Brno University of Technology
Faculty of Information Technology

Static Analysis and Verification

Project — PRISM

21st December 2020

Dominik Harmim (`xharmi00`)

Contents

1 Introduction

This project aims to study a selected software tool for *static analysis and verification*, do some experiments with the tool, and describe it appropriately. It was selected **PRISM** — a *probabilistic model checker*.

The rest of the essay is organised as follows. Chapter 2 describes PRISM and its principles. Chapter 3 then deals with a reproduction of publicly available case studies. In Chapter 4, there are discussed made custom experiments. Finally, Chapter 5 concludes the essay.

2 PRISM — Probabilistic Model Checker

This Chapter describes PRISM — a *probabilistic model checker* — and its principles, usage, used algorithms, *modelling language*, implementation, theory it is based on, etc. The description of *model checking* in general together with the definition of fundamentals of *temporal logic* and basic concepts in model checking can be found in books [1, 2]. This Chapter is based on papers [5, 6] where are discussed principles and algorithms that is PRISM based on, implementation of these algorithms in PRISM, and related theory. Some of the mentioned algorithms are in more detail covered in [3]. The most recent features of PRISM (mainly *probabilistic timed automata*) are introduced in the PRISM tool paper [7]. Some information was also acquired from a web page of PRISM[1]. Further details about *probabilistic systems* and perhaps more formal and theoretic insight is available at Chapter 10 of [1] and Chapter 28 of [2]. Moreover, it was used information and other materials from lecture slides [4, 8, 9, 10].

PRISM[1] is a *probabilistic model checker* developed mainly at the University of Birmingham by David Parker, the University of Glasgow by Gethin Norman, and the University of Oxford by Marta Kwiatkowska. It is *free* and *open-source* (released under the GNU General Public License). It is available for Linux, Unix, macOS, and Windows. It accepts *probabilistic models* described in its *modelling language* — a simple, high-level state-based language further described in Chapter 2.4. Three types of probabilistic models (see Chapter 2.1) are supported directly; these are *discrete-time Markov chains (DTMCs)*, *continuous-time Markov chains (CTMCs)*, and *Markov decision processes (MDPs)*, i.e., *probabilistic automata (PAs)*. Additionally, *probabilistic timed automata (PTAs)* are partially supported, with the subset of diagonal-free PTAs supported directly via *digital clocks*. Note that since version 4.0 of PRISM [7], PTAs are fully supported. Also, *priced* PTAs are supported, i.e., PTAs augmented with *costs* and *rewards*. PTAs are finite-state automata enriched with real-valued clocks, in the style of *timed automata* (defined in [1]), and with discrete probabilistic choice, in the style of MDPs. So, PTAs can be viewed as DTMCs with *continuous-time* and *non-determinism*.

Properties are formally specified using *probabilistic computation tree logic (PCTL)* for DTMCs and MDPs, and *continuous stochastic logic (CSL)* for CTMCs. These are discussed in Chapter 2.2. PTAs have *probabilistic timed computation tree logic (PTCTL)*, an extension of *timed computation tree logic (TCTL)* defined in [1].

PRISM first parses the model description and constructs an internal representation of the probabilistic model, computing the *reachable state-space* of the model and discarding any unreachable states. The resulting model represents the set of all *feasible configurations* which can arise in the modelled system. This process is depicted in Figure 2. Next, the specification is parsed, and appropriate model checking algorithms are performed on the model by induction over syntax. The whole process is outlined in Figure 1. In some cases, such as for properties which include a *probability bound*, PRISM will report a `True/False` outcome, indicating whether or not each property is satisfied by the current model. More

[1] A web page of a *probabilistic model checker* **PRISM**: `http://www.prismmodelchecker.org`.

often, however, properties return *quantitative* results and PRISM reports, e.g., the actual probability of a specific event occurring in the model. Furthermore, PRISM supports the notion of *experiments*, which is a way of automating multiple instances of model checking. These allow one to quickly obtain the outcome of one or more properties as functions of model and property parameters. The resulting table of values can either be viewed directly, exported for use in an external application, or plotted as a graph. Moreover, PRISM incorporates substantial graph-plotting functionality. It is often a handy way of identifying interesting patterns or trends in the behaviour of a system.

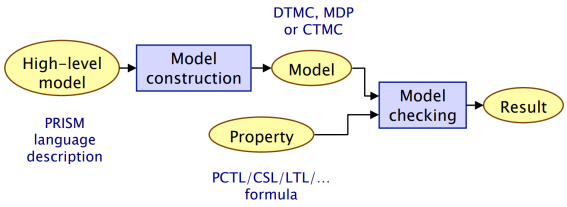


Figure 1: Overview of the workflow of PRISM [9]

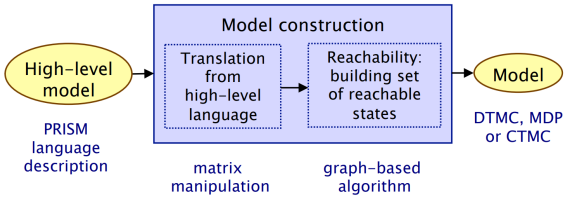


Figure 2: Overview of the model construction process in PRISM [9]

Figure 3 shows a screenshot of the PRISM graphical user interface, illustrating the results of a model checking experiment being plotted on a graph. The tool also features a built-in text-editor for the PRISM language and a *simulator* for model debugging. Alternatively, all model checking functionality is also available in a command-line version of the tool.

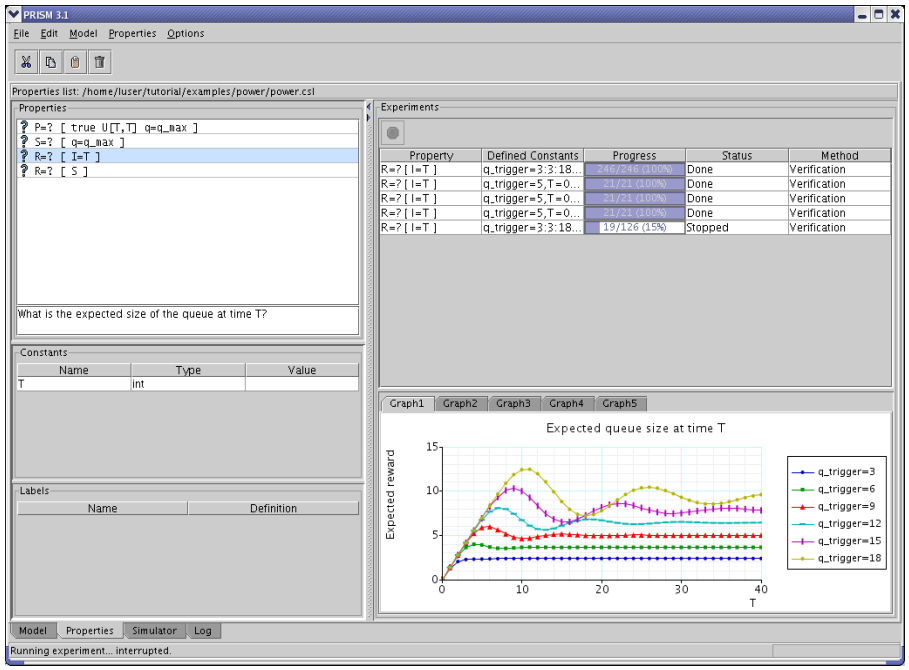


Figure 3: A screenshot of the PRISM graphical user interface (http://www.prismmodelchecker.org/screenshots.php)

2.1 Probabilistic Models

This Chapter briefly describes and formally defines three major types of *probabilistic models* supported by PRISM. For particular algorithms that reason about the probability of certain events occurring in such models see [3, 5]. In more detail, the models are covered in [1, 2, 3, 5].

2.1.1 Discrete-Time Markov Chains (DTMCs)

Discrete-time Markov chains (DTMCs) model systems whose behaviour at each point in time can be described by a *discrete probabilistic choice* over several possible outcomes. Essentially, a DTMC can be thought of as a *labelled state-transition system* in which each transition is annotated with a probability value indicating a likelihood of its occurrence. That is to say, the successor state of state s, say, is chosen according to a *probability distribution*. This probability distribution only depends on the current state s, and not, e.g., the path fragment that led to state s from some initial state. Accordingly, the system evolution does not depend on the history (i.e., the path fragment that has been executed so far), but only on the current state s. This is known as the *memory-less property*. Intuitively, DTMC is defined by (i) states — *set of states* representing possible configurations of the system being modelled, (ii) transitions — transitions between states model evolution of system's state; occur in *discrete time-steps*, (iii) and probabilities — probabilities of making transitions between states are given by discrete probability distributions.

Definition 2.1 (DTMC)**.** Formally, a (labelled) DTMC $\mathcal{D}$ is a tuple $(S, s_{init}, \boldsymbol{P}, AP, L)$ where:

- S is a (countable non-empty) set of *states* (*state-space*);
- $s_{init} \in S$ is the *initial state* (can be generalised to an *initial distribution* $\iota_{init} : S \to [0,1]$, such that $\sum_{s \in S} \iota_{init}(s) = 1$);
- $\boldsymbol{P} : S \times S \to [0,1]$ is the *transition probability matrix* such that $\sum_{s' \in S} \boldsymbol{P}(s,s') = 1$ for all $s \in S$;
- AP is a set of *atomic prepositions*;
- $L : S \to 2^{AP}$ is a *labelling function* that assigns, to each state $s \in S$, a set $L(s)$ of *atomic prepositions*.

For a state $s \in S$ of a DTMC $\mathcal{D}$, the probability of moving to a state $s' \in S$ in one discrete step is given by $\boldsymbol{P}(s,s')$. An (infinite) *path* of $\mathcal{D}$, which gives one possible evolution of the Markov chain, is a non-empty alternating (infinite) sequence of states $s_0 s_1 s_2 \ldots \in S^\omega$ such that $\boldsymbol{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$.

Alternatively, a DTMC can be defined as a *family of random variables* $\{X(k) \mid k = 0, 1, 2, \ldots\}$ where $X(k)$ are observations at discrete time-steps, i.e., $X(k)$ is the state of the system at time-step k. This satisfies the *Markov property* (memory-less property): $Pr(X(k) = s_k \mid X(k-1) = s_{k-1}, \ldots, X(0) = s_0) = Pr(X(k) = s_k \mid X(k-1) = s_{k-1})$, i.e., for a given current state, future states are independent of past.

2.1.2 Continuous-Time Markov Chains (CTMCs)

DTMCs are *discrete-time* models: the progress of time is modelled by discrete time steps, one for each transition of the model. For many systems, it is preferable to use a *continuous* model of time, where the delays between transitions can be arbitrary real values. One popular model is a **continuous-time Markov chain (CTMC)**, in which transition delays are assumed to be modelled by *exponential distributions*.

Definition 2.2 (CTMC)**.** Formally, a (labelled) CTMC $\mathcal{C}$ is a tuple $(S, s_{init}, \boldsymbol{R}, AP, L)$ where:

- S, $s_{init} \in S$, AP, and $L : S \to 2^{AP}$ are as for DTMCs;
- $\boldsymbol{R} : S \times S \to \mathbb{R}_{\geq 0}$ is the *transition rate matrix*.

The matrix $\boldsymbol{R}$ assigns a *rate* to each pair of states in the CTMC. A transition can only occur between states s and s' if $\boldsymbol{R}(s,s') > 0$ and, in this case, the delay before the transition can occur is modelled as an exponential distribution with rate $\boldsymbol{R}(s,s')$, i.e., the probability of this transition being triggered within t time-units is $1 - e^{-\boldsymbol{R}(s,s')\cdot t}$. Usually, in a state s, there is more than one state s' for which $\boldsymbol{R}(s,s') > 0$, which is referred to as a *race condition*. The first transition to be triggered determines the next state of the CTMC. The time spent in state s, before any such transition occurs, is exponentially distributed with rate $E(s)$, where: $E(s) = \sum_{s' \in S} \boldsymbol{R}(s,s')$. $E(s)$ is known as the *exit rate* of state s. It can also be determined the actual probability of each state s' being the next state to which a transition is made from state s, independent of the time at which this occurs. This is defined by the *embedded* DTMC of $\mathcal{C}$, which is $emb(\mathcal{C}) = (S, s_{init}, \boldsymbol{P}^{emb(\mathcal{C})}, AP, L)$, where for $s, s' \in S$:

$$\boldsymbol{P}^{emb(\mathcal{C})}(s,s') = \begin{cases} \frac{\boldsymbol{R}(s,s')}{E(s)} & \text{if } E(s) \neq 0 \\ 1 & \text{if } E(s) = 0 \wedge s = s' \\ 0 & \text{otherwise.} \end{cases}$$

The behaviour of the CTMC can be considered in an alternative way using the above definitions. It will remain in a state s for a delay which is exponentially distributed with rate $E(s)$ and then make a transition. The probability that this transition is to state s' is given by $\boldsymbol{P}^{emb(\mathcal{C})}(s,s')$. An (infinite) path of $\mathcal{C}$ is a non-empty (infinite) sequence $s_0 t_0 s_1 t_1 s_2 t_2 \ldots \in (S \times \mathbb{R}_{\geq 0})^\omega$ where $\boldsymbol{R}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. The value t_i represents the amount of *time spent* in the state s_i.

As for DTMCs, a CTMC can be alternatively defined as a family of random variables $\{X(t) \mid t \in \mathbb{R}_{\geq 0}\}$ where $X(t)$ are observations made at time instant t, i.e., $X(t)$ is the state of the system at time instant t. Furthermore, again, it satisfies the Markov property.

2.1.3 Markov Decision Processes (MDPs)

DTMCs represent a *fully probabilistic* model of a system, i.e., in each state of the model, the exact probability of moving to each other state is always known. In many instances, this does not realistically model a system's behaviour, because it behaves in a *non-deterministic* fashion. **Markov decision processes (MDPs)** (similar to *probabilistic automata*) are one of the most common formalisms for modelling systems with both probabilistic and non-deterministic behaviour.

Definition 2.3 (MDP)**.** Formally, a (labelled) MDP $\mathcal{M}$ is a tuple $(S, s_{init}, Act, \boldsymbol{P}, AP, L)$ where:

- S, $s_{init} \in S$, AP, and $L : S \to 2^{AP}$ are as for DTMCs;
- Act is a (finite non-empty) set of *action labels*;
- $\boldsymbol{P} : S \times Act \times S \to [0,1]$ is the *transition probability matrix* where for all $s \in S$ and $\alpha \in Act$ holds $\sum_{s' \in S} \boldsymbol{P}(s,\alpha,s') = 1$ if α is *enabled* in s ($\alpha \in Act(s)$), 0 otherwise; and $Act(s) \neq \emptyset$ (to prevent *deadlocks*). ($Act(s)$ denotes the set of enabled actions in s, i.e., $Act(s) = \{\alpha \in Act \mid \exists\, s' \in S : \boldsymbol{P}(s,\alpha,s') > 0.\}$)

In each state s of an MDP, the successor state is decided in two steps: first, non-deterministically selecting an enabled action $\alpha \in Act$ (i.e., one of $Act(s)$); and, second, randomly choosing the successor according to the probability distribution $\boldsymbol{P}(s,\alpha,\cdot)$. That is, with probability $\boldsymbol{P}(s,\alpha,s')$ the next state is s'. An (infinite) path of $\mathcal{M}$ is a non-empty alternating (infinite) sequence $s_0 \alpha_0 s_1 \alpha_1 s_2 \alpha_2 \ldots \in (S, Act)^\omega$ where $\boldsymbol{P}(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \geq 0$.

To reason formally about the behaviour of MDPs, the notion of *adversaries* (also known as *schedulers*, *policies*, or *strategies*) is used, which resolve all the non-deterministic choices in a model — it induces

a DTMC. In the case, for instance, where non-determinism is used to model concurrency between components, an adversary represents one possible *scheduling* of the components over the lifetime of the system. Under a particular adversary, the behaviour of an MDP is fully probabilistic and, as for DTMCs, it can be reasoned about the best-case or worst-case system behaviour by quantifying over all possible adversaries. For example, it can be computed the minimum or maximum probability that some event occurs. Formally, an adversary is a function mapping every finite path $s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \dots s_n$ to an action taken in s_n. Further information about adversaries can be found in [1, 3].

2.2 Formal Specification of Properties

Classically, analysis of DTMCs often focuses on *transient* or *steady-state* behaviour, i.e., the probability of being in each state of the chain at a particular instant in time or the *long-run*, respectively. Probabilistic model checking adds to this the ability to reason about *path-based properties*, which can be used to specify constraints on the probability that certain desired behaviours are observed. Formally, this is done by defining a *probability space* over the set of all paths through the model. Properties are then expressed using *temporal logic*. Properties of PRISM models are expressed in *probabilistic computation tree logic (PCTL)* — a probabilistic extension of the logic *CTL* [1, 2] — for DTMCs and MDPs, and *continuous stochastic logic (CSL)* for CTMCs.

Below, logic PCTL and CSL are briefly defined together with *costs* and *rewards* extensions. In great detail, these are defined and discussed in [1, 2, 3, 5].

2.2.1 PCTL Logic for DTMCs

Definition 2.4 (PCTL syntax)**.** The syntax of PCTL is given by:

$$\begin{array}{ll} \Phi ::= \mathit{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathrm{P}_{\sim p}[\Psi] & (\textit{state formulae}) \\ \Psi ::= \mathrm{X}\,\Phi \mid \Phi\,\mathrm{U}^{\leq k}\,\Phi \mid \Phi\,\mathrm{U}\,\Phi & (\textit{path formulae}) \end{array}$$

where a is an *atomic preposition*, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$, and $k \in \mathbb{N}$.

PCTL formulae are interpreted over the states of a DTMC. A state $s \in S$ *satisfies* a PCTL formula Φ, denoted as $s \models \Phi$, if it is true for s. The key operator in PCTL is $\mathrm{P}_{\sim p}[\Psi]$ which means that the probability of a *path formula* Ψ being true in a state satisfies the bound $\sim p$. As path formulae (specified above as Ψ), it is allowed $\mathrm{X}\,\Phi$ ("Φ is satisfied in the *next* step"), $\Phi_1\,\mathrm{U}^{\leq k}\,\Phi_2$ ("Φ_2 is satisfied within k steps and Φ_1 is true *until* that point"), and $\Phi_1\,\mathrm{U}\,\Phi_2$ ("Φ_2 is *eventually* satisfied and Φ_1 is true *until* then"). In practice, it is common to take a more *quantitative* approach; instead, it can be written formulae of the following kind: $\mathrm{P}_{=?}[\neg \mathit{fail}_A\,\mathrm{U}\,\mathit{fail}_B]$, which asks "*what* is the probability that component B fails before component A?". Several other useful operators can be derived from the PCTL syntax given above. That includes, for example, $\mathrm{F}\,\Phi \equiv \mathit{true}\,\mathrm{U}\,\Phi$ ("*eventually* Φ becomes true"), $\mathrm{G}\,\Phi \equiv \neg\mathrm{F}\,\neg\Phi$ ("Φ is *always* true"), and *time-bounded* variants of these.

Model checking of a PCTL formula over a DTMC requires a combination of *graph-based* algorithms and *numerical solution* techniques. For the latter, the most common problems (e.g., computing the probabilities that a U, F, or G path formula is satisfied) reduce to solving a *system of linear equations*. For scalability reasons, in implementations of probabilistic model checking this is usually done with *iterative numerical* methods such as *Gauss-Seidel*, rather than more classical *direct* methods such as *Gaussian elimination*. A wide range of properties can be expressed by using more expressive logic such as LTL or PCTL* [1]. However, model checking becomes slightly more expensive.

2.2.2 CSL Logic for CTMCs

As for DTMCs, a probability space over the paths through a CTMC can be defined, and it can be reasoned about the probability of certain events occurring. To specify such properties, the logic CSL has been proposed, which extends PCTL with continuous versions of the time-bounded operators, i.e., $\Phi\, \mathrm{U}^{\leq t}\, \Phi$ (where $t \in \mathbb{R}_{\geq 0}$), and a *steady-state operator* $\mathrm{S}_{\sim p}[\Psi]$ (in the "long-run" Φ is true with probability $\sim p$).

Model checking for CTMCs is similar to DTMCs, but also uses techniques from performance evaluation, in particular *uniformisation*, an efficient iterative numerical method for computing transient probabilities of CTMCs.

2.2.3 PCTL Logic for MDPs

To formally specify properties in MDPs, it is again used the logic PCTL, with identical syntax to the DTMC case, but with an implicit quantification over *adversaries* (i.e., the semantics of probabilistic operator $s \models \mathrm{P}_{\sim p}[\Psi]$ is that the probability of the set of paths starting in s and satisfying Ψ meets $\sim p$ *for all schedulers*). The $\mathrm{P}_{=?}$ operator used for DTMCs is replaced with two variants $\mathrm{P}_{\mathrm{min}=?}$ and $\mathrm{P}_{\mathrm{max}=?}$ (*minimum* and *maximum* probabilities over all schedulers, that corresponds to an analysis of *best-case* or *worst-case* behaviour of the system).

Model checking MDPs requires the solution of *linear optimisation problems*, rather than linear equation systems. In practice, this is often done using *dynamic programming*.

2.2.4 Costs and Rewards

DTMCs (and other probabilistic models) can be augmented with *reward* (or *cost*) information, which allows reasoning about a wide range of additional quantitative measures. Formally, a *reward* structure $(\underline{\rho}, \iota)$ is defined, for a DTMC $\mathcal{D} = (S, s_{init}, \boldsymbol{P}, AP, L)$, that allows specification of two distinct types of rewards: *state rewards*, which are assigned to states using the reward function $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$, and *transition rewards*, which are assigned to transitions by means of the reward function $\iota : S \times S \to \mathbb{R}_{\geq 0}$. The state reward $\underline{\rho}(s)$ is the reward acquired in state s per time step, i.e., a reward of $\underline{\rho}(s)$ is incurred if the DTMC is in state s for one time step, and transition reward $\iota(s, s')$ is acquired each time a transition between states s and s' occurs.

To express *reward-based properties* of DTMCs, the logic PCTL can be extended with additional operators:

$$\mathrm{R}_{\sim r}[\mathrm{C}^{\leq k}] \mid \mathrm{R}_{\sim r}[\mathrm{I}^{=k}] \mid \mathrm{R}_{\sim r}[\mathrm{F}\, \Phi] \mid \mathrm{R}_{\sim r}[\mathrm{S}]$$

where $\sim \in \{<, \leq, \geq, >\}$, $r \in \mathbb{R}_{\geq 0}$, $k \in \mathbb{N}$, and Φ is a PCTL formula. The R operators specify properties about the *expected* value of rewards. The formula $\mathrm{R}_{\sim r}[\Psi]$, where Ψ denotes one of the four possible operators given in the grammar above, is satisfied in a state s if, from s, the expected value of reward Ψ meets the bound $\sim r$. The possibilities for Ψ are: $\mathrm{C}^{\leq k}$, which refers to the reward *cumulated* over k time steps; $\mathrm{I}^{=k}$, the state reward at time *instant* k (i.e., after exactly k time steps); $\mathrm{F}\, \Phi$, the reward cumulated before a state satisfying Φ is *reached*; and S, the *long-run steady-state rate* of reward accumulation. As for P operator, properties of the form $\mathrm{R}_{=?}[\Psi]$, with the meaning "what is the expected reward?", are often used.

Like for the basic operators of PCTL, model checking for the reward operators above reduces to a combination of graph algorithms and linear equation system solution.

2.3 Implementation of PRISM

One of the most notable features of PRISM is that it is a *symbolic* model checker, meaning that its implementation uses data structures based on *binary decision diagrams (BDDs)* [2]. These provide compact representation and efficient manipulation of large, structured probabilistic models by exploiting *regularity* that is often present in those models because they are described in a structured, high-level modelling language. More specifically, since it is needed to store numerical values, PRISM uses *multi-terminal binary decision diagrams (MTBDDs)* [9], and some variants developed to improve the efficiency of probabilistic analysis, which involve combinations of symbolic data structures such as MTBDDs and conventional *explicit* storage schemes such as *sparse* matrices and arrays.

The underlying computation in PRISM involves a combination of (i) *graph-theoretical algorithms*, for *reachability analysis*, conventional temporal logic model checking and *qualitative* probabilistic model checking; (ii) and *numerical computation*, for *quantitative* probabilistic model checking, e.g., solution of *linear equation systems* (for DTMCs and CTMCs) and *linear optimisation problems* (for MDPs). Graph-theoretical algorithms are comparable to the operation of a conventional, non-probabilistic model checker and are always performed in PRISM using BDDs. For numerical computation, PRISM uses *iterative* methods rather than *direct* methods due to the size of the models that need to be handled. For a solution of linear equation systems, PRISM supports a range of well-known techniques, including the *Jacobi*, *Gauss-Seidel*, and *successive over-relaxation (SOR)* methods. For the linear optimisation problems which arise in the analysis of MDPs, PRISM uses *dynamic programming* techniques, in particular, *value iteration*. Finally, for *transient analysis* of CTMCs, PRISM incorporates another iterative numerical method, *uniformisation*. An overview of the model checking process in PRISM can be seen in Figure 4.

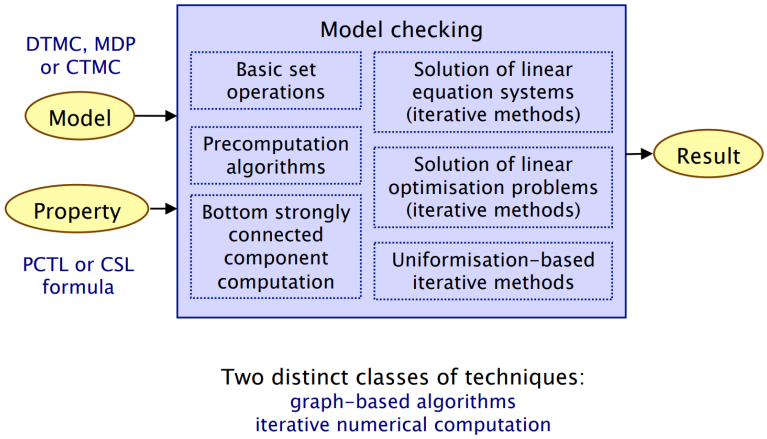


Figure 4: Overview of the probabilistic model checking process in PRISM [9]

In fact, for numerical computation, PRISM provides three distinct *numerical engines*: (i) *sparse* — uses sparse matrices; (ii) *symbolic* — implemented purely in MTBDDs (and BDDs); (iii) and *hybrid* — uses a combination of the first two. Performance (time and space) of the tool may vary depending on the choice of the engine. Typically the sparse engine is quicker than its MTBDD counterpart but requires more memory. The hybrid engine aims to provide a compromise, providing faster computation than pure MTBDDs but using less memory than sparse matrices.

Note that in the latest major release of PRISM [7], there were introduced several new features, such as support for *probabilistic timed automata (PTAs)*; *quantitative abstraction refinement* toolkit; *explicit-state* probabilistic model checking library, that can be used, e.g., for prototyping new model checking algorithms; *discrete-event simulation* engine; and others.

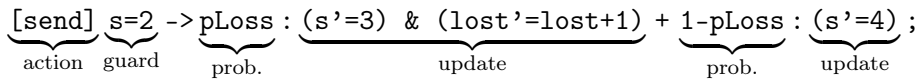
2.4 The PRISM Modelling Language

The PRISM *modelling language* is a simple, *state-based* language based on the so-called *Reactive Modules* formalism. In this Chapter, it is given an elementary outline of the language. For a full definition of the language and its semantics, see the manual for PRISM[2].

The fundamental components of the PRISM language are *modules* and *variables*. Variables are typed (integers, reals, and booleans are supported) and can be local or global. A model is composed of modules which can interact with each other. A module contains a number of local variables. The values of these variables at any given time constitute the state of the module. The *global state* of the whole model is determined by the *local state* of all modules, together with the values of the global variables. A module is defined using the `module` "*`module_name`*" ... `endmodule` construct. Modules contain *data* in the form of variables (defined by its type, range, and initial value), and *behaviour* in the form of a set of *commands*. A command takes the form:

$$[a]\ g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n\,;$$

The *guard* g is a predicate over all the variables in the model (including those belonging to other modules). Each *update* u_i describes a transition which the module can make if the guard is true. A transition is specified by giving the new values of the variables in the module, possibly as an *expression* formed from other variables or *constants*. The expressions λ_i are used to assign probabilistic information (i.e., a probability or a rate in the case of CTMCs) to the transitions. The (optional) *action* a serves for *synchronisation*. It can be used to force two or more modules to make transitions simultaneously (i.e., modules synchronise over all their common actions). The rate of these transactions is equal to the product of their individual rates. Actions can also be used to specify *reward structures* defined later. An example of a command is given below. Here, if `s=2`, the first update is performed with a probability `pLoss`, and the second update is performed with a probability `1-pLoss`. Moreover, the command is labelled with action `send`.



In general, the probabilistic model corresponding to a PRISM language description is constructed as the *parallel composition* of its modules. In every state of the model, there is a set of commands (belonging to any of the modules) which are enabled, i.e., whose guards are satisfied in that state. The choice between which command is performed (i.e., the *scheduling*) depends on the model type. For a DTMC (denoted with a keyword `dtmc`), the choice is *probabilistic*, with each enabled command selected with equal probability. For CTMCs (denoted with a keyword `ctmc`), it is modelled as a *race condition*. Finally, for an MDP (denoted with a keyword `mdp`), the choice is *non-deterministic*. Note that there is also a keyword `pta` for PTAs.

2.4.1 Reward Structures

PRISM includes support for the specification and analysis of properties based on *reward* (or *cost*) structures. Reward structures are associated with models using the `rewards` "*`reward_name`*" ... `endrewards` construct and are specified using multiple reward items of the form:

$$g : r\,; \quad \text{or} \quad [a]\ g : r\,;$$

depending on whether a state or transition rewards are being specified, where g is a predicate (over all variables of the module), a is an action label appearing in the commands of the model, and r is

[2]The **manual** for PRISM: http://www.prismmodelchecker.org/manual.

a real-valued expression (containing any variables, constants, etc. from the model). A single reward item can assign different rewards to different states or transitions, depending on the values of model variables in each one. Any states/transitions which do not satisfy the guard of a reward item will have no reward assigned to them. For states/transitions which satisfy multiple guards, the reward assigned is the sum of the rewards for all the corresponding reward items.

3 Reproduction of Available Case Studies

This Chapter describes the reproduction of available case studies. Many case studies from different fields are publicly available at a web page of PRISM[3]. For this project, it was selected a *Randomised Dining Philosophers* problem[4]. A description of the problem and its probabilistic model is covered in Chapter 4.1. Further, performed experiments are discussed in Chapter 4.2.

All the experiments (experiments in Chapter 4 as well) were carried out on the Windows distribution of PRISM version 4.5.

3.1 Description of the Problem and its Model

The case study — **Randomised Dining Philosophers** — is based on Lehmann and Rabin's *randomised* solution to the well-known dining philosophers problem. Here, it is considered the version which removes the need to consider *fairness* assumptions on the *scheduling mechanism*. The model is presented as an MDP because it contains both *probabilistic choices* as well as *non-deterministic actions*.

Let N denotes the number of philosophers. In Listing 1, there is given a source code of the considered problem in the PRISM modelling language in the case where $N = 3$. However, experiments were also performed on models where $N = 4, 5, 6$. The modelled system works as follows. There is *module* `phil1` that represents both *behaviour* and *data* of the first philosopher. The other philosophers are modelled with the same module created by *renaming* the original module and its variables. The module contains variable `p1` that represents a *state* of a given philosopher. It is an integer variable in a range $[0, 11]$. Each number represents a different state of the philosopher, e.g., 2 means that the philosopher is picking up the left fork, 8 means that the philosopher has both forks and he is moving to eat, etc. All the states are described in the Listing using comments. Essentially, the *transitions* are as follows. First, there is *probabilistic reasoning* about which fork to pick up. Then, the philosopher picks up one of the forks only if it is free; he waits otherwise. Further, he tries to pick up the other fork. If it is not free, he puts down that one he holds. Finally, when the philosopher has both forks, he eats, and then he puts the forks down in a *non-deterministic* order. Of course, because there are several philosophers not *synchronised* one with each other, it brings another non-determinism. Note, that there are used `formula` structures which are just to avoid duplication of a code (a shorthand for an expression). Also, there are used `label` structures which is a way of identifying sets of states that are of particular interest. It can only be used when specifying properties. Moreover, there is `num_steps` *reward* that counts the number of steps made. It assigns a value 1 to every transition.

3.2 Performed Experiments

Several experiments were done with different instances of the problem, i.e., models for a different number of philosophers. In particular, experiments were performed with $N = 3, 4, 5, 6$. Table 1

[3]PRISM **case studies**: http://www.prismmodelchecker.org/casestudies.

[4]A case study of a **Randomised Dining Philosophers** problem is available at: http://www.prismmodelchecker.org/casestudies/phil.php.

```
mdp

formula lfree = (p2>=0 & p2<=4) | p2=6 | p2=11; // left fork free
formula rfree = (p3>=0 & p3<=3) | p3=5 | p3=7 | p3=10; // right fork free

module phil1
    p1: [0..11];

    [] p1=0 -> (p1'=1); // trying
    [] p1=1 -> 0.5 : (p1'=2) + 0.5 : (p1'=3); // draw randomly
    [] p1=2 & lfree -> (p1'=4); // pick up left
    [] p1=3 & rfree -> (p1'=5); // pick up right
    [] p1=4 & rfree -> (p1'=8); // pick up right (got left)
    [] p1=4 & !rfree -> (p1'=6); // right not free (got left)
    [] p1=5 & lfree -> (p1'=8); // pick up left (got right)
    [] p1=5 & !lfree -> (p1'=7); // left not free (got right)
    [] p1=6 -> (p1'=1); // put down left
    [] p1=7 -> (p1'=1); // put down right
    [] p1=8 -> (p1'=9); // move to eating (got forks)
    [] p1=9 -> (p1'=10); // finished eating and put down left
    [] p1=9 -> (p1'=11); // finished eating and put down right
    [] p1=10 -> (p1'=0); // put down right and return to think
    [] p1=11 -> (p1'=0); // put down left and return to think
endmodule

// construct further modules through renaming
module phil2 = phil1 [p1=p2, p2=p3, p3=p1] endmodule
module phil3 = phil1 [p1=p3, p2=p1, p3=p2] endmodule

rewards "num_steps" // rewards (number of steps)
    [] true : 1;
endrewards

label "hungry" = (p1>0 & p1<8) | (p2>0 & p2<8) | (p3>0 & p3<8);
label "eat" = (p1>=8 & p1<=9) | (p2>=8 & p2<=9) | (p3>=8 & p3<=9);
```

Listing 1: A Randomised Dining Philosophers problem modelled in the PRISM modelling language

shows statistics for the MDPs built for different values of N. The table include: the number of *states* and *transitions* in the MDP representing the model; both the number of *nodes* and *leaves* of the MTBDD which represents the model; the *construction time* which equals the time taken for the system description to be parsed and converted to the MTBDD representing it, and the time to perform reachability, identify the non-reachable states and filter the MTBDD accordingly; and the number of *iterations* required to find the reachable states (which is performed via a BDD based *fixpoint algorithm*).

It was model checked the following *liveness* property: "If a philosopher is hungry, then *eventually*, some philosopher eats.". This was specified using the following formula:

$$\texttt{"hungry"} \Rightarrow \mathrm{P}_{\geq 1}[\mathit{true}\ \mathrm{U}\ \texttt{"eat"}]$$

This property *holds in all states* for all the instances of the problem.

N	Model		MTBDD		Construction	
	States	Transitions	Nodes	Leaves	Time (s)	Iterations
3	956	3,048	765	3	0.004	18
4	9,440	40,120	1,650	3	0.085	24
5	93,068	494,420	2,854	3	0.151	30
6	917,424	5,848,524	4,339	3	0.243	36

Table 1: Statistics for the MDP models built for different values of N for a Randomised Dining Philosophers problem

Further, using model checking, it was found out the answer to a question "What is the *minimum probability*, from a state where someone is hungry, that a philosopher will eat *within* K steps?". For that, the following formula was used:

$$\mathrm{P}_{\mathrm{min}=?}[\mathit{true}\ \mathrm{U}^{\leq K}\ \texttt{"eat"}\ \{\texttt{"hungry"}\}\{\mathrm{min}\}]$$

The resulting probabilities are plotted to the graphs in Figure 5, where K ranges from 0 to 200 steps.

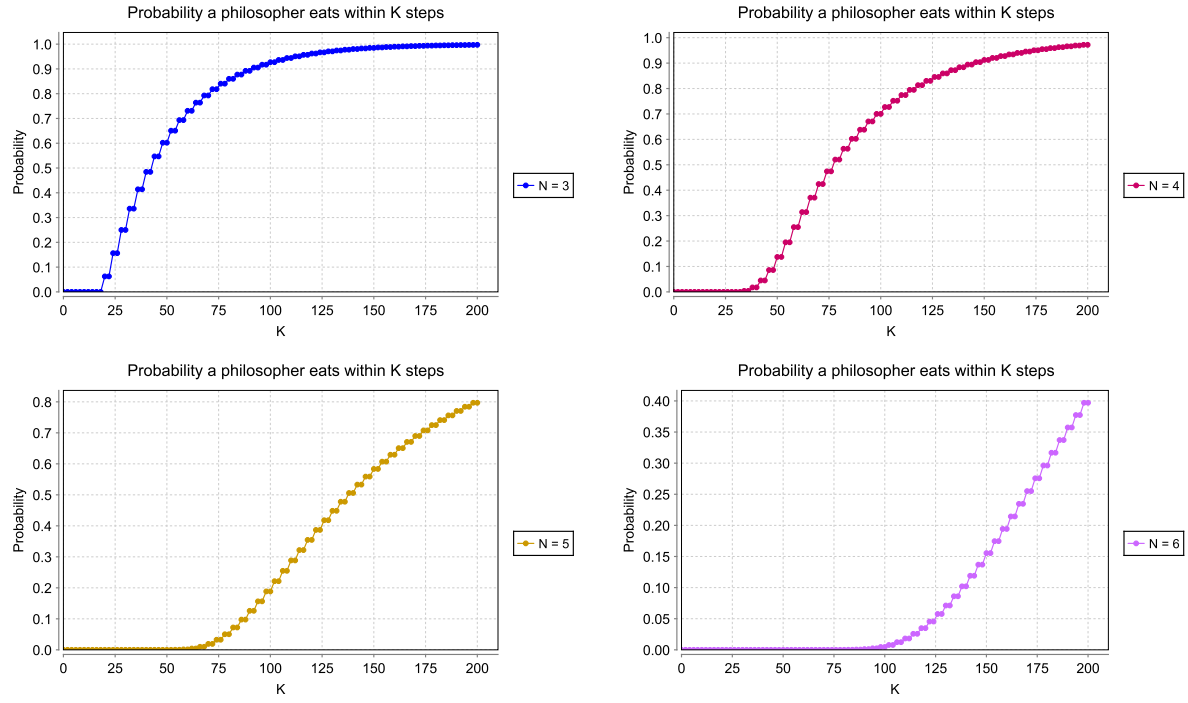


Figure 5: Probabilities that a philosopher eats within K steps plotted for different values of N

The last property decided using model checking, in this example, is "From a state where someone is hungry, what is the *maximum expected number of steps* until a philosopher eats?" It can be answered using the defined *reward* with the following formula:

$$\mathrm{R}\{\texttt{"num\_steps"}\}_{\mathrm{max}=?}[\mathrm{F}\ \texttt{"eat"}\ \{\texttt{"hungry"}\}\{\mathrm{max}\}]$$

The result is ≈ 51 for $N = 3$, ≈ 89 for $N = 4$, ≈ 149 for $N = 5$, and ≈ 244 for $N = 6$.

4 Custom Experiments

The Chapter discusses performed custom experiments, concerned *population models*. In particular, it focuses on the modelling of *virus development*. The idea is based on the project assignment from [10]. A description of the model is given in Chapter 4.1. Next, experiments are described in Chapter 4.2.

4.1 Description of the Model

Within this project, *population models* that use *chemical reaction networks* are considered, where *transition rates* correspond to *mass-action kinetics*. Here is a very simplified example: it is given a chemical reaction network $A + B \xrightarrow{k} C$, where A, B, C are some species. That means that A and B together products C. And a transition rate is $r = |A| \cdot |B| \cdot k$.

In the considered model, there is virus development in a *closed population*. The population contains (i) *healthy individuals (Z)* that can be infected, (ii) *infected individuals (N)* that can be healed, (iii) and *healed individuals (U)* that that can not be infected anymore. The initial population is $Z = 95$, $N = 5$, $U = 0$. The development of the virus is influenced by the following two *reactions*: (i) *infection*: $Z + N \xrightarrow{k_i} N + N$, where $k_i \in [0.001, 0.011]$; (ii) and *healing*: $N \xrightarrow{k_r} U$, where $k_r \in [0.01, 0.11]$.

It is modelled as a CTMC because these chemical reactions naturally operate in *continuous-time*. A source code in the PRISM modelling language is given in Listing 2. In the code, there are some defined constants define that the initial population, and undefined constants with reactions rates, these are given during model checking of particular properties. The model contains a `virus` *module* that implements appropriate reaction network. There are variables that correspond to Z, N, and U. Further, there are implemented commands that correspond to appropriate reactions. Transaction rates in these commands are as follows (w.r.t. mass-action kinetics): $r_i = |Z| \cdot |N| \cdot k_i = z \cdot n \cdot k_i$, and $r_r = |N| \cdot k_r = n \cdot k_r$.

```
1  ctmc
2
3  // initial values
4  const int z_init=95;
5  const int n_init=5;
6  const int u_init=0;
7  const int total=z_init+n_init+u_init;
8
9  const double ki; // rate of infection
10 const double kr; // rate of healing
11
12 module virus
13   z: [0..total] init z_init; // healthy individuals
14   n: [0..total] init n_init; // infected individuals
15   u: [0..total] init u_init; // healed individuals
16
17   [] z>0 & n>0 & n<total -> z*n*ki : (z'=z-1) & (n'=n+1); // infection
18   [] n>0 & u<total -> n*kr : (n'=n-1) & (u'=u+1); // healing
19 endmodule
```

Listing 2: Virus development modelled in the PRISM modelling language

4.2 Performed Experiments

Table 2 demonstrates statistics for the CTMC built for the model.

$k_i; k_r$	Model		MTBDD		Construction	
	States	Transitions	Nodes	Leaves	Time (s)	Iterations
0.011; 0.11	5,136	10,076	104,598	1,241	0.266	196

Table 2: Statistics for the CTMC model built for the virus development model

Firstly, it was model checked the following: "What is the probability that the infection *eventually disappears*?". It was specified using the formula:

$$\mathrm{P}_{=?}[\mathrm{F}\ n = 0]$$

The outcome is plotted to the graph in Figure 6, where k_i and k_r range in proper intervals. It is evident that the infection *eventually, indeed disappears*.

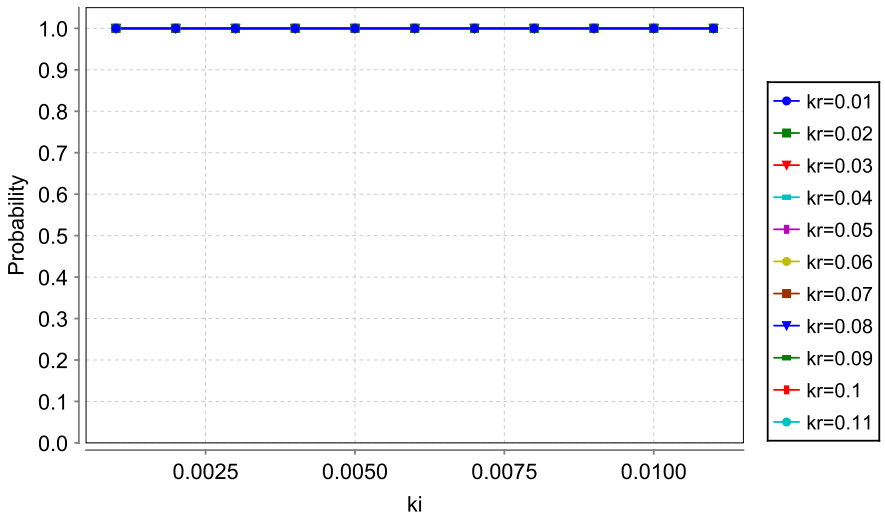


Figure 6: The probability of the disappearance of the infection plotted for different values of parameters k_i and k_r

Furthermore, it was computed "What is the probability that the infection *takes at least* 100 time units, and it *disappears to* 120 time units?". For this, it was constructed the following formula:

$$\mathrm{P}_{=?}[n > 0\ \mathrm{U}^{[100,120]}\ n = 0]$$

The result of this model checking is visible on the plotted graph in Figure 7. Again, the parameters rage over the permissible intervals.

5 Conclusion

This essay sums up the *probabilistic model checker* PRISM and its usage. In the first part, it discusses *probabilistic models* and *logic* that PRISM support. Moreover, it briefly describes implementation techniques used in PRISM together with the PRISM *modelling language*. In the next parts, it describes some performed experiments which partially outlines features of PRISM. In particular, there is a case

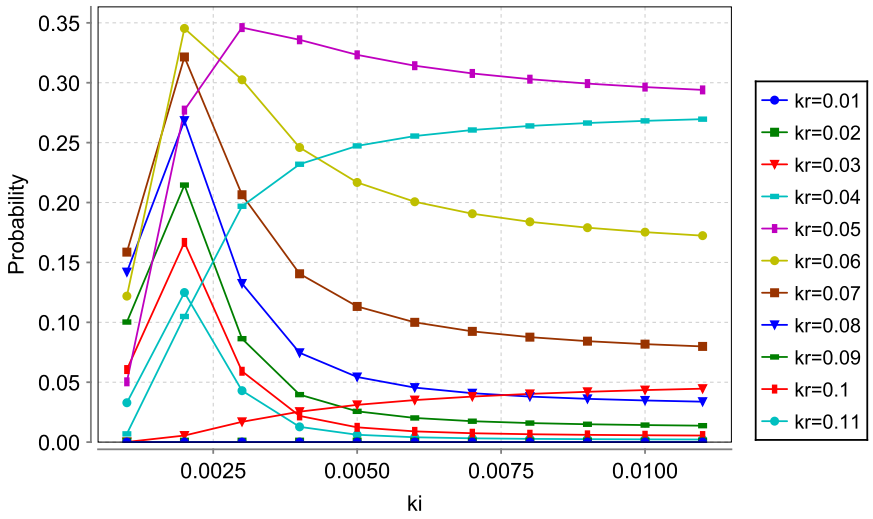


Figure 7: The probability that the infection takes at least 100 time units, and it disappears to 120 time units

study from *randomised distributed algorithms*, where the system is modelled as a *Markov decision process*; and experiments with *population models* that use *chemical reaction networks*, i.e., the system is modelled as a *continuous-time Markov chain*.

References

[1] Baier, C.; Katoen, J.-P.: *Principles of model checking*. Cambridge, Massachusetts; London, England: MIT Press. 2008. ISBN 978-0-262-02649-9.

[2] Clarke, E. M.; Henzinger, T. A.; Veith, H.; Bloem, R.: *Handbook of Model Checking*. Cham: Springer International Publishing. first edition. 2018. ISBN 978-3-319-10574-1. doi:10.1007/978-3-319-10575-8.

[3] Forejt, V.; Kwiatkowska, M.; Norman, G.; Parker, D.: Automated Verification Techniques for Probabilistic Systems. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*, *Lecture Notes in Computer Science*, vol. 6659, edited by M. Bernardo; V. Issarny. Springer, Berlin, Heidelberg. June 2011. ISBN 978-3-642-21454-7. pp. 53–113. doi:10.1007/978-3-642-21455-4_3.
Retrieved from: http://www.prismmodelchecker.org/bibitem.php?key=FKNP11

[4] Kwiatkowska, M.; Norman, G.; Parker, D.: Probabilistic Model Checking. Lecture Notes in BISS'7 School. University of Oxford, Department of Computer Science. March 2007.
Retrieved from: http://www.prismmodelchecker.org/lectures/biss07

[5] Kwiatkowska, M.; Norman, G.; Parker, D.: Stochastic Model Checking. In *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, *Lecture Notes in Computer Science (Tutorial Volume)*, vol. 4486, edited by M. Bernardo; J. Hillston. Springer, Berlin, Heidelberg. June 2007. ISBN 978-3-540-72482-7. pp. 220–270. doi:10.1007/978-3-540-72522-0_6.
Retrieved from: http://www.prismmodelchecker.org/bibitem.php?key=KNP07a

[6] Kwiatkowska, M.; Norman, G.; Parker, D.: Advances and Challenges of Probabilistic Model Checking. In *Proc. 48th Annual Allerton Conference on Communication, Control, and Computing*. Invited paper. Allerton, IL, USA: IEEE Press. October 2010. ISBN 978-1-4244-8215-3. pp. 1691–1698. doi:10.1109/ALLERTON.2010.5707120.
Retrieved from: http://www.prismmodelchecker.org/bibitem.php?key=KNP10c

[7] Kwiatkowska, M.; Norman, G.; Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, *Lecture Notes in Computer Science*, vol. 6806, edited by G. Gopalakrishnan; S. Qadeer. Springer, Berlin, Heidelberg. July 2011. ISBN 978-3-642-22109-5. pp. 585–591. doi:10.1007/978-3-642-22110-1_47.
Retrieved from: http://www.prismmodelchecker.org/bibitem.php?key=KNP11

[8] Kwiatkowska, M.; Parker, D.: Probabilistic Model Checking. Lecture Notes in ESSLLI'10 Summer School, Copenhagen. University of Oxford, Department of Computer Science. August 2010.
Retrieved from: http://www.prismmodelchecker.org/lectures/esslli10

[9] Parker, D.: Probabilistic Model Checking. Lecture Notes in Probabilistic Model Checking. University of Oxford, Department of Computer Science. 2011.
Retrieved from: http://www.prismmodelchecker.org/lectures/pmc

[10] Češka, M.: Markov Chains — Modelling and Analysis of Probabilistic Systems. Lecture Notes in Model-Based Analysis. Brno University of Technology, Faculty of Information Technology. 2020.