Quantitative analysis with the probabilistic model checker PRISM [1]

Marta Kwiatkowska Gethin Norman David Parker [2]

School of Computer Science, University of Birmingham
Edgbaston, Birmingham B15 2TT, UK

Abstract

Probabilistic model checking is a formal verification technique for establishing the correctness, performance and reliability of systems which exhibit stochastic behaviour. As in conventional verification, a precise mathematical model of a real-life system is constructed first, and, given formal specifications of one or more properties of this system, an analysis of these properties is performed. The exploration of the system model is exhaustive and involves a combination of graph-theoretic algorithms and numerical methods. In this paper, we give a brief overview of the probabilistic model checker PRISM (`www.cs.bham.ac.uk/~dxp/prism`) implemented at the University of Birmingham. PRISM supports a range of probabilistic models and specification languages based on temporal logic, and has been recently extended with costs and rewards. We describe our experience with using PRISM to analyse a number of case studies from a wide range of application domains. We demonstrate the usefulness of probabilistic model checking techniques in detecting flaws and unusual trends, focusing mainly on the quantitative analysis of a range of best, worst and average-case system characteristics.

Key words: Automatic verification, temporal logic, Markov models, probabilistic model checking, performability, reliability, dependability.

1 Introduction

Model checking is an automatic model-based verification approach that explores all system executions and is therefore more powerful than testing or simulation-based system analysis techniques. Models can be created manually using modelling languages tailored to the particular application domain,

[1] Supported in part by FORWARD and EPSRC projects GR/S11107 and GR/S46727.

[2] {mzk,gxn,dxp}@cs.bham.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

for example a hardware description language, or extracted via abstract interpretation from actual source code in C/C++ or Java. A model checker aims to establish that the model satisfies a given specification, usually stated in a variant of temporal logic, or else produce error diagnostics. Since its introduction in 1980s, model checking has made great advances, becoming a leading research focus as well as standard industrial practice. When applied as part of a design process, its ability to detect errors in the designs before manufacture can improve reliability and reduces production costs both in hardware designs (e.g. Intel) and software (e.g. SLAM at Microsoft).

The vast majority of research in model checking has concerned discrete behavioural aspects, such as nondeterminism and concurrency. Recent developments in technology – the increasing trend for mobile, portable, ubiquitous, adaptive, self-organising systems – have substantially raised the profile of probabilistic, and more generally *quantitative* modelling and verification technologies for software. Features such as real-time and probability are already utilised in real-world distributed protocols (e.g. Bluetooth, IEEE 802.11). Indeed, randomisation is key to achieve symmetric distributed solutions, self-configuring protocols, self-organising systems, fault-tolerant algorithms and scalable protocols. Probability also plays an important role in modelling uncertainty, in planning and decision making, and for analysing performance and dependability.

Probabilistic model checking is a relatively recent development which aims to deliver automatic verification technology for probabilistic systems. The theoretical aspects of probabilistic model checking have been studied since first introduced by Hart, Sharir and Pnueli [18]. As in conventional model checking, a model of the probabilistic system, usually some variant of a Markov chain, is built and then subjected to algorithmic analysis in order to establish whether it satisfies a given specification. The specifications are usually stated as formulae of probabilistic temporal logic which, in addition to conventional modalities, may include probabilistic operators whose outcome is true/false depending on the probability of certain executions. The model checking procedure combines traversal of the underlying transition graph with numerical solution methods. The model checker can either produce an answer yes or no, by comparing the obtained probability with the given threshold, or simply return the likelihood of the occurrence of executions. For example, suitable correctness properties for a randomised leader election protocol are "the leader is eventually elected with probability 1" and "the expected time to leader election is 10 ms", and for a multimedia protocol "the probability of a frame being delivered within 5 ms is at least 90%" and "the worst case time to deliver a frame is 1.5 ms".

Although algorithms for model checking probabilistic systems have been known since the mid-1980s [53], it is only recently that experimental, tool implementation work has begun. Software tools available for probabilistic model checking include PRISM [27,45], $\mathsf{E} \vdash \mathsf{MC}^2$ [20], and Rapture [23]. PRISM, the internationally leading and widely used Probabilistic model checker, has been developed at the University of Birmingham and used to model and analyse over 30 real-world protocols. PRISM provides support for three types of models and a range of quantitative analysis techniques such as expected time, average power consumption, and probability of delivery by a deadline. It relies on the use of symbolic model checking technology, employing sophisticated data structures based on binary decision diagrams (BDDs) [28,34] and efficient algorithms [34,55]. PRISM combines graph-theoretical analysis with numerical solution (for exact best/worst case analysis), statistical, simulation-based methods (for approximate analysis) and parallelisation.

In this paper, we report on our experiences with using PRISM to model and analyse a range of case studies, from domains as wide-ranging as randomised coordination algorithms and biochemical reactions. We found that the techniques are expressive enough to analyse properties of genuine interest to protocol designers, and have indeed proved useful through discovering errors. We demonstrate the usefulness of *quantitative* analysis against properties based on temporal logic, but returning quantities computed by model checking, for example likelihood of termination or average power consumption, rather than a true/false answer. The advantage of such quantitative analysis is that the results can be plotted as graphs that can be inspected for trends and anomalies. We also illustrate the merits of exhaustive probabilistic model checking over simulation-based techniques. In particular, we are able to compute exact quantities, rather than approximations based on a large number of simulations, thus enabling to arrive at complete, exhaustive conclusions, e.g., computing the best- and worst-case performance for all possible parameter values, or under any scheduling.

The paper is structured as follows. We begin with a brief introduction to the topic of probabilistic model checking in Section 2. Then we give an overview of the PRISM model checker in Section 3, followed by its modelling language and property specification notation respectively in Sections 4 and 5. Three case studies, self-stabilisation, dynamic voltage and molecular reactions, are described in Section 6, and results of their analysis discussed together with example models and property specifications. We conclude with Section 7.

2 Probabilistic model checking

A probabilistic model checker takes two types of inputs, a *probabilistic model* and a *property specification*. The former is usually described in a high-level model description language, which is then transformed into an internal representation suitable for analysis. The latter is typically based on temporal logic, enriched with probabilistic operators, and may include additional features such as time bounds or costs depending on the model. The model checker explores the model and produces two types of outputs, either true/false to indicate whether the specification holds in the model, or the numerical value, for example, the probability or expected time for each state.

2.1 The Models

The simplest probabilistic models are variants of (discrete space) Markov chains, and namely *discrete-time Markov chains* (DTMCs), *Markov decision processes* (MDPs) and *continuous-time Markov chains* (CTMCs). All give rise to transition systems where the transition relation between states is probabilistic. The models can be endowed with additional information labelling the states and transitions, for example atomic propositions or costs.

A DTMC is fully probabilistic; it is given as a (finite) set of states S, a subset of initial states $\bar{S} \subseteq S$ and a transition probability matrix $\mathbf{P} : S \times S \to [0,1]$. For each pair s, s' of states, the probability of making a transition from s to s' is given by $\mathbf{P}(s,s')$. We require that $\sum_{s' \in S} \mathbf{P}(s,s') = 1$ for all states $s \in S$. Terminating states can be modelled by adding a self-loop, i.e. setting $\mathbf{P}(s,s) = 1$.

Markov decision processes (MDPs) extend DTMCs by additionally allowing non-deterministic behaviour that is needed, for example, to model asynchronous parallel composition. An MDP is defined by a set of states S, a set of initial states $\bar{S} \subseteq S$ and a function *Steps* which maps each state in S to finite non-empty set of probability distributions over S. Intuitively, the next transition from a state $s \in S$ is determined by nondeterministically selecting an element μ of $Steps(s)$ and then choosing a state probabilistically, according to the distribution μ.

Continuous-time Markov chains (CTMCs), similarly to DTMCs, can model probabilistic behaviour only, but they allow the modelling of real (continuous) time, rather than discrete time-steps. Formally, a CTMC is defined by a set of states S, a set of initial states $\bar{S} \subseteq S$ and a transition rate matrix $\mathbf{R} : S \times S \to \mathbb{R}$. This gives the *rate* $\mathbf{R}(s,s')$ at which transitions occur between each pair of states s, s'. The probability of moving from s to s' within t ($\in \mathbb{R}^{>0}$) time units is described as a negative exponential distribution $1 - e^{-\mathbf{R}(s,s')t}$ with the rate taken as the parameter. If $\mathbf{R}(s,s') > 0$ for more than one state s', a *race* between the outgoing transitions from s occurs. This means that the probability of moving from s to s' is equal to the probability that the delay of going from s to s' "finishes before" the delays of any other outgoing transition from s.

In models of all three types, a *path* is a sequence of states, each consecutive pair of which is connected by a transition, i.e. for which the corresponding probability or rate is non-zero. A path of a model corresponds to a single run or execution of the system which the model represents.

2.2 Property specifications

In conventional model checking, the specifications typically aim to ascertain whether a particular event eventually occurs, possibly with additional quantification over paths. In probabilistic model checking, since the transition relation is probabilistic, we are interested in calculating *the probability* of events occurring. This is achieved through extending temporal logics such as CTL or LTL with a *probabilistic operator* $\mathtt{P} \sim p\,[\cdot]$, that can be applied to sets of paths returning their likelihood. This operator has a probability bound ($p \in [0, 1]$) and a relational operator ($\sim \in \{<, \leq, \geq, >\}$). Comparing the resulting probability of the specified set of paths with the bound p of the enclosing operator yields a conventional boolean formula. In the case of MDP models, there are two types of branching, nondeterministic, determined by a scheduler, and probabilistic, governed by the probability distribution. This must be reflected in the interpretation of properties, which must be of the form '*under any scheduling of processes*, yielding the *minimum/maximum* over all the possible ways of resolving nondeterminism instead of the exact probability.

The specification notations also take into account additional decorations in the model, for example real-valued time and costs and rewards. Real-valued time bounds are allowed to reason about CTMCs (the logic CSL), and the logics are further enhanced with additional *expectation operators*.

2.3 Probabilistic analysis methods

Once a probabilistic model has been built, it can be subjected to various types of analysis. Conventional model checking methods focus on the underlying transition graph, allowing for *reachability* analysis and temporal logic *model checking*. On the other hand, conventional probabilistic analysis uses *simulation* or *analytical solution* methods to obtain performance and quality of service estimates, typically represented as functions of system parameters. *Probabilistic* model checking combines probabilistic analysis and conventional reachability in a single tool. In comparison with simulation, its advantage is full coverage of the executions and therefore exact answers, and in contrast with analytical approaches a more detailed analysis, especially the 'corner cases', is often possible. The drawback is the state-space explosion, which can be addressed through *statistical*, simulation-based methods at a cost of approximate answers.

Probabilistic model checking algorithm proceed through a combination of graph-traversal and numerical computation. *Qualitative* probabilistic model checking, i.e. the case of probability bounds being 0 or 1, involves only graph traversal. For *quantitative* model checking, the core component is numerical solution of linear equation systems and linear optimisation problems. For a detailed introduction to the field of probabilistic model checking see, for example, [46,4].

3 The PRISM model checker

We now give a high-level overview of the functionality of the PRISM model checker.

3.1 Tool overview

PRISM [27,45] is a probabilistic model checker developed at the University of Birmingham. It accepts probabilistic models described in its *modelling language*, a simple, high-level state-based language. Three types of probabilistic models are supported directly; these are discrete-time Markov chains (DTMCs), Markov decision processes (MDPs), and continuous-time Markov chains (CTMCs). Additionally, probabilistic timed automata (PTAs) are partially supported, with the subset of diagonal-free PTAs with *digital clocks* supported directly [29].

The *property specification* language of PRISM is based on two existing temporal logics. PCTL [17,8] is the probabilistic computation tree logic, an extension of CTL with the probabilistic operator, which is appropriate for the discrete-time models (DTMCs and MDPs). For CTMCs, CSL [2,5] (continuous stochastic logic) is supported; it also includes the probabilistic operator but additionally allows real-valued time bounds to be expressed. Probabilistic timed automata have a logic PTCTL, an extension of TCTL, a subset of which is supported via connection to Kronos [12].

A simplified version of the overall structure of the tool is shown in Figure 1.

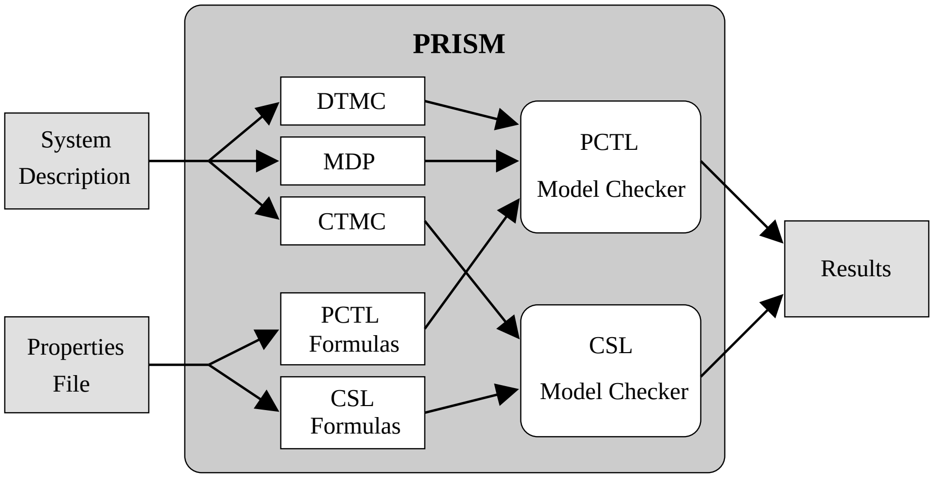


Fig. 1. The structure of PRISM

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. This represents the set of all feasible configurations which can arise in the modelled system.

Next, the specification is parsed and appropriate model checking algorithms are performed on the model by induction over syntax. In some cases, such as for properties which include a probability bound, PRISM will simply report a true/false outcome, indicating whether or not each property is satisfied by the current model. More often, however, properties return *quantitative* results and PRISM reports, for example, the actual probability of a certain event occurring in the model. Furthermore, PRISM supports the notion of *ex-*

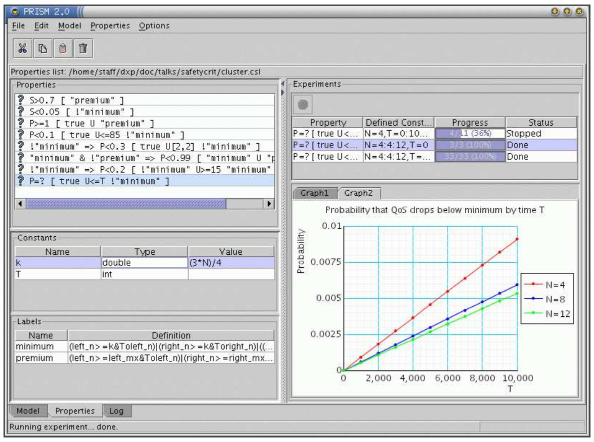


Fig. 2. A screenshot of the PRISM graphical user interface

periments, which is a way of automating multiple instances of model checking. This allows the user to easily 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 such as a spreadsheet, or plotted as a graph. For the latter, PRISM incorporates substantial graph-plotting functionality. This is often a very useful way of identifying interesting patterns or trends in the behaviour of a system. The reader is invited to consult the "Case Studies" section of the PRISM website [45] for many examples of this kind of analysis.

Figure 2 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. Alternatively, all model checking functionality is also available in a command-line version of the tool. PRISM is a *free*, *open source* application. It presently operates on Linux, Unix, Windows and Macintosh operating systems. Both binary and source code versions can be downloaded from the website [45].

3.2 Implementation

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). These provide compact representations 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 we need to store numerical values, PRISM uses *multi-terminal* binary decision diagrams (MTBDDs) [11,3] and a number of variants [28,43,34] 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. Since its release in 2001, the model size capacity and tool efficiency has increased substantially (10^{10} is feasible for CTMCs and higher for other types of models). PRISM employs and builds upon the Colorado University Decision Diagram package [49] by Fabio Somenzi which implements BDD/MTBDD operations.

The underlying computation in PRISM involves a combination of:

- *graph-theoretical algorithms*, for reachability analysis, conventional temporal logic model checking and *qualitative* probabilistic model checking, 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 solution of linear equation systems, it supports a range of well-known techniques, including the Jacobi, Gauss-Seidel and SOR (successive over-relaxation) 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, which is also known as randomisation or Jensen's method.

In fact, for numerical computation, the tool actually provides three distinct numerical *engines*. The first is implemented purely in MTBDDs (and BDDs); the second uses sparse matrices; and the third is a hybrid, using a combination of the 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 (see [28,43]). By default, PRISM uses the hybrid engine.

4 The PRISM modelling language

The PRISM modelling language is a simple, state-based language based on the Reactive Modules formalism of Alur and Henzinger [1]. In this section, we give a brief outline of the language. For a full definition of the language and its semantics, see [25]. A wide range of examples can be found both in the "Case Studies" section of the PRISM website [45] and in the distribution of the tool itself.

4.1 Modules, variables and commands

The fundamental components of the PRISM language are *modules* and *variables*. Variables are typed (integers, reals and booleans are supported) and

```
// A coin process

dtmc

const int HEADS = 1;
const int TAILS = 2;

module coin
    x : [0..3] init 0;

    [] (x = 0) → 0.5 : (x' = HEADS) + 0.5 : (x' = TAILS);
    [] (x > 0) → 1 : (x' = x);
endmodule
```

Fig. 3. The PRISM Language: Example of a Coin

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. The behaviour of each module is described by a set of *commands*. A command takes the form:

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

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 to the transitions.

An example of a module is given in Figure 3. It implements an electronic coin, which will assign *HEADS* or *TAILS* to the variable x with probability 0.5 (x' denotes updated variable) when x is zero, and otherwise it will retain the previous value of x. In this case, there is only one initial state, but PRISM allows the specification of a set of initial states, see [25].

The module is a DTMC (keyword `dtmc`). The other two possibilities are Markov decision processes and continuous-time Markov chains, keywords `mdp` and `ctmc` respectively. The interpretation of λ_i varies depending on the model, i.e. it is a probability for DTMCs and rate for CTMCs. If the guards are overlapping, say $(x > 0)$ were replaced with $(x = 0)\&(x > 0)$, this indicates (local) *nondeterminism*, which is illegal within a DTMC but allowed in MDPs.

4.2 Composing modules

The probabilistic model corresponding to a PRISM language description is constructed as the parallel composition of its modules. In every state of the

```
// N-place queue + server

ctmc

const int N = 10;
const double mu = 1/10;
const double lambda = 1/2;
const double gamma = 1/3;

module queue
    q : [0..N];

    [] q < N -> mu : (q' = q + 1);
    [] q = N -> mu : (q' = q);
    [serve] q > 0 -> lambda : (q' = q - 1);
endmodule

module server
    s : [0..1];

    [serve] s = 0 -> 1 : (s' = 1);
    [] s = 1 -> gamma : (s' = 0);
endmodule
```

Fig. 4. The PRISM Language: N-place queue and server example

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, the choice is *probabilistic*, with each enabled command selected with equal probability; for an MDP, it is *nondeterministic*; and for CTMCs it is modelled as a *race condition*.

PRISM also supports multi-way *synchronisation* in the style of process algebras. For synchronisation to take effect, commands are labelled with *actions* that are placed between the square brackets. We illustrate this with an example of a model of an N-place queue of jobs and a server which removes jobs from the queue and processes them. The PRISM code can be found in Figure 4.

There are two modules, one modelling the queue and the other the server. For example, the *serve* action in this command from Figure 4:

$$[serve]\ q > 0 \ \rightarrow\ lambda : (q' = q - 1);$$

is used to force two or more modules to make transitions simultaneously (i.e. to synchronise). For example, in state $(3, 0)$ (i.e. $q = 3$ and $s = 0$), the composed model can move to state $(2, 1)$, synchronising over the *serve* action. The rate of this transition is equal to the product of the two individual rates (in this case, *lambda* $\cdot$ 1 = *lambda*). The product of two rates does not always meaningfully represent the rate of a synchronised transition. A common technique, as seen here, is to make one action *passive*, with rate 1, and one action *active*, which actually defines the rate for the synchronised transition. By default, all modules are combined using the standard CSP parallel composition (i.e. modules synchronise over all their common actions). In addition, PRISM supports several other CSP parallel operators (alphabetised parallel, interleaving, etc) and is able to import models written in a subset of the stochastic process algebra PEPA [21].

4.3 Costs and rewards

PRISM includes support for the specification and analysis of properties based on *costs* and *rewards*. This means that PRISM can be used to reason, for example, about properties such as "expected time", "expected number of lost messages" or "expected power consumption".

The basic idea is that probabilistic models (of all three types) developed in PRISM can be augmented with costs or rewards: real values associated with certain states or transitions of the model (costs are generally perceived to be "bad" and rewards to be "good" but, numerically, the two are identical).

Rewards are associated with models using the `rewards`...`endrewards` construct. State rewards can be specified using multiple reward items, each of the form "*guard* : *reward*;", where *guard* is a predicate (over all the variables of the model) and *reward* is an expression (containing any variables, constants, etc. from the model). For example:

```
rewards
    x = 0 : 100;
    x > 0 & x < 10 : 2 * x;
    x = 10 : 100;
endrewards
```

assigns a reward of 100 to states satisfying $x = 0$ or $x = 10$ and a reward of $2 * x$ to states satisfying $x > 0$ & $x < 10$. Note that a single reward item can assign different rewards to different states, depending on the values of model variables in each one. Any states which do not satisfy the guard of any reward item will have no reward assigned to them. For states which satisfy multiple guards, the reward assigned to the state is the sum of the rewards for all the corresponding reward items.

Rewards can also be assigned to transitions of a model, which are specified in a similar fashion, see [25].

5 Property specifications

Properties of PRISM models are expressed in a language based on the logics PCTL (for DTMCs and MDPs) and CSL (for CTMCs), probabilistic extensions of the classical temporal logic CTL originally introduced in [17,8] (PCTL) and [2,5] (CSL). PRISM supports numerous additional customisations and extensions of these two logics; for full details see [25].

As an illustration, we list some typical examples of properties which PRISM can handle, giving both the PRISM syntax with respect to presumed atomic propositions (e.g. *elected*, *init*) and a natural language translation:

- $\mathtt{P} \geq 1 \; [\; \mathtt{true} \; \mathtt{U} \; \mathit{elected} \;]$
  "the algorithm eventually elects a leader with probability 1"
- $\mathit{init} \;\Rightarrow\; \mathtt{P} < 0.1 \; [\; \mathtt{true} \; \mathtt{U}{\leq}100 \;\; \mathit{num\_errors} > 5 \;]$
  "from an initial state, the probability that more than 5 errors occur within the first 100 time units is less than 0.1"
- $\mathit{down} \;\Rightarrow\; \mathtt{P} > 0.75 \; [\; !\mathit{fail} \; \mathtt{U}[1,2] \; \mathit{up} \;]$
  "when a shutdown occurs, the probability of system recovery being completed in between 1 and 2 hours without further failures occurring is greater than 0.75"
- $\mathtt{S} < 0.01 \; [\; \mathit{num\_routers} < \mathit{min\_routers} \;]$
  "in the long-run, the probability that an inadequate number of routers are operational is less than 0.01"

The satisfaction of a property (i.e. whether it is true or false) is defined for a single state of a model. When analysing a property, PRISM considers it to be true if it is satisfied in *all* states of the model, and false otherwise. As in the second example above, properties can be prefixed with an implication to check satisfaction in a certain subset of model states.

The two principal operators in PRISM's property specification language are the `P` (probabilistic) and `S` (steady-state) operators. By default, in both cases, these operators include a probability bound (≥ 1, < 0.1, > 0.75 and < 0.01 in the examples above). Informally, a property using the probabilistic operator, such as $\mathtt{P}{>}0.75 \, [\mathit{pathprop}]$, is true in a state s of a DTMC, MDP or CTMC if "the probability that path property *pathprop* is satisfied by the paths from state s is greater than 0.75".

PRISM supports path properties constructed from three temporal operators: `X` ("next"), `U` ("until"), and $\mathtt{U}\mathit{time}$ ("bounded until"). The first, $\mathtt{X}\; a$, is satisfied if a is true in the next state. The second, $a \; \mathtt{U} \; b$, is satisfied if b is eventually true and a is true up until that point. One common usage of this type of property is the case where a is `true` (as in several of the examples above). The path property $\mathtt{true} \; \mathtt{U} \; b$ means simply that b is eventually true. The third type, $a \; \mathtt{U}\mathit{time} \; b$, is satisfied if b becomes true within the time interval *time* and a is true up until that point. For DTMCs and MDPs, where time proceeds in discrete steps, the time interval *time* is simply an integer upper bound, e.g. $\mathtt{U} \leq 10$. For CTMCs, which model real (continuous) time, *time* can be an arbitrary interval of the reals, as in these examples: ≤ 1.5, ≥ 5.0, $[12.75, 13.25]$.

For a DTMC, the probability measure of the set of paths from a state s which satisfy a particular path property of the types discussed above is well-defined; see e.g. [17]. Similarly, for a CTMC, the probability measure for such a set of paths can also be defined; see e.g. [5]. For MDPs, however, a probability measure can only be feasibly defined once all nondeterminism has been removed. Hence, the actual meaning of the property `P` *bound* [*pathprop*] for an MDP is taken to be "the probability that path property *pathprop* is satisfied by the paths from state s meets the bound *bound* for *all possible resolutions of nondeterminism*". This means that, for an MDP, properties using the `P` operator actually reason about the *minimum* or *maximum* probability, over all possible resolutions of nondeterminism, that a certain type of behaviour is observed. This depends on the bound attached to the `P` operator: a lower bound ($>$ or $\geq$) relates to minimum probabilities and an upper bound ($<$ or $\leq$) to maximum probabilities. For more details on this, see e.g. [6].

The steady-state operator `S` is used to reason about the "steady-state" behaviour of a model, i.e. its behaviour in the "long-run" or "equilibrium". Although this could in principle relate to all three model types, PRISM currently only provides support for CTMCs. The definition of steady-state (long-run) probabilities for finite CTMCs is well defined (see e.g. [50]). Informally, a property such as `S`*bound* [*prop*] is true in a state s of a CTMC if "starting from s, the steady-state (long-run) probability of being in a state which satisfies *prop*, meets the bound *bound*".

5.1 Quantitative probability calculations

In PRISM, we can also directly specify properties which evaluate to a *numerical value*. This is achieved by replacing the probability bounds from the `P` and `S` operators with =? and is illustrated in the following examples:

- `P` =? [! *proc2_terminate* `U` *proc1_terminate*]
  "the probability that process 1 terminates before process 2 does"
- `Pmax` =? [`true` `U`$\leq T$ (*message_lost*>10)]
  "the maximum probability that more than 10 messages have been lost by time T"
- `S` =? [(*queue_size*/*max_size*)>0.75]
  "the long-run probability that the queue is more than 75% full"

Note that this is only allowed when the `P` or `S` in question is the outermost operator of the property.

For MDPs, the probabilities can only be computed once the nondeterminism has been resolved. Hence, PRISM actually computes either the *minimum* or *maximum* probability of a path property being satisfied, quantifying over all possible resolutions (i.e. the best and worst cases). Therefore, for MDPs we use either `Pmin` =? or `Pmax` =?.

By default, the result for properties of this kind is the probability for the initial state of the model. It is also possible, however, to obtain the probability for an arbitrary state, as shown in the following example:

- $\mathtt{P} =?\ [\ \mathit{queue\_size} \leq 5\ \mathtt{U}\ \mathit{queue\_size} < 5\ \{\mathit{queue\_size} = 5\}\ ]$
  "the probability, from the state where the queue contains 5 jobs, of the queue processing at least one job before another arrives"

Furthermore, it is possible to compute the minimum or maximum probability for a particular class of states, e.g.:

- $\mathtt{P} =?\ [\ !\,\mathit{proc2\_terminate}\ \mathtt{U}\ \mathit{proc1\_terminate}\ \{\mathit{init}\}\{\mathtt{min}\}\ ]$
  "the minimum probability, over all possible initial configurations, that process 1 terminates before process 2 does"

5.2 Specification of reward-based properties

As described in Section 4.3, PRISM models can be augmented with information about rewards. Properties can then be analysed by PRISM which relate to the *expected value* of these rewards. These are specified using the `R` operator, which works in a very similar fashion to the `P` and `S` operators. The following are some typical examples:

- $\mathtt{R} =?\ [\ \mathtt{I} = 100\ ]$
  "after 100 time units, the expected number of packets awaiting delivery"
- $\mathtt{R} =?\ [\ \mathtt{C} \leq 24\ ]$
  "the expected power consumption during the first 24 hours of operation"
- $\mathtt{Rmax} =?\ [\ \mathtt{F}\ \mathit{completed}\ ]$
  "the worst-case (over all possible scheduling of processes) expected number of messages lost during the execution of the protocol"
- $\mathtt{R} =?\ [\ \mathtt{F}\ \mathit{elected}\ \{\mathtt{init}\}\{\mathtt{max}\}\ ]$
  "from any initial configuration, the worst-case expected number of steps required for the leader election algorithm to complete"
- $\mathtt{R} < 10\ [\ \mathtt{S}\ ]$
  "the long-run expected queue-size is less than 10"

Note the meaning ascribed to the properties is, of course, dependent on the definitions of the rewards themselves. Note also that there are two distinct types of interpretations of rewards. Firstly, they can be considered to be *instantaneous*, where a measure of interest is simply the value of a state reward at a particular time instant, e.g. "queue size". These are typically used with the `I` or `S` reward operators. Secondly, rewards can be *cumulative*, where the values must be summed to provide a meaningful result, e.g. "number of messages lost", "number of steps", "time", "power consumption". These are usually analysed with the `C`, `F` or `S` reward operators. Finally, we also point out that, for cumulative rewards in CTMC models, the reward assigned to each state is assumed to be the *rate* at which reward is accumulated in that state, i.e. if the reward in a state is r and the state is occupied for time t, the reward cumulated during this time will be $r \cdot t$.

6 PRISM case studies

PRISM has been successfully applied to a large number of case studies in a wide range of application domains, listed below. PRISM code for many of them is also distributed with the tool itself.

- Analysis of quality of service (QoS) properties of several real-time communication protocols, including Bluetooth [15], IEEE 1394 FireWire root contention [12,32], Zeroconf [29], IEEE 802.3 CSMA/CD [33,14] and IEEE 802.11 wireless LANs [31].
- Verification of probabilistic security protocols for anonymity (Crowds protocol [48], synchronous batching [13]), fair exchange and contract signing [42], and non-repudiation [35].
- Analysis of randomised distributed algorithms for self-stabilisation, consensus [30], Byzantine agreement [24], mutual exclusion and leader election [16].
- Evaluation of the performance, reliability and dependability of a wide range of systems, including dynamic power management schemes [41], NAND multiplexing for nanotechnology [37,38], controller systems [26], product data management systems [51,52], PC clusters, manufacturing systems and queueing systems.

Below, we select three case studies that illustrate different aspects of *quantitative* analysis with PRISM. For further information about all the examples described in this paper, and more, see the case studies section of the PRISM website [45].

6.1 Self-stabilisation algorithms

A *self-stabilising protocol* for a network of processes is a protocol which transforms a system from an *unstable* state to a *stable* state in a finite number of steps and without any outside intervention. Here we consider a class of *randomised* self-stabilising algorithms. Randomisation is often used in distributed coordination problems as a symmetry breaker, to provide simple, elegant and fast solutions. Randomised distributed algorithms can be difficult to analyse because of non-trivial interactions between the probabilistic behaviour of each process and the nondeterminism arising from concurrency between them. This makes probabilistic model checking an attractive option.

In each of the protocols we consider, the network is a ring of identical processes $P_1, \ldots, P_n$. The stable states are those where there is exactly one process designated as "privileged" (has a token). Once a stable state is reached this privilege (token) should be passed around the ring forever in a fair manner. For each of the protocols, we check that the minimum probability of reaching a stable state is 1 for all possible initial configurations and then compute both the maximum and minimum expected time (number of steps) to reach a stable state over every possible initial configuration of the protocol. Note that, to allow us to consider every possible configuration, we include, in the PRISM description of each protocol, a `init`...`endinit` statement which specifies all possible initial states.

6.1.1 *Herman's protocol*

Our first example is the algorithm of Herman [19]. The protocol operates synchronously, the ring is oriented and the number of processes in the ring must be odd. Tokens can be passed unidirectionally around the ring, and when two tokens meet they are both eliminated. At every step of the algorithm, each process with a token decides whether to keep it or pass it on based on the outcome of a random coin toss. More precisely, each process P_i in the ring has a local boolean variable x_i, and processor P_i has a token if $x_i = x_{i-1}$. In a basic step of the protocol, if the current values of x_i and x_{i-1} are equal, then processor P_i makes a (uniform) random choice as to the next value of x_i, and otherwise it sets it equal to the current value of x_{i-1}. The PRISM source code for a ring of size 3 is given in Figure 5.

We first verify the property "a stable state is reached with probability 1", expressed as $\mathtt{P}{\geq}1\ [\ \mathtt{true}\ \mathtt{U}\ \mathit{stable}\ ]$ where *stable* is the atomic proposition representing the fact that there is only one token, for example in the case of three processes *stable* is given by the expression:

$$(\mathit{x3}{=}\mathit{x1}?1{:}0){+}(\mathit{x1}{=}\mathit{x2}?1{:}0){+}(\mathit{x2}{=}\mathit{x3}?1{:}0){=}1\,.$$

Secondly, since we assign a cost of one unit to each step of the algorithm (see Figure 5), PRISM can be used to compute "the expected time (number of steps) for self-stabilisation to complete", expressed as $\mathtt{R}=?[\ \mathtt{F}\ \mathit{stable}]$. More precisely, we compute the worst case and best case expected times for all initial configurations with K tokens, for different values of K, which, in the case of three processes, is expressed by the specifications:

$$\mathtt{R}{=}?\ [\ \mathtt{F}\ \mathit{stable}\ \{(\mathit{x3}{=}\mathit{x1}?1{:}0){+}(\mathit{x1}{=}\mathit{x2}?1{:}0){+}(\mathit{x2}{=}\mathit{x3}?1{:}0){=}K\}\{\mathtt{max}\}\ ]$$

$$\mathtt{R}{=}?\ [\ \mathtt{F}\ \mathit{stable}\ \{(\mathit{x3}{=}\mathit{x1}?1{:}0){+}(\mathit{x1}{=}\mathit{x2}?1{:}0){+}(\mathit{x2}{=}\mathit{x3}?1{:}0){=}K\}\{\mathtt{min}\}\ ]$$

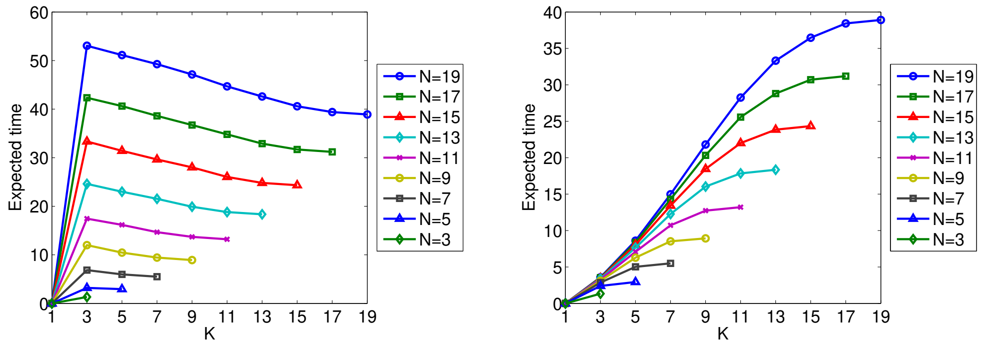
In Figure 6 we present the results obtained with PRISM when verifying these specifications for a range of numbers of processes (N) and a range of values of K.

This PRISM case study illustrates an unproven conjecture from [36] that the worst case execution time for this algorithm always results from the case where there are initially three tokens. The results also show that the minimum and maximum expected times respectively increase and decrease as

```
// Herman's self-stabilising algorithm [Her90]
// gxn/dxp 13/07/02

// the protocol is synchronous with no non-determinism (a DTMC)
probabilistic

// module for process 1
module process1

    // bits in the ring (initially all the same i.e. a token in every place)
    x1 : [0..1];

    [step] x1 = x3 → 0.5 : (x1' = 0) + 0.5 : (x1' = 1);
    [step] x1! = x3 → (x1' = x3);

endmodule

// add further processes through renaming
module process2 = process1[x1 = x2, x3 = x1 ] endmodule
module process3 = process1[x1 = x3, x3 = x2 ] endmodule

// cost - 1 for each transition (expected steps)
rewards

    [] true : 1;

endrewards

// initial states (at least one token i.e. all states)
init

    true

endinit
```

Fig. 5. PRISM language description of Herman's self-stabilisation algorithm (3 processes)

one increases the value of K. Furthermore, we can make a comparison with simulation-based techniques. Model checking with PRISM involves a single DTMC model with multiple initial states, for which PRISM executes a single analysis. On the other hand, simulation, for example, would have to be performed separately for each initial state (for example in the case when $N = 19$ the are approximately half a million possible configurations to consider) to obtain comparable results.



(a) Maximum expected time (b) Minimum expected time

Fig. 6. Expected time results for Herman's self-stabilisation algorithm.

6.1.2 Israeli and Jalfon protocol

This protocol originates from [22]. It operates asynchronously with an arbitrary scheduler, the ring is oriented and communication is bidirectional in the ring. Each process has a boolean variable q_i which represents the fact that a token is in place i. A process is active if it has a token and only active processes can be scheduled. When an active process is scheduled, it makes a (uniform) random choice as to whether to move the token to its left or right. As before, tokens colliding are merged into a single one.

We model the protocol in PRISM, which yields an MDP model (due to the fact that the protocol is asynchronous). In Figure 7 we demonstrate the outcome of model checking the minimum and maximum expected times to reach a stable state given that the initial number of tokens equals K, as K varies. In the case of three processes, this corresponds to checking the following specifications (since this model is an MDP we use `Rmax` and `Rmin` as opposed to `R`):

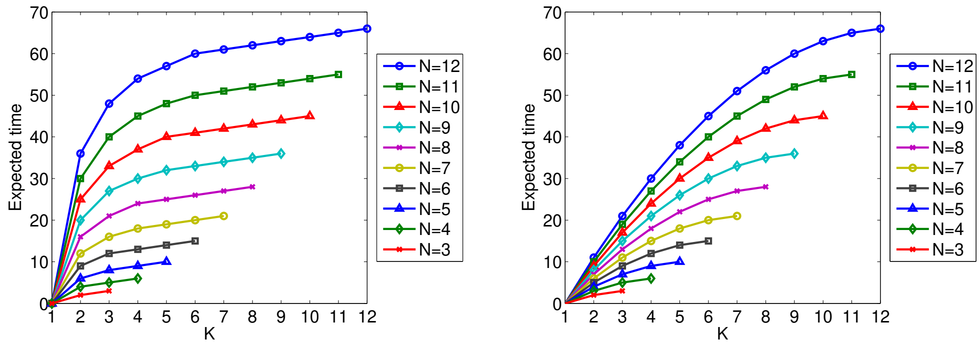
$$\texttt{Rmax}{=}?\ [\ \texttt{F}\ (q1{+}q2{+}q3{=}1)\ \{q1{+}q2{+}q3{=}K\}\{\texttt{max}\}\ ]$$

$$\texttt{Rmin}{=}?\ [\ \texttt{F}\ (q1{+}q2{+}q3{=}1)\ \{q1{+}q2{+}q3{=}K\}\{\texttt{min}\}\ ]$$

Note that this algorithm does not exhibit the trend observed for the Herman ring, i.e. both the minimum and maximum expected times increase as we increase K.

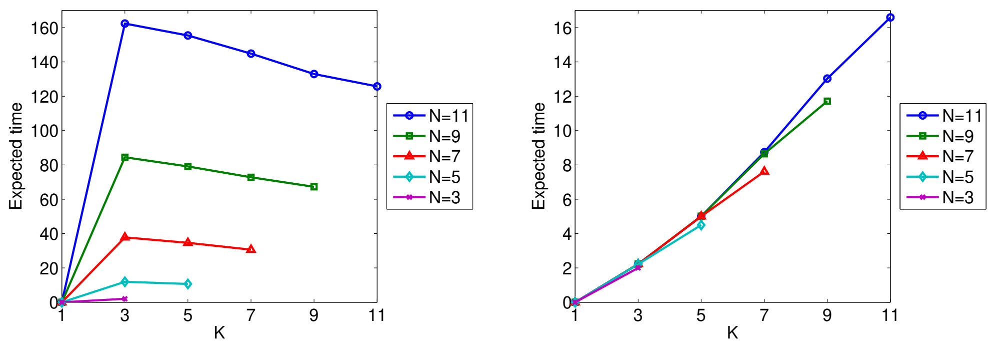
6.1.3 Beauquier, Gradinariu and Johnen protocol

Finally, we consider the self-stabilising protocol of [7] for an arbitrary scheduler. It operates asynchronously, the ring is oriented, communication is unidirectional in the ring, and the number processes in the ring must be odd. Each process has two boolean variables, d_i and p_i, where if $d_i = d_{i-1}$, process i is said to have a deterministic token; and if $p_i = p_{i-1}$ process i is said to have a probabilistic token. The stable states are those where there is only one probabilistic token. A process is active if it has a deterministic token and only active processes can be scheduled. When an active process is scheduled, it



(a) Maximum expected time (b) Minimum expected time

Fig. 7. Expected time results for Israeli and Jalfon's self-stabilisation algorithm.



(a) Maximum expected time (b) Minimum expected time

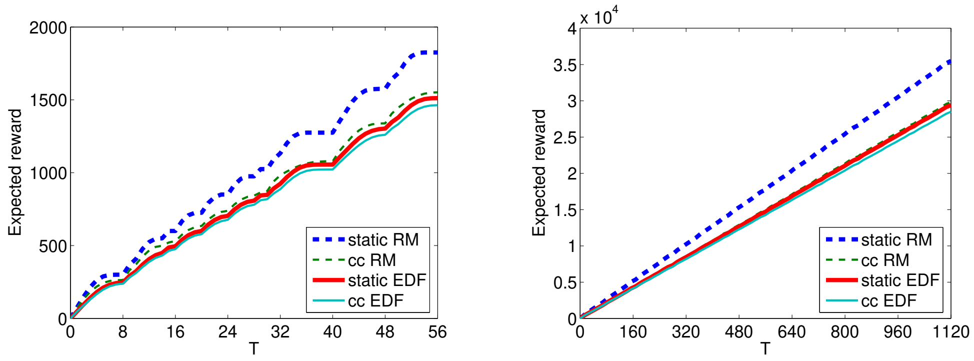
Fig. 8. Expected time results for Beauquier, Gradinariu and Johnen's self-stabilisation algorithm.

sets d_i to be the negation of its current value (passes the deterministic token) and, if it also has a probabilistic token, it makes a (uniform) random choice as to the next value of p_i (randomly selects whether to pass the probabilistic token or not).

Figure 8 shows the results obtained with PRISM when computing the worst- and best-case expected time until a stable state is reached as K and N varies. In the results for this model, we observe the trend as in Herman's ring, namely that the configurations which achieve the maximum expected time are those where only three processes have probabilistic tokens (compare Figure 6 and Figure 8).

6.2 Dynamic voltage scaling

Our next example concerns *power management*, an area that has become extremely relevant because of the need to preserve battery life and hence power efficiency. Here, we consider a technique called *dynamic voltage scaling*, used in real-time embedded systems to achieve a compromise between battery life and performance. The technique is used to schedule a number of tasks



(a) Small time scale

(b) Large time scale

Fig. 9. Expected energy consumption for four different dynamic voltage scaling scheduling schemes over time.

which must be executed periodically. Each task has an associated period and a worst-case execution time. The voltage of the system can also be varied during scheduling, which has the effect of reducing the power consumption of the system. This will, however, slow down the execution of the current task. The aim is to schedule tasks and voltage changes in such a way that power consumption is minimised whilst ensuring that all tasks are executed within their deadlines.

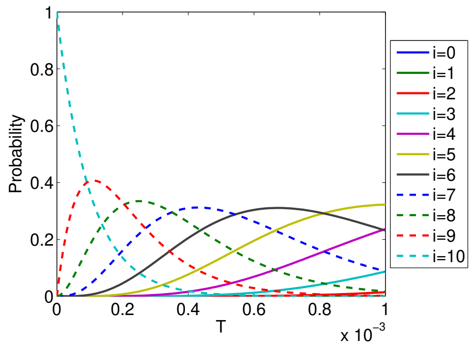
We have modelled and analysed in PRISM the performance of several scheduling schemes from [44]. The need for probability arises because the actual execution time of each task is random (only a worst-case figure is known). Nondeterminism also has to be modelled, to represent the fact that it is sometimes unspecified which task a scheduling scheme will pick. Thus, the derived model is an MDP, and hence we examine the worst-case behaviour of *any* implementation of each algorithm.

Figure 9 shows a comparison of "the maximum expected energy consumed by a given time bound" for four scheduling schemes (see [45] for more details). The actual cost measured is the square of the system's voltage, which is proportional to the energy consumed. The comparisons match those observed in [44], obtained through simulation.

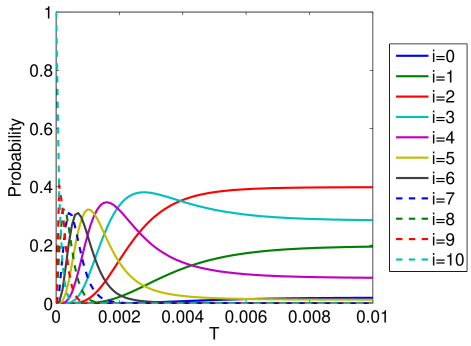
Another probabilistic model checking case study in this area can be found in [39,40], which studies stochastic dynamic power management strategies. Here, a wide range of properties can be analysed, e.g.: "the expected number of jobs awaiting service at time T", "the probability that 50 job requests have been lost by time T" and "the expected long-run power consumption".

6.3 Biological process modelling

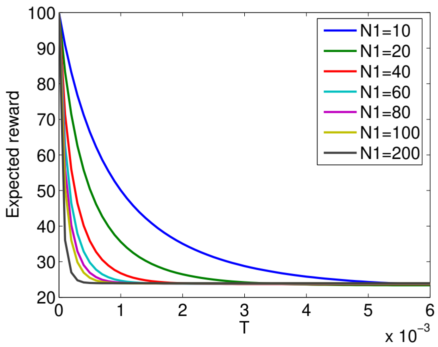
Our last example comes from a new area of applications for probabilistic model checking – biological processes. It is well known that the time until a reaction occurs between two molecules can be adequately modelled as an exponential distribution, and therefore CTMC models are appropriate. At the same



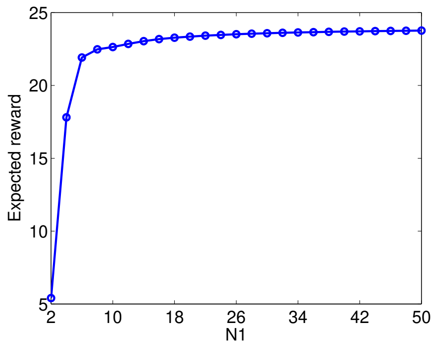
(a) Probability i Na molecules at time T (small time scale)



(b) Probability i Na molecules at time T (large time scale)



(c) Expected percentage of Na molecules at time T



(d) Long run average percentage of Na molecules

Fig. 10. Model checking results for the biochemical reaction case study

time, molecular interactions can be described using process algebra. In [47], the stochastic pi-calculus was used to model a number of biochemical reactions which were then analysed through simulation. Here we use PRISM to model the same reactions and analyse the models through probabilistic model checking.

We construct a CTMC model of the system where the states of the CTMC correspond to the number of molecules of each type and the transitions correspond to the possible reactions between the molecules. The rate of a reaction is determined by a base rate and the concentrations of the reactants (i.e. the number of each type of molecule that takes part in the reaction).

We consider the ionic reaction:

$$\mathrm{Na} + \mathrm{Cl} \longleftrightarrow \mathrm{Na}^{+} + \mathrm{Cl}^{-}$$

involving the oxidation of sodium (Na) and reduction of chlorine (Cl). In our experiments we suppose that initially the number of Na molecules and Cl molecules equals N (and there are no Na^+ or Cl^- molecules).

The first property we consider is the probability that the number of Na molecules at time T equals i for $i = 0, \ldots, N$. This property is specified by the formula

$$\texttt{P}{=}?[\texttt{true}\ \texttt{U}[T,T]\ na{=}i]\,.$$

Figures 10(a) and 10(b) plot, for the case when $N = 10$, these probabilities as the value of T varies.

The second property we consider is the expected percentage of Na molecules at time T. This property is specified by the CSL formula $\texttt{R} =?[\texttt{I} = T]$ where the reward in a state is the percentage of Na molecules, i.e. if na is the variable denoting the number of Na molecules, then the reward in each state equals $(100 \cdot na)/N$. Figure 10(c) presents the results, for a range of values of N, obtained with PRISM when verify this formula as the value of T varies.

Finally we consider the expected long-run percentage of Na molecules. This property is specified by the CSL formula $\texttt{R} =?[\texttt{S}]$. where the reward in a state is again the percentage of Na molecules. Figure 10(d) presents these expected values as N varies.

Biological systems have been expressed in the process algebra PEPA [9] and analysed using probabilistic model checking in PRISM in [10].

We also observe that conventional temporal model checking can be employed to analyse biological systems, for example to consider the possibility and impossibility of certain temporal relationships between events.

7 Conclusion

We have given a high-level overview of probabilistic model checking with PRISM, the software tool developed at the University of Birmingham. Many researchers have participated in the development of probabilistic model checking techniques and PRISM, as well as performing case studies, see the "People" link at [45]; their contributions are gratefully acknowledged. We have demonstrated the usefulness of probabilistic model checking in domains as wide-ranging as performance analysis, reliability and biology, focusing in particular on the quantitative analysis which allows one to obtain exact best-, worst- and average-case system characteristics. Of the 30 or so case studies that were modelled, six contained flaws.

In comparison with simulation, probabilistic model checking has a number of advantages; it is exhaustive, good for 'corner cases' and analysis such as 'for all possible initial states' or 'for all schedules'. On the other hand, simulation is more amenable to more complex stochastic scenarios such as those featuring general distributions. State-space explosion is the main limitation of probabilistic model checking, and techniques for abstraction, model reduction and compositional reasoning are subject of active research.

Currently, PRISM models are finite-state and have to be manually described in the modelling language, as opposed to being automatically extracted from source code. We are extending PRISM in a number of directions, for example with real-time [33], simulation-based approximate analysis [54,14], parallelisation [55], and biology applications.

References

[1] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[3] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conference on Computer-Aided Design (ICCAD'93)*, pages 188–191, 1993. Also available in *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[5] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In J. Baeten and S. Mauw, editors, *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 146–161. Springer, 1999.

[6] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[7] J. Beauquier, M. Gradinariu, and C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 199–208, 1999.

[8] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

[9] M. Calder, S. Gilmore, and J. Hillston. Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. In *Proc. Workshop on Concurrent Models in Molecular Biology (BIOCONCUR'04)*, ENTCS. Elsevier, 2004. To appear.

[10] M. Calder, V. Vyshemirsky, D. Gilbert, and R. Orton. Analysis of signalling pathways using the PRISM model checker. In G. Plotkin, editor, *Proc. Computational Methods in Systems Biology (CMSB'05)*, 2005.

[11] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*,

pages 1–15, 1993. Also available in *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[12] C. Daws, M. Kwiatkowska, and G. Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):221–236, 2004.

[13] R. Dingledine, V. Shmatikov, and P. Syverson. Synchronous batching: From cascades to free routes. In *Proc. 4th Workshop on Privacy Enhancing Technologies (PET'04)*, 2004.

[14] M. Duflot, L. Fribourg, T. Hérault, R. Lassaigne, F. Magniette, S. Messika, S. Peyronnet, and C. Picaronny. Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC. In *Proc. 4th Workshop on Automated Verification of Critical Systems (AVoCS'04)*, volume 128(6) of *Electronic Notes in Theoretical Computer Science*, pages 195–214. Elsevier Science, 2004.

[15] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. A formal analysis of Bluetooth device discovery. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, 2004. To appear.

[16] W. Fokkink and J. Pang. Simplifying Itai-Rodeh leader election for anonymous rings. In *Proc. 4th Workshop on Automated Verification of Critical Systems (AVoCS'04)*, volume 128(6) of *Electronic Notes in Theoretical Computer Science*, pages 53–68. Elsevier Science, 2004.

[17] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[18] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5(3):356–380, 1983.

[19] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[20] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In S. Graf and M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 347–362. Springer, 2000.

[21] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[22] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizating mutual exclusion. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.

[23] B. Jeannet, P. D'Argenio, and K. Larsen. Rapture: A tool for verifying Markov decision processes. In I. Cerna, editor, *Proc. Tools Day, affiliated to 13th Int. Conf. Concurrency Theory (CONCUR'02)*, Technical Report FIMU-RS-2002-05, Faculty of Informatics, Masaryk University, pages 84–98, 2002.

[24] M. Kwiatkowska and G. Norman. Verifying randomized Byzantine agreement. In D. Peled and M. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *LNCS*, pages 194–209. Springer, 2002.

[25] M. Kwiatkowska, G. Norman, and D. Parker. PRISM users' guide. Available from `www.cs.bham.ac.uk/~dxp/prism`.

[26] M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. In *Proc. 11th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'04)*, 2004.

[27] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 322–323. IEEE Computer Society Press, 2004.

[28] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[29] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. In K. Larsen and P. Niebert, editors, *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 105–120. Springer-Verlag, 2003.

[30] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 194–206. Springer, 2001.

[31] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In H. Hermanns and R. Segala, editors, *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, volume 2399 of *LNCS*, pages 169–187. Springer, 2002.

[32] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.

[33] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. In Y. Lakhnech and S. Yovine, editors, *Joint Conference on Formal Modelling and Analysis of Timed Systems (FORMATS) and Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT)*, volume 3253 of *LNCS*, pages 293–308. Springer, 2004.

[34] M. Kwiatkowska, D. Parker, Y. Zhang, and R. Mehmood. Dual-processor parallelisation of symbolic probabilistic model checking. In D. DeGroot and P. Harrison, editors, *Proc. 12th International Symposium on Modeling,*

Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04), pages 123–130. IEEE Computer Society Press, 2004.

[35] R. Lanotte, A. Maggiolo-Schettini, and A. Troina. Automatic analysis of a non-repudiation protocol. In *Proc. 2nd International Workshop on Quantitative Aspects of Programming Languages (QAPL'04)*, 2004.

[36] A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems.* Springer, 2004.

[37] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla. Evaluating the reliability of defect-tolerant architectures for nanotechnology with probabilistic model checking. In *Proc. International Conference on VLSI Design (VSLI'04)*, pages 907–914. IEEE Computer Society Press, 2004.

[38] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla. Evaluating the reliability of NAND multiplexing with PRISM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1629–1637, 2005.

[39] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Formal analysis and validation of continuous time Markov chain based system level power management strategies. In W. Rosenstiel, editor, *Proc. 7th Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02)*, pages 45–50. IEEE Computer Society Press, 2002.

[40] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Proc. 3rd Workshop on Automated Verification of Critical Systems (AVoCS'03)*, Technical Report DSSE-TR-2003-2, University of Southampton, pages 202–215, April 2003.

[41] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17(2):160–176, 2005.

[42] G. Norman and V. Shmatikov. Analysis of probabilistic contract signing. In A. Abdallah, P. Ryan, and S. Schneider, editors, *Proc. BCS-FACS Formal Aspects of Security (FASec'02)*, volume 2629 of *LNCS*, pages 81–96. Springer, 2003.

[43] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.

[44] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-powered embedded operating systems. *Operating Systems Review*, 35(5):89–102, 2001.

[45] PRISM web site. `www.cs.bham.ac.uk/~dxp/prism`.

[46] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.), volume 23 of *CRM Monograph Series.* American Mathematical Society, 2004.

[47] E. Shapiro. Biomolecular processes as concurrent computation. Course at the Weizmann Institute of Science, Israel, 2001.

[48] V. Shmatikov. Probabilistic model checking of an anonymity system. *Journal of Computer Security*, 12(3/4):355–377, 2004.

[49] F. Somenzi. CUDD: Colorado University decision diagram package. Public software, Colorado Univeristy, Boulder, `http://vlsi.colorado.edu/~fabio`, 1997.

[50] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton, 1994.

[51] M. ter Beek, M. Massink, and D. Latella. Towards model checking stochastic aspects of the thinkteam user interface. In *Proc. 12th International Workshop on Design, Specification and Verification of Interactive Systems (DSVIS'05).* Springer, 2005. To appear.

[52] M. ter Beek, M. Massink, and D. Latella. Towards model checking stochastic aspects of the thinkteam user interface - full version. Technical Report 2005-TR-18, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2005.

[53] M. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 327–338. IEEE Computer Society Press, 1985.

[54] H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005. To appear.

[55] Y. Zhang, D. Parker, and M. Kwiatkowska. A wavefront parallelisation of CTMC solution using MTBDDs. In *Proc. International Conference on Dependable Systems and Networks (DSN'05)*, pages 732–742. IEEE Computer Society Press, 2005.