CONTROLLER DEPENDABILITY ANALYSIS BY PROBABILISTIC MODEL CHECKING

Marta Kwiatkowska, Gethin Norman and David Parker

School of Computer Science, University of Birmingham, Birmingham, B15 2TT, United Kingdom

Abstract: We demonstrate how probabilistic model checking, a formal verification method for the analysis of systems which exhibit stochastic behaviour, can be applied to the study of dependability properties of software-based control systems. We provide an overview of these techniques and of the probabilistic model checking tool PRISM, illustrating the usefulness of the approach through a small case study. By using existing formalisms and tool support, we show how it is possible to construct large and complex Markov models from an intuitive high-level description. Furthermore, we are able to take advantage of the efficient implementation techniques which have been developed for these tools.

Keywords: Formal verification, performance analysis, Markov models

1. INTRODUCTION

Industrial use of software-based control systems is now increasingly widespread. Programmable controllers can be found in domains as diverse as the manufacturing, transport, energy and power industries. In many of these cases, the safety and reliability of the system is a crucial issue. This fact is evidenced, for example, by the existence, and indeed prominence, of the IEC 61508 and ANSI/ISA S84 standards, which include strict guidelines as to the functional safety of such systems.

A key feature of these standards is the definition of Safety Integrity Levels (SILs). To adhere to a particular SIL, it is necessary to accurately quantify the probability or rate of all safety-related faults which can occur in the system under study. Two common techniques for performing such dependability analysis are reliability block diagrams and fault tree analysis. Both, however, are relatively simplistic. For example, it is often necessary to assume that the probability of a certain fault arising is independent of the occurrence of other failures in the system, whereas in practice, this may be unrealistic and can lead to inaccuracies in the analysis.

A more sophisticated approach which resolves this problem is Markov modelling, where a more accurate, state-based model of the system is derived and analysed. This does, however, have two disadvantages. Firstly, it is more complex for the end-user to implement. Secondly, the size of model required to represent a system accurately has a tendency to increase exponentially. This is a phenomenon often known as the state space explosion problem. Not only does it make the process more expensive in terms of the time, computing power and memory required, it can make the reliability analysis infeasible.

In this paper, we show how formal verification methods can be applied to the problem of analysing the dependability of controller-based systems. In particular, we focus on probabilis-

[1] Partially supported by EPSRC grants GR/S46727 and GR/S11107

tic model checking, an automatic technique for checking whether or not probabilistic models satisfy certain specifications. We show how existing formalisms and tools can be used to specify and analyse these Markov models. In addition, we can then make use of the efficient implementation techniques which have been integrated into these tools in an attempt to curb the effects of state space explosion.

2. PROBABILISTIC MODEL CHECKING

Model checking is a successful and well established technique for formally verifying the correctness of finite-state systems. In recent years, such methods have become increasingly prevalent in industry. Model checking involves the construction of a formal model of the real-life system which is to be verified. This is usually a labelled state transition system, which represents all the possible configurations which the system can be in and all the transitions which can occur between them. The properties of the system to be verified are then also formally specified, usually as formulas of a temporal logic, and passed to a model checker, which automatically determines whether or not each property is satisfied via a systematic exploration of the model.

An extension of this approach which has seen a significant amount of development of late is *probabilistic model checking*, a technique which permits automatic formal verification of systems which exhibit stochastic behaviour. Potential candidates for such analysis include randomised algorithms, which use probabilistic choices or electronic coin flipping, and unreliable or unpredictable processes, such as fault-tolerant systems or communication networks.

Probabilistic model checking is again based on the construction and analysis of a formal model of the system. In this case, the model is enriched with probabilistic information, typically by labelling each transition of the model with information about the likelihood that it will occur. The type of model which we shall focus upon in this paper is *continuous-time Markov chains* (CTMCs). A CTMC comprises a set of states S and a transition rate matrix $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$. The *rate* $\mathbf{R}(s, s')$ defines the delay before which a transition between states s and s' is enabled. The delay is sampled from a negative exponential distribution with parameter equal to this rate, i.e. the probability of the transition being enabled within t time units is $1 - e^{-\mathbf{R}(s,s')\cdot t}$. When $\mathbf{R}(s, s')>0$ for two states s', a *race* occurs and the transition which becomes enabled first is the one taken. Exponentially distributed delays are often suitable for modelling component lifetimes and inter-arrival times. Furthermore, they can be used to approximate more complex probability distributions.

Other models commonly used for probabilistic model checking are discrete-time Markov chains (DTMCs), which specify the probability of moving between states in discrete time-steps, and Markov decision processes (MDPs), which can model systems which exhibit both probabilistic and nondeterministic behaviour, for example a system which comprises a number of probabilistic processes operating asynchronously in parallel.

Given a probabilistic model, it is then necessary to specify its required properties. Traditionally, in the model checking paradigm, properties are expressed using temporal logic, which provides a concise and unambiguous specification. We will use the temporal logic CSL (Continuous Stochastic Logic) Aziz *et al.* (1996); Baier *et al.* (1999) which is designed for specifying properties of CTMCs. We do not present the full syntax and semantics of the logic here, instead providing a number of illustrative examples with their natural language translation:

- $\mathcal{P}_{\leq 0.01}[\Diamond\ \mathit{shutdown}]$ – "shutdown eventually occurs with probability at most 0.01"
- $\mathcal{P}_{\geq 0.95}[\neg \mathit{repair}\ \mathcal{U}^{\leq 200}\ \mathit{complete}]$ – "with probability 0.95 or greater, the process will successfully complete within 200 hours and without requiring any repairs"
- $\mathcal{S}_{>0.75}[\mathit{num\_sensors} \geq \mathit{min}]$ "in the long-run, the probability that an adequate number of sensors are operational is greater than 0.75"

Note that the use of probability bounds (≤ 0.01, ≥ 0.95, > 0.75) ensures that the properties above constitute questions which can be verified either to be true or false, as is traditionally the case in formal verification. In practice, though, it is often more useful to request the actual values, writing for example:

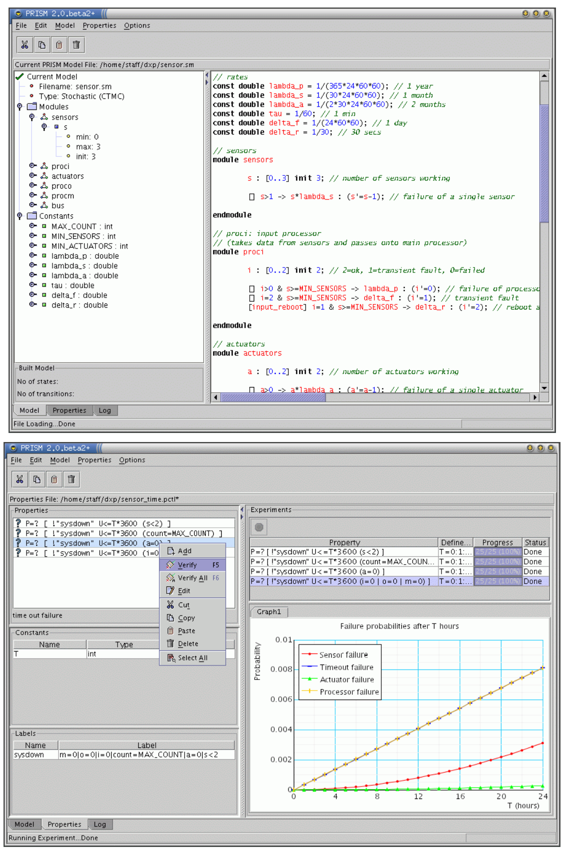
- $\mathcal{P}_{=?}[\Diamond^{\leq T}\mathit{shutdown}]$ – "what is the probability that the system shuts down by time T?"

Furthermore, the most useful way to analyse the model and to gain insights into its reliability may be to compute and plot such a value as some parameter is varied (e.g. T in the formula above, or a constant in the model itself).

Additional properties can be specified by adding the notion of *rewards*. We can assign each state (or transition) of the model a real-valued reward and then write queries such as:

- $\mathcal{R}_{=?}[\Diamond \mathit{success}]$ – "what is the expected reward accumulated before the system successfully terminates?"

Rewards can be used to specify a wide range of measures of interest. Of course, conversely, we



PRISM 2.0.beta2+

Current PRISM Model File: /home/staff/dxp/sensor.sm

```
Current Model
  Filename: sensor.sm
  Type: Stochastic (CTMC)
Modules
  sensors
    s
      min: 0
      max: 3
      init: 3
  proci
  actuators
  proco
  procm
  bus
Constants
  MAX_COUNT : int
  MIN_SENSORS : int
  MIN_ACTUATORS : int
  lambda_p : double
  lambda_s : double
  lambda_a : double
  tau : double
  delta_f : double
  delta_r : double

Built Model
No of states:
No of transitions:
```

```
// rates
const double lambda_p = 1/(365*24*60*60); // 1 year
const double lambda_s = 1/(30*24*60*60); // 1 month
const double lambda_a = 1/(2*30*24*60*60); // 2 months
const double tau = 1/60; // 1 min
const double delta_f = 1/(24*60*60); // 1 day
const double delta_r = 1/30; // 30 secs

// sensors
module sensors

        s : [0..3] init 3; // number of sensors working

        [] s>1 -> s*lambda_s : (s'=s-1); // failure of a single sensor

endmodule

// proci: input processor
// (takes data from sensors and passes onto main processor)
module proci

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

        [] i>0 & s>=MIN_SENSORS -> lambda_p : (i'=0); // failure of processo
        [] i=2 & s>=MIN_SENSORS -> delta_f : (i'=1); // transient fault
        [input_reboot] i=1 & s>=MIN_SENSORS -> delta_r : (i'=2); // reboot a

endmodule

// actuators
module actuators

        a : [0..2] init 2; // number of actuators working

        [] a>0 -> a*lambda_a : (a'=a-1); // failure of a single actuator
```

File Loading...Done

PRISM 2.0.beta2+

Properties File: /home/staff/dxp/sensor_time.pctl*

```
P=? [ !"sysdown" U<=T*3600 (s<2) ]
P=? [ !"sysdown" U<=T*3600 (count=MAX_COUNT) ]
P=? [ !"sysdown" U<=T*3600 (a=0) ]
P=? [ !"sysdown" U<=T*3600 (i=0
```

time out failure

Constants

Name	Type
T	int

Labels

Name	Label
sysdown	m=0|o=0|i=0|count=MAX_COUNT|a=0|s<2

Experiments

Property	Define...	Progress	Status
P=? [!"sysdown" U<=T*3600 (s<2)]	T=0:1:...	25/25 (100%)	Done
P=? [!"sysdown" U<=T*3600 (count=MAX_COUN...	T=0:1:...	25/25 (100%)	Done
P=? [!"sysdown" U<=T*3600 (a=0)]	T=0:1:...	25/25 (100%)	Done
P=? [!"sysdown" U<=T*3600 (i=0 | o=0 | m=0)]	T=0:1:...	25/25 (100%)	Done

Running Experiment...Done

Fig. 1. Screenshots of PRISM running

can also consider the rewards to be costs. We can then analyse, for example, expected power consumption, expected number of failures, etc. Suitable temporal logics for expressing cost- and reward-based properties can be found in e.g. Baier *et al.* (2000); de Alfaro (1997).

A *probabilistic model checker* applies algorithmic techniques to analyse the state space of a probabilistic model and determine whether its specifications are satisfied. Typically, this involves computation of one or more probabilities or performance measures. The operations required are graph-based analysis and methods for solving either linear equation systems or linear optimisation problems.

3. PRISM

PRISM Kwiatkowska *et al.* (2002) is a probabilistic model checker developed at the University of Birmingham. It provides automatic verification of CSL properties for CTMCs. In addition, it supports analysis of DTMCs and MDPs. The tool has been used to analyse a wide range of case studies, including: randomised algorithms for problems such as leader election, mutual exclusion and consensus; real-time communication protocols; security protocols; communication networks; and dynamic power management schemes.

Probabilistic models to be analysed in PRISM are specified in the PRISM language, which is based on the Reactive Modules formalism of Alur and Henzinger Alur and Henzinger (1999). A model is described as a number of *modules*, each of which corresponds to a component of the real-life system. Each module has a set of finite-ranged *variables*. These determine the possible states of each module. The whole model is constructed as the parallel composition of these modules.

The behaviour of an individual module is specified by a set of guarded commands. For a CTMC, as is the case here, a command takes the form:

$$[] \ \texttt{<guard>} \ \rightarrow \ \texttt{<rate>} \ : \ \texttt{<action>} \ ;$$

The *guard* is a predicate over the variables of all the modules in the model. The *update* comprises `<rate>`, an expression which evaluates to a positive real number, and `<action>`, which describes a transition of the module in terms of how its variables should be updated. The interpretation of the command is that if the guard is satisfied, then the module can make the corresponding transition with that rate (see Section 2 for a definition of rate). A simple command for a module with one variable x might be:

$$[] \ (x = 0) \ \rightarrow \ 4.5 \ : \ (x' = x + 1) \ ;$$

which states that, if x is 0, it is incremented by one and this action occurs with rate 4.5.

The overall functionality of the PRISM tool is as follows. First, it reads and parses a system description in the PRISM language. It then constructs, from this, the corresponding probabilistic model, in this case a CTMC (although the same description language can be used for both DTMCs and MDPs). PRISM also computes the set of all states which are reachable from the initial state and identifies any deadlock states (i.e. reachable states with no outgoing transitions). If required, the transition matrix of the probabilistic model constructed can be exported for use in another tool. Typically, though, PRISM then parses one or more temporal logic properties (e.g. in CSL) and performs model checking, determining whether the model satisfies each property. The graphical user interface of the tool also allows automatic construction of graphs to visualise results more easily. Figure 1 shows two screenshots of the tool running.

Another important feature of the tool is its implementation. PRISM is a "symbolic" model checker, meaning that it has been developed using data structures base on binary decision diagrams (BDDs). BDDs are reduced, directed acyclic graphs which can be very effective for compact storage of models which exhibit structure and

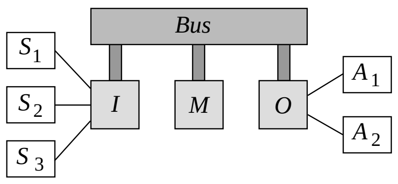


Fig. 2. The embedded system

regularity (e.g. those which have been specified in a high-level description language such as the PRISM language). In practice, this allows construction and analysis of larger models than would otherwise be possible. It also allows efficient BDD-based algorithms developed for non-probabilistic model checking to be easily integrated.

The tool and its source code can be freely downloaded under the GNU General Public License from www.cs.bham.ac.uk/~dxp/prism. Tool documentation, relevant research papers, and detailed information about a wide range of previous case studies can also be found here. In this paper, we have made use of a prototype extension of PRISM which supports model checking of the cost- and reward-based properties mentioned in the previous section. These features will be integrated into the main version in due course.

4. CASE STUDY

To illustrate the applicability of probabilistic model checking and the PRISM tool to the process of dependability analysis, we now present a small case study. We model an embedded system, closely based on the one presented in Muppala *et al.* (1994), the structure of which is shown in Figure 2. The system comprises an input processor (I) which reads and processes data from three sensors (S_1, S_2, S_3). It is then polled by a main processor (M) which, in turn, passes instructions to an output processor (O). This process occurs cyclically, with the length of the cycle controlled by a timer in the main processor. The output processor takes the instructions it receives and uses them to control two actuators (A_1, A_2). Finally, a bus connects the three processors together. A concrete example of such a system might be a gas boiler, where the sensors are thermostats and the actuators are valves.

Any of the three sensors can fail, but they are used in *triple modular redundancy*: the input processor can determine sufficient information to proceed provided two of the three are functional. If more than one becomes unavailable, the processor reports this fact to the main processor and the system is shut down. In similar fashion, it is sufficient for only one of the two actuators to be working, but if this is not the case, the output processor tells the main processor to shut the system down.

```
// a sensor fails on average once a month
const double λ_s = 1/(30 · 24 · 60 · 60);
module sensors
    // number of sensors working
    s : [0..3] init 3;
    // failure of a single sensor
    [] s > 0  →  s · λ_s : (s' = s − 1);
endmodule
```

Fig. 3. PRISM code for the sensors

The I/O processors themselves can also fail. This can be either a permanent fault or a transient fault. In the latter case, the situation can be rectified automatically by the processor rebooting itself. In either case, if the I or O processor is unavailable and this leads to M being unable either to read data from I or send instructions to O, then M is forced to skip the current cycle. If M detects that the number of consecutive cycles skipped exceeds a limit, K, then it assumes that either I or O has failed and shuts the system down. Unless specified otherwise, we assume a value of 2 for K. Lastly, the main processor can also fail, in which case the system is automatically shut down.

The mean times to failure for a sensor, actuator and processor are 1 month, 2 months and 1 year, respectively. A transient fault occurs in a processor once a day on average. The mean times for a timer cycle to expire and for a processor reboot to complete are 1 minute and 30 seconds, respectively. We assume that all these delays are distributed exponentially; hence the system can be modelled as a CTMC.

Our PRISM model of the system comprises 6 modules, one for the sensors, one for the actuators, one for each processor and one for the connecting bus. In Figure 3, we show the section of the PRISM language description which models the sensors. This constitutes a single module *sensors* with an integer variable s representing the number of sensors currently working. The module's behaviour is described by one guarded command which represents the failure of a single sensor. Its guard "$s > 0$" states this can occur at any time, except when all sensors have already failed. The action ($s' = s - 1$) simply decrements the counter of functioning sensors. The rate of this action is $s \cdot \lambda_s$, where λ_s is the rate for a single sensor and s is the PRISM variable referred to previously which denotes the number of active sensors.

In Figure 4, we show a second module which is the PRISM language description of the input processor. The module has a single variable i with range [0..2] which indicates which of the three possible states the processor is in, i.e. whether it is working, is recovering from a transient fault, or has failed. The three guarded commands in the

```
const double $\lambda_p = 1/(365 \cdot 24 \cdot 60 \cdot 60)$;
const double $\delta_f = 1/(24 \cdot 60 \cdot 60)$;
const double $\delta_r = 1/30$;
module proci
    // state: 2=ok, 1=transient fault, 0=failed
    $i : [0..2]$ init 2;
    // failure of processor
    [] $i > 0$ & $s \geq 2 \;\rightarrow\; \lambda_p : (i' = 0)$;
    // transient fault
    [] $i = 2$ & $s \geq 2 \;\rightarrow\; \delta_f : (i' = 1)$;
    // reboot after transient fault
    [input_reboot] $i = 1 \;\rightarrow\; \delta_r : (i' = 2)$;
endmodule
```

Fig. 4. PRISM code for the input processor

module correspond, respectively, to the processor failing, suffering a transient fault, and rebooting. The commands themselves are fairly self explanatory. Two points of note are as follows. Firstly, the guards of these commands can refer to variables from other modules, as evidenced by the use of $s \geq 2$. This is because the input processor ceases to function once it has detected that less than two sensors are operational. Secondly, the last command contains an additional label *input_reboot*, placed between the square brackets at the start of the command. This is used for synchronising actions between modules, i.e. allowing two or more modules to make transitions simultaneously. Here, this is used to notify the main processor of the reboot as soon as it occurs.

The full version of the PRISM code for this case study is available from the tool website.

5. RESULTS

We have used PRISM to construct the CTMC representing the embedded system described in the previous section and to analyse a number of dependability properties using probabilistic model checking. First, we consider the probability of the system shutting itself down. Note that there are four distinct types of failure which can cause a shutdown: faults in (1) the sensors (2) the actuators (3) the input/output processors (4) the main processor. We can analyse how likely each of these is to be the cause of the shutdown, as time passes. We use the CSL property:

- $\mathcal{P}_{=?}[\neg down\ \mathcal{U}^{\leq T}\ fail_j]$

where $j = 1 \dots 4$, refers to one of the four failures above and *down* denotes that any of the failures has occurred, i.e. $down = fail_1 \vee fail_2 \vee fail_3 \vee fail_4$. The atomic propositions which make up the property would in practice be predicates over the variables from the PRISM language description. For example, $fail_1$, the failure of more than one sensor, is written "$s < 2 \wedge i = 2$", meaning that the number of working sensors has dropped below 2 and the input processor is functioning (and so can report the failure).

The above property denotes the probability that failure j is the *first* to occur. Note that, for example, if an actuator fails, the sensors, unaware of this, will continue to operate and may subsequently fail. Hence, we need to determine the likelihood of each failure occurring, before any of the others do. This is a good illustration of how non-trivial properties can be captured using temporal logic.

In Figure 5(a), we plot the results of this computation over two ranges of values for T: the first 24 hours and the first month of operation. We can see, for example, that while initially the I/O processors are more likely to cause a system shutdown, in the long run it is the actuators which are most unreliable. By omitting the bound $\leq T$ from the CSL formula:

- $\mathcal{P}_{=?}[\neg down\ \mathcal{U}\ fail_j]$

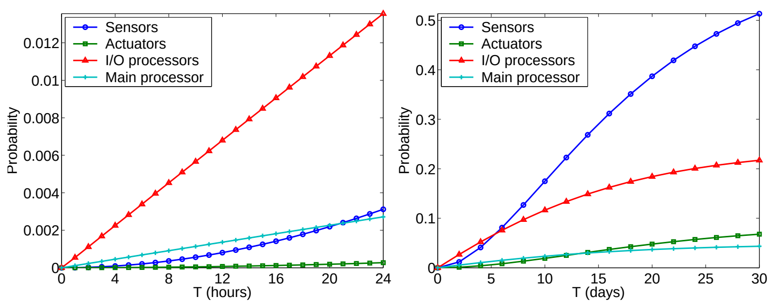
we can ask the model checker to compute the long-run failure probability (i.e. as $T \rightarrow \infty$). The results are: (1) 0.6216 (2) 0.0877 (3) 0.0484 (4) 0.2423. Note that these sum to one, i.e. the system will eventually shut down with probability 1.

Second, we consider a number of cost-based properties. We classify the states of our model into three types: "up", where everything is functioning, "shutdown", where the system has shut down, and "danger", where a (possibly transient) failure has occurred but has yet to cause a shutdown (e.g. if the I or O processor has failed but the M processor has yet to detect this). We can reason about the time spent in each class by assigning a cost of 1 to those states, 0 to all others, and then computing the total cumulated reward. We use the properties:

- $\mathcal{R}_{=?}[\mathcal{C}^{\leq T}]$
- $\mathcal{R}_{=?}[\mathcal{F}\, down]$

These give us the expected time spent in each class, up until time T and before the system shuts down, respectively. We plot the results for the first value in Figure 5(b), again for two ranges of T: the first day and the first month of operation. In the table in Figure 5(c), we give the results of the second property, as we vary K, the number of skipped cycles which the main processor waits before deciding that the input/output processors have failed. We see that increasing the value of K increases the expected time until failure, but also has an adverse effect on the expected time spent in "danger" states.

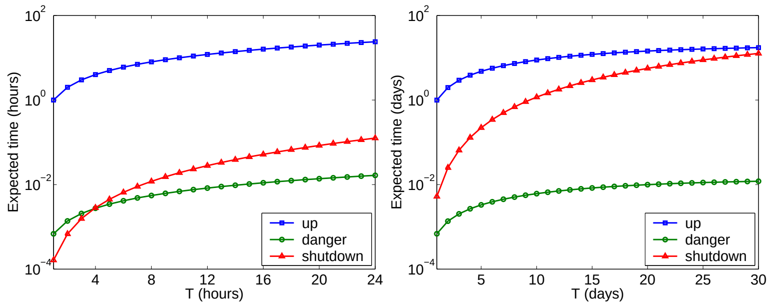
Finally, to illustrate a different type of cost structure, we compute the expected number of processor reboots which occur over time. This is done by assigning a cost of 1 to all transitions which



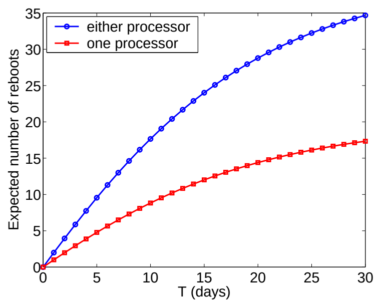
(a) Probability of each cause of shutdown

K	Expected time	
	danger (hrs)	up (days)
1	0.236	14.323
2	0.293	17.660
3	0.318	19.100
4	0.327	19.628
5	0.330	19.809
6	0.331	19.871
7	0.332	19.891

(c) Expected time



(b) Expected time spent in "up"/"shutdown"/"danger"



(c) Expected reboots

Fig. 5. Reliability results for the case study obtained using the probabilistic model checker PRISM

constitute a reboot and 0 to all others. The results are plotted in Figure 5(d).

In the experiments we have presented in this section, the number of states in the CTMC, which depends on the parameter K, varied between 2,633 and 7,703. The models were all constructed in less than second. For the model checking itself, the time required to compute each value varied from a few seconds to a few minutes, using a standard workstation.

6. CONCLUSION

We have illustrated how probabilistic model checking, a formal verification technique which has already been applied to a wide range of domains including distributed randomised algorithms, real-time protocols, security protocols and dynamic power management, can be used to analyse dependability properties of controller-based systems. We have used the probabilistic model checker PRISM whose simple system description language provides an intuitive way to construct complex Markov models. For specifying properties to check, we used temporal logic, which allows us to reason about non-trivial behaviour. We were also able to make use of the efficiency improvements which have been developed in this area. A further advantage of this formal verification approach is that it could easily be combined with more traditional non-probabilistic verification processes, which are becoming increasingly common, particularly in safety-critical areas.

For more detailed information about probabilistic model checking, PRISM and its application to the case study described in this paper, see the tool website: www.cs.bham.ac.uk/~dxp/prism

REFERENCES

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

Aziz, A., K. Sanwal, V. Singhal and R. Brayton (1996). Verifying continuous time Markov chains. In: *Proc. CAV'96*. Vol. 1102 of *LNCS*. Springer. pp. 269–276.

Baier, C., B. Haverkort, H. Hermanns and J.-P. Katoen (2000). On the logical characterisation of performability properties. In: *Proc. ICALP'00*. Vol. 1853 of *LNCS*. Springer. pp. 780–792.

Baier, C., J.-P. Katoen and H. Hermanns (1999). Approximate symbolic model checking of continuous-time Markov chains. In: *Proc. CONCUR'99*. Vol. 1664 of *LNCS*. Springer. pp. 146–161.

de Alfaro, L. (1997). Formal Verification of Probabilistic Systems. PhD thesis. Stanford University.

Kwiatkowska, M., G. Norman and D. Parker (2002). PRISM: Probabilistic symbolic model checker. In: *Proc. TOOLS'02*. Vol. 2324 of *LNCS*. Springer. pp. 200–204.

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