors walks through the splitter from the last to the first state and looks for predecessors per state. If the splitter splits itself, one has to take extra care that no state in the splitter is forgotten or considered twice.

The splitter actually consists of four parts: predecessor states whose predecessors have or have not been searched, and other states whose predecessors have or have not been searched. In the extreme case, up to four elements must be moved:



The first case is easy: one always chooses the first list element, which is easy to delete from the list. In the second case, careful bookkeeping should allow to find the predecessor of the largest block and adapt its u.next_Sp pointer instead of clearing the flag in the largest block. However, one still needs a $\mathcal{O}(1)$ test whether a block is a splitter or not.

To make sure that this works, the splitter has to be searched from the end to the beginning, i. e. from s_pos to begin.

After having found the predecessors, every predecessor block has to be split such that $\mathbf{P}(\cdot, Sp)$ is constant for every subblock. The states remaining in the old subblock are not predecessors and obviously have $\mathbf{P}(\cdot, Sp) = 0$, so only the new blocks in the list of predecessor blocks need to be considered further.

D.2 Optimal sorting

To be able to refine a block C one actually has to sort its states according to $\mathbf{P}(\cdot, Sp)$, so that one can construct the C_i consisting of elements with the same value $\mathbf{P}(\cdot, Sp)$.

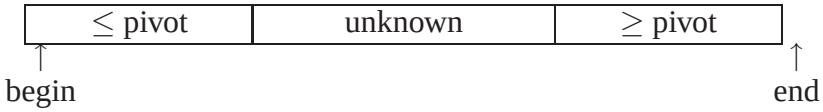
D.2.1 Optimal time bound

A simple sort algorithm requires time in $\mathcal{O}(|C| \log |C|)$. Later, it may happen that some of the resulting subblocks C_i are split again by some other splitter, which requires another time in $\mathcal{O}(|C_i| \log |C_i|)$. In the end, using a simple sort algorithm requires time $\mathcal{O}(m \log^2 n)$.

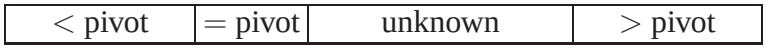
To achieve a better time bound, one may use a sort algorithm that takes into account the equal keys. An optimal multiset sort algorithm uses at most $|C| \log |C| - \sum_{i=1}^{k} |C_i| \log |C_i| + \mathcal{O}(|C|)$ three-branch-comparisons [MS76]. Now, if we add the number of comparisons needed to split (some of) the C_i, we get as the total worst-case required number of comparisons to split C and all its subblocks $\mathcal{O}(|C| \log |C|)$. Therefore, in the end, splitting the set of states requires at most $\mathcal{O}(n \log n)$ comparisons. Together with the other operations in partition refinement, we get an upper bound of $\mathcal{O}(m \log n)$ if we use a sort algorithm that is optimal for sets with equal keys.

D.2.2 Adapting quicksort for equal keys to splitting

In quicksort, a block to be sorted is split into two parts: one part containing "small" elements and another part containing "large" ones. After that, the two parts are sorted recursively. The split is achieved by picking a pivot element and regarding the elements $<$ pivot as small and those $>$ pivot as large. The elements $=$ pivot may become members of either part. During the split operation, the block looks something like this:

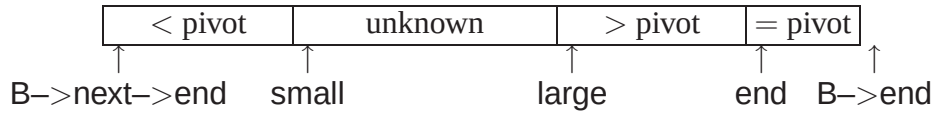


However, if there are many elements equal to the pivot, it is advantageous to split the block into three parts: the small elements, the large elements and those $=$ pivot. The last part does not need to be sorted further. Where should one place the third part during the split operation? [BM93] say that the "Dutch national flag" scheme proposed by Dijkstra



leads to complex and slow code. They propose to use a scheme:

and to swap the $=$ pivot elements to the middle at the end of the splitting step. We decided to use an asymmetric variant of this scheme, because in our case, we do not require that the $=$ pivot elements end up in the middle; it is only required that they be together. So our scheme becomes:



The function pass_file_noerror in the file src/lumping/sort.c generates this partition. The above scheme also contains the variables used in this function to save the array indices where the block is split (B–>next–>end and B–>end are fields of the block structure; the others are local variables). At the end of pass_file_noerror, we have small $=$ large and we proceed by creating three subblocks in the partition (in the function sort_and_split_block_internal, also in src/lumping/sort.c). The first two subblocks are then sorted recursively.

D.3 Sparse matrix data structure

The new data structure for sparse matrices is described in [KZH+09, KZH+10]. It is an adapted version of the compressed-row data format for sparse matrices. In this format, there is an array of *row pointers* that indicate where the data for each row starts; each row is organised as a vector of pairs (column, value), with the understanding that columns containing 0 are suppressed.

This data structure allows to find all successors of a state (i. e., all nonzero elements in a row of the transition probability matrix) easily. To find the predecessors of a state (i. e., all nonzero elements in a column), one has to check every row vector whether it contains the desired column.

To make this task more efficient, we proposed to add a *backpointer list* to the data structure. This is, for each column, a vector of pointers to the appropriate pairs (column, value).[3] Further details on this data structure can be found in the articles mentioned above.

Not released. Recent tests have shown that while this improved sparse matrix data structure speeds up bisimulation minimisation somewhat, it leads to serious degradation in simulation. Therefore, we decided not to release the sparse matrix data structure in the main version of MRMC. If desired, it is available as a patch from the MRMC download page.

[3] This backpointer list has to be distinguished from the backlist of earlier versions of MRMC, which only contained the row numbers. That would help in finding the predecessors, but not their transition probabilities; the latter have still to be looked up by searching through one row vector.