PRISM: probabilistic model checking for performance and reliability analysis

Marta Kwiatkowska, Gethin Norman, David Parker

PRISM: Probabilistic Model Checking for Performance and Reliability Analysis

Marta Kwiatkowska, Gethin Norman and David Parker

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD

{marta.kwiatkowska,gethin.norman,david.parker}@comlab.ox.ac.uk

ABSTRACT

Probabilistic model checking is a formal verification technique for the modelling and analysis of stochastic systems. It has proved to be useful for studying a wide range of quantitative properties of models taken from many different application domains. This includes, for example, performance and reliability properties of computer and communication systems. In this paper, we give an overview of the probabilistic model checking tool PRISM, focusing in particular on its support for continuous-time Markov chains and Markov reward models, and how these can be used to analyse performability properties.

1. INTRODUCTION

Formal verification techniques, and in particular model checking [8], offer a powerful and rigorous approach for establishing the correctness of complex systems. Improvements in the efficiency and usability of this technology mean that it us now applied in the design phase of a wide range of computerised systems, from microchips to device drivers.

Probabilistic model checking is a generalisation of these techniques, aimed at systems whose behaviour is stochastic in nature. This arises naturally in many situations, such as the unreliable or unpredictable behaviour exhibited by computer networks and communication systems or through the use of randomisation, e.g. in distributed protocols. Probabilistic model checking is based on the construction and analysis of a probabilistic model, typically a Markov chain or Markov process. In this paper, we focus on the use of continuous-time Markov chains and Markov reward models, which are also widely used in well-established model-based performance evaluation techniques.

Probabilistic model checking requires two inputs:

- a description of the system to be analysed, typically given in some high-level modelling language;
- a formal specification of quantitative properties of the system that are to be analysed, usually expressed in variants of temporal logic.

From the first of these inputs, a *probabilistic model checker* constructs the corresponding probabilistic model. This is a probabilistic variant of a state-transition system: each *state* represents a possible configuration of the system being modelled; and each *transition* represents a possible evolution of the system from one configuration to another over time. Transitions are labelled with quantitative information regarding the probability and/or timing of the transition's occurrence. In the case of continuous-time Markov chain, transitions are assigned *rates*: positive, real values that are interpreted as the rates of negative exponential distributions. Markov chains can also be augmented with *rewards*, used to specify additional quantitative measures of interest. For more detailed information about these models, see for example [17, 10].

The power of probabilistic model checking comes from the fact that these models are constructed in an exhaustive fashion, based on a systematic exploration of all possible states that can occur. Once this model has been constructed, it can be used to analyse a wide range of quantitative properties of the original system, relating for example to its performance or reliability. In contrast to, say, discrete-event simulation techniques, which generate approximate results by averaging results from a large number of random samples, probabilistic model checking applies numerical computation to yield exact results.

PRISM [16] is an open-source probabilistic model checker developed initially at the University of Birmingham and now at the University of Oxford. It provides support for building and analysing several types of probabilistic models: discrete- and continuous-time Markov chains, Markov decision processes, and extensions of these models with rewards. This paper provides an overview of PRISM and, in particular, how it can be applied to the analysis of performance, dependability and performability properties of computer and communication systems.

Paper outline. The remainder of the paper is structured as follows. In the next section, we provide a brief introduction to the PRISM modelling language, which is used to specify models for the tool. In Section 3, we discuss the PRISM property specification language, and give a large number of examples of its use. Section 4 provides further information about the tool itself: its functionality and the underlying algorithms and techniques used. Sections 5 and 6 conclude the paper with details about obtaining the tool and access to further resources and information.

2. MODEL SPECIFICATION

A variety of formalisms have been developed for specifying probabilistic models. These include stochastic variants of Petri nets and process algebras, stochastic activity networks and many others. PRISM provides a simple, textual modelling language, based on the Reactive Modules formalism

```
// Component failure rates
// MTTFs: 1 month, 1 year, 1 day
const double λs = 1/(30 · 24 · 60 · 60);
const double λp = 1/(365 · 24 · 60 · 60);
const double δf = 1/(24 · 60 · 60);
// Rate for processor reboot
const double δr = 1/30;

module sensors

    // Number of sensors operational
    s : [0..3] init 3;

    // Failure of a single sensor
    [] s > 0 → s · λs : (s′ = s − 1);

endmodule

module input_processor

    // State: 2=ok, 1=transient fault, 0=failed
    i : [0..2] init 2;

    // Failure of processor
    [] i > 0 → λp : (i′ = 0);
    // Transient fault
    [] i = 2 → δf : (i′ = 1);
    // Reboot after transient fault
    [input_reboot] i = 1 → δr : (i′ = 2);

endmodule
```

Figure 1: Part of the PRISM modelling language description from a simple reliability case study.

[1], in which models to be analysed by the tool are described. In this section, we provide a brief introduction to this modelling language. It provides a uniform way of describing models of all the types supported by the tool: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes. Here we focus on continuous-time Markov chains and, through the addition of information about rewards, Markov reward models.

A PRISM model comprises a set of *modules* which represent different components of the system being modelled. The state of each module is represented by a set of finite-ranging *variables*. The global state of the model at any point in time is determined by the values of all such module variables and, optionally, a set of global variables.

Figure 1 shows an example of the PRISM modelling language: a fragment taken from the model description of a simple reliability case study [12] (based on a model from [14]). The model includes a set of sensors and actuators, monitored and controlled by an input and output processor, respectively, and a main processor which communicates between the other components. The model fragment in Figure 1 shows modules representing the sensors and input processor. As can be seen from the comments labelling the code, the state of the sensors is simply an integer value (between 0 and 3) representing the number of sensors currently operational and the state of the processor is a value indicating whether it is operational or has suffered either either a transient or permanent fault.

The behaviour of a module, i.e. the changes to its state

```
// Total elapsed time
rewards "time"
    true : 1;
endrewards

// Time operational
rewards "oper"
    (s ≥ 2 & i = 0) : 1;
endrewards

// Number of sensors currently operational
rewards "num_sensors"
    true : s;
endrewards

// Number of input processor reboots
rewards "num_reboots"
    [input_reboot] true : 1;
endrewards
```

Figure 2: A selection of reward structures for use with the PRISM model illustrated in Figure 1.

that can occur, is specified by a set of *guarded commands*. These take the form:

$$[act]\ guard\ \rightarrow\ rate\ :\ update;$$

where *act* is an (optional) action label, *guard* is a predicate over the variables of the model, *rate* is a (non-negative) real-valued expression and *update* is of the form:

$$(x_1'{=}u_1)\ \&\ (x_2'{=}u_2)\ \&\ \ldots\ \&\ (x_k'{=}u_k)$$

where $x_1, x_2, \ldots, x_k$ are local variables of the module and $u_1, u_2, \ldots, u_k$ are expressions over all variables.

Intuitively, a command is enabled in a global state of the PRISM model if the state satisfies the predicate *guard*. If a command is enabled, a transition that updates the module's variables according to *update* can occur with rate *rate*. When multiple commands with the same update are enabled, the corresponding transitions are combined into a single transition whose rate is the sum of the individual rates.

The first command of module *sensors* in Figure 1, for example, describes the changes that occur when a single sensor fails. This can happen when at least one of the sensors is currently operational ($s > 0$) and the resulting change in state is a decrement in the counter ($s' = s - 1$). The rate of a failure occurring is proportional to the number of sensors that can fail: it is given as $s \cdot \lambda_s$, where λ_s is a constant defined earlier in the model, representing the failure rate of a single sensor.

Interactions between multiple modules, i.e. simultaneous changes in their state, are modelled using *synchronisation*, which is specified by augmenting guarded commands with action labels. The last command in Figure 1, for example, is labelled with *input_reboot*. This is because the rebooting of the input processor is initiated by a communication from a separate processor (not shown). The rate of a synchronous transition is defined as the product of the rates for each component command. PRISM also includes the ability to specify more precisely how modules synchronise using process-algebraic operators.

Finally, we discuss the addition of *rewards* to a model, which are used to specify additional quantitative measures

of interest. At the level of a Markov chain, these are simply labellings of states and transitions with real values, referred to in PRISM as state rewards and transitions rewards, respectively. Figure 2 illustrates the specification of rewards in the PRISM modelling language, for the example model from Figure 1. PRISM allows multiple, named *reward structures* each defining a labelling of state and/or transition rewards.

State rewards typically represent the *rate* at which reward is accumulated; the first two reward structures in Figure 2, for example, associate a state reward of 1 to either all states or all states in which the system is operational. These can be used to reason about the total elapsed time and the time spent operational, respectively. The third reward structure, however, shows that state rewards can also represent an *instantaneous* measure of interest, at a particular moment in time. The final example in Figure 2 specifies transition rewards (sometimes called impulse rewards), which are accumulated when a transition between states occurs. This example associates a reward of 1 to transitions labelled with the action *input_reboot* from any state of the model, and could be used to count the number of input processor reboots that occur in a given time frame.

For further information about the PRISM modelling language, see the manual and case study repository [16].

3. PROPERTY SPECIFICATION

In order to analyse a PRISM model, it is necessary to specify one or more properties. PRISM's property specification language is based on *temporal logic*, which offers a flexible and unambiguous means of describing a wide range of properties. In particular, the language incorporates operators from PCTL [9], CSL [2, 6] and some of its extensions [4]. These logics have already been shown able to express a wide range of performance, dependability and performability properties. Other logics, such as CTL and LTL, are currently also being incorporated into the property specification language.

PRISM puts particular emphasis on *quantitative* properties. For example, PCTL (and CSL) allow expression of logical statements such as "the probability of eventual system failure is less than p", denoted $\mathrm{P}_{<p}$ [F *fail*]. In PRISM, it is more typical to simply ask "what is the probability of eventual system failure?", expressed as $\mathrm{P}_{=?}$ [F *fail*]. The property specification language also allows numerical values such as these to be combined in arithmetic expressions, allowing more complex measures to be expressed.

The key constructs in the PRISM property specification language are the P, S and R operators. The P operator refers to the probability of an event occurring (more precisely, the probability that the observed execution of the model satisfies a given specification). The S operator is used to reason about the steady-state probabilities of the model. Finally, the R operator is used to express properties that relate to rewards (more precisely, the expected value of a random variable, associated with particular reward structure).

For a presentation of the formal semantics of the PRISM property specification language, see for example [13]. In the remainder of this section, we give an overview of the different types of property that are available and examples of the kinds of performability measures that they can be used to express.

Transient and steady-state probabilities. The probability that the system is in a particular state of interest, either at a specific time instant (transient) or in the long-run (steady-state) can be expressed using the P and S operators, respectively. Consider, for example, a service-providing system whose state can be classified as either "operational" or "failed" and assume that *oper* is a Boolean variable whose value is true if system is operational. Alternatively, *oper* could be a more complex expression over state variables expressing this fact. The following properties describe the availability of the system:

- $\mathrm{P}_{=?}$ [$\mathrm{F}^{[t,t]}$ *oper*] - "the *instantaneous availability* of the system, i.e. the probability that it is operational at time instant t";
- $\mathrm{S}_{=?}$ [*oper*] - "the *long-run availability* of the system, i.e. the steady-state probability that it is operational".

Timing, occurrence and ordering of events. The P operator in the example above is used with the $\mathrm{F}^{[t,t]}$ operator to refer to a single instant in time. This is a special case of the *time-bounded* operator F^{I} where I is a time interval $[t_1, t_2] \in \mathbb{R}$. Also permitted is the *unbounded* variant F, equivalent to $\mathrm{F}^{[0,\infty)}$. These allow us to reason about the probability of an event occurring either within in a particular period of time, or at any point in the lifetime of the system. Consider a system comprising two components, A and B, each of which can fail independently. We can express:

- $\mathrm{P}_{=?}$ [$\mathrm{F}^{[0,600]}$ $fail_A$] - "the probability that component A fails within 10 minutes";
- $\mathrm{P}_{=?}$ [F $fail_A$| $fail_B$] - "the probability that either component A or B fails at some point".

There are also several other operators that can be used with the P operator. One example is G, which can be seen as the dual of F: it expresses the fact that a condition remains, rather than becomes, true. Like F, it has both timed and untimed variants. The former gives, for example:

- $\mathrm{P}_{=?}$ [$\mathrm{G}^{[0,3600]}$!($fail_A$| $fail_B$)] - "the probability of no failures occurring in the first hour".

Another related operator is U which, for two given conditions, states that the first remains true *until* the second becomes true. Examples of its use include:

- $\mathrm{P}_{=?}$ [!$fail_B$ $\mathrm{U}^{[3600,7200]}$ $fail_B$] - "the probability of component B failing *for the first time* during the second hour of operation";
- $\mathrm{P}_{=?}$ [!$fail_A$ U $fail_B$] - "the probability that component B fails *before* component A".

To be more precise, the latter property gives the probability that component B eventually fails *and* no failure of component A occurs before this point. An alternative is the *weak* form of the operator, denoted W, for which the first part of this is not required:

- $\mathrm{P}_{=?}$ [!$fail_A$ W $fail_B$] - "the probability that a failure of component B (if it occurs) happens before any failure of component A, i.e. the probability that either B fails before A or neither ever fail".

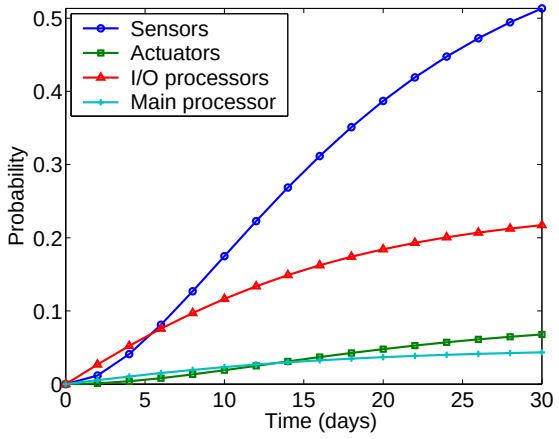


Figure 3: Example results from a PRISM experiment: transient probability of component failures

Reward-based properties. The PRISM property specification language also includes an R operator which is used to refer to the *expected value* of a random variable defined in terms of a reward structure. As we show below, R, like P, can be combined with a variety of operators. Since a model will often be decorated with multiple reward structures, we augment the R operator with a label: properties of the form $\mathrm{R}_{\{\text{“}rew\text{”}\}=?}$ [...] refer to the expected value of reward structure *rew*.

Consider, as in our first set of examples above, a system which provides a service when it is operational. We assume a reward structure *oper* that assigns a state reward of 1 to all states of the model in which the system is operational (and 0 to all others). By reasoning about the amount of reward accumulated over a period of time, we can represent:

- $\mathrm{R}_{\{\text{“}oper\text{”}\}=?}$ [$\mathrm{C}^{\leq t}$] - “the expected *cumulative operational time* of the system in the time interval $[0, t]$”.

Another possibility is to consider the reward accumulated until a given event occurs. Assume that we have a simple reward structure *time*, which assigns a state reward of 1 to every state, and a Boolean state variable *fail*, which is true when a system failure has occurred. We have:

- $\mathrm{R}_{\{\text{“}time\text{”}\}=?}$ [F *fail*] - “the *mean-time-to-failure* of the system, i.e. the expected amount of time that elapses before the first failure occurs”.

A third operator to reason about cumulative rewards is the S operator, which gives the long-run expected rate of reward accumulation. For example, consider a model of a queue storing jobs to be processed by a server. Assume a reward structure *proc* that assigns a transition reward of 1 to every transition corresponding to the a job being processed by the server. Then we can use:

- $\mathrm{R}_{\{\text{“}proc\text{”}\}=?}$ [S] - “the *throughput* of the system, i.e. the expected steady-state rate of job completion”.

Finally we consider a fourth operator for R which deals, not with accumulation, but the *instantaneous* value of rewards. In this context, we deal only with state rewards, but these do not need to represent a *rate* at which rewards accumulate: they can describe any measure of interest that can be defined in terms of the state variables of the model. Consider, as above, a model of a job queue and a reward structure *size* which assigns a state reward to each state equal to the number of jobs in the queue at that point. We can can then compute:

- $\mathrm{R}_{\{\text{“}size\text{”}\}=?}$ [$\mathrm{I}^{=t}$] - “the expected number of jobs waiting in the queue at time t”;

Best- and worst-case scenarios. Because model checking is exhaustive and computes exact answers, values are usually generated for *all* states of a model. For example, when model checking $\mathrm{P}_{=?}$ [F *fail*], PRISM computes the probability of reaching a state in which *fail* is true, starting from any state of the model. The PRISM property specification language includes a feature called *filters*, which computes the minimum or maximum value of a property over a range of states. Examples of its use are as follows:

- $\mathrm{R}_{\{\text{“}time\text{”}\}=?}$ [F *oper* {“*fail*”}{*max*}] - “the *worst-case* mean time required for system repair, from any possible failure state of the system”;
- $\mathrm{P}_{=?}$ [$\mathrm{F}^{[t,t]}$ *oper* {“*init*”}{*max*}] - “the *best-case* instantaneous availability of the system at time t, starting from any initial configuration”.

Arithmetic expressions. Since the properties we have presented here, using operators of the form $\mathrm{P}_{=?}$, $\mathrm{R}_{=?}$ and $\mathrm{S}_{=?}$, return numerical values, it is natural to also allow these to be built into more complex expressions. This extends further the range of properties expressible in PRISM. Examples include:

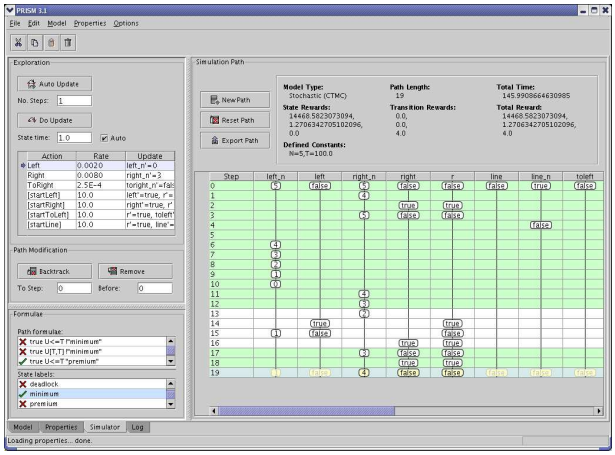
- $1 - \mathrm{P}_{=?}$ [$\mathrm{F}^{[3600,7200]}$ *oper*] - “the probability that the system is *not* operational at any point during the second hour of operation”.
- $\mathrm{R}_{\{\text{“}oper\text{”}\}=?}$ [$\mathrm{C}^{\leq t}$] / t - “the expected *interval availability* of the system in the time interval $[0, t]$, i.e. the fraction of that time which it is available”.
- $\mathrm{P}_{=?}$ [F $fail_A$] / $\mathrm{P}_{=?}$ [F *any_fail*] - “the (conditional) probability that component A eventually fails, *given that* at least one component fails”.

Another use of arithmetic expressions is to compute statistical properties such as variance or standard deviation. Consider the reward structure *size* used earlier to represent the size of a job queue. If we define a second reward structure *size_sq*, which assigns the square of this value to each state, we can define:

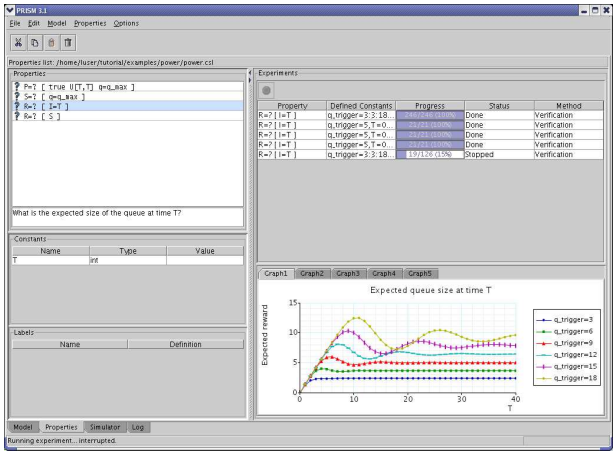
- $\mathrm{R}_{\{\text{“}size\_sq\text{”}\}=?}$ [$\mathrm{I}^{=t}$] $-$ *pow*($\mathrm{R}_{\{\text{“}size\text{”}\}=?}$ [$\mathrm{I}^{=t}$], 2) - “the *variance* of the jobs queue size at time t”.

4. THE PRISM TOOL

In simple terms, the purpose of the PRISM tool is to construct a probabilistic model, from high-level descriptions of the form described in Section 2, and then analyse one or more properties such as those discussed in Section 3. In the next section, we summarise the functionality of the PRISM tool which is provided to achieve this. In Section 4.2, we give a brief outline of the techniques that are implemented in the tool.



(a) Model exploration with the simulator



(b) Experiment generation and graph plotting

Figure 4: Screenshots of the PRISM graphical user interface

4.1 Functionality

User interfaces. All of the functionality of PRISM is available from either a command-line version of the tool or through its graphical user interface. The former is useful for running lengthy or process-intensive jobs, executing large batches of model checking runs or for combining PRISM with other tools through scripting. The graphical user interface (GUI) provides a more intuitive entry-point for newcomers to the tools, as well as invaluable features for more experienced users. The GUI provides:

- a model editor for the PRISM modelling language, with syntax highlighting and error reporting;
- an editor for PRISM properties;
- a simulator tool for exploring and debugging PRISM models (see Figure 4(a));
- graph-plotting tools (see below).

Experiments. As illustrated by the discussion of properties in the Section 3, PRISM places emphasis on the analysis of quantitative properties. Often, it is more instructive to generate and plot a range of such values, as a parameter of either the property or model is varied. This can be invaluable for studying trends in quantitative results or identifying anomalous behaviour of the system. Examples are "the instantaneous availability of the system at time t" for a range of time values t or "the expected throughput of the system" for a range of different component failure rates. PRISM is designed to facilitate such queries and includes a simple mechanism, known as *experiments*, for this purpose. The tool includes an integrated graph plotting tool for visualising and displaying results. Figure 3 shows the results of an example experiment and Figure 4(b) shows a screenshot of PRISM being used to generate such results.

Discrete-event simulation. PRISM also incorporates a discrete-event simulation engine, which has two purposes. Firstly, it forms the basis of a tool for debugging models (see Figure 4(a)). This can be used for either manual exploration or creation of random traces. Secondly, it provides generation of approximate solutions to the numerical computations that underlie the model checking process, by applying Monte Carlo methods and sampling. These techniques complement the main model model checking functionality of the tool, offering increased scalability, but at the expense of numerical accuracy.

4.2 Underlying techniques

In this section, we give an overview of the techniques that are implemented in PRISM for the probabilistic model checking of continuous-time Markov chains and Markov reward models. More detailed coverage of this topic, including details for the other types of probabilistic models supported by PRISM (discrete-time Markov chains and Markov decision processes) can be found on the PRISM website [16].

Model checking algorithms. The basic algorithms required for model checking are those proposed for PCTL [9] and for CSL [5]. Algorithms for the reward operators, which are based on standard techniques for Markov reward models, can be found in [13]. The key ingredients of these algorithms can be classified into two groups: *graph-theoretical algorithms* and *numerical computation*. The former class, which operate on the underlying graph structure of a Markov chain, are used, for example, to determine the set of reachable states in a model or to model check qualitative properties. Numerical computation is required for the calculation of probabilities and reward values, such as those illustrated in Section 3.

For numerical computation, model checking typically requires either solution of linear equation systems or calculation of the transient probabilities of a Markov chain. Because of the large size of the models that often arise, the tool uses iterative methods rather than direct methods. For solution of linear equation systems, PRISM supports a range of well-known techniques including the Jacobi, Gauss-Seidel and SOR (successive over-relaxation) methods; for transient probability computation, it uses uniformisation. See for example [17] for good coverage of both these topics.

Symbolic implementation. An important aspect of the implementation of PRISM is its use of state-of-the-art *symbolic* techniques, applying data structures based on binary decision diagrams (BDDs). Some aspects of the probabilistic model checking process, such as reachability and graph-theoretical algorithms, can be implemented with well known BDD-based approaches. In general, however, representing

and manipulating probabilistic models requires extensions of these techniques. PRISM uses a generalisation of BDDs called *multi-terminal BDDs* (MTBDDs) [7, 3]. These provide compact representation and efficient manipulation of large, structured models by exploiting regularities exhibited in the high-level modelling language descriptions.

In fact, for the numerical solution aspect of the probabilistic model checking process, which is typically the most resource-intensive, PRISM provides a choice of three distinct computation *engines*. The first is purely symbolic, using BDDs and MTBDDs only. For models with a large degree of regularity, this option can be extremely efficient, allowing PRISM to scale to the analysis of probabilistic models with as many as 10^{10} states. Often though, especially for analysis of CTMCs, MTBDDs perform poorly in this phase due to irregularity in the numerical values computed. By contrast, the second engine is implemented using explicit data structures: sparse matrices and arrays. This provides more predictable performance and, where usable, is usually the fastest engine. It is, though, more demanding in terms of memory usage, limiting its applicability to large models.

The final PRISM engine is called the *hybrid* engine, which uses extensions of MTBDDs [11, 15] to combine advantages of the other two engines. It is generally faster (and more often feasible) than the symbolic engine but uses less memory than sparse matrices and is the default engine in PRISM. This choice of engines provides a flexible implementation which can be adjusted to improve performance depending on the type of models and properties being analysed.

5. RESOURCES AND INFORMATION

Obtaining PRISM. PRISM is an open source project and the tool is distributed under the GNU General Public License (GPL). Both source code and binary versions are available from the website [16]. The tool runs on all major platforms, including Linux, Unix, Windows and Macintosh operating systems. In addition to regular "public" releases, "development" versions of PRISM are made available through the web site. These provide access to recent and ongoing additions to the tool.

Case studies. PRISM has been used to analyse the performance and reliability of case studies from a wide range of sources. This includes dynamic power management systems, embedded control systems, nano-scale circuitry, queueing systems, computer networks and manufacturing systems. It has also been used to analyse systems as diverse as communication and multimedia protocols, randomised security protocols and biological processes.

The tool website hosts a repository which provides detailed information for more than 45 case studies. These include both contributions from the PRISM team and from external sources. Further contributions are always welcome. The website also contains many pointers to work describing other case studies and includes a detailed bibliography of publications on this topic.

Further information. The website [16] includes a selection of additional resources for those interested in learning more about PRISM and the techniques on which it is based. This includes related publications, an 11-part lecture course, an online manual and additional technical documentation. There is also a discussion forum to support users of the tool.

6. CONCLUSION

This paper presented a summary of the probabilistic model checker PRISM, with particular emphasis on its application to performance analysis of computer and communication systems. Development of PRISM is ongoing. Current work includes extending the range of supported temporal logics, improving efficiency and integrating techniques such as symmetry reduction and bisimulation minimisation.

Acknowledgements. The authors are supported in part by the EPSRC grants EP/D07956X and EP/D076625.

7. 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 *Proc. 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. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In *Proc. ICALP'00*, volume 1853 of *LNCS*, pages 780–792. Springer, 2000.

[5] 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.

[6] 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. CONCUR'99*, volume 1664 of *LNCS*, pages 146–161. Springer, 1999.

[7] 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. *Formal Methods in System Design*, 10((2/3):149–169, 1997.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

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

[10] V. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.

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

[12] M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427–1434, 2006.

[13] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.

[14] J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability*, 1(2):9–20, 1994.

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

[16] PRISM web site. www.prismmodelchecker.org.

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