Probabilistic Symbolic Model Checking: Theory and Practice

Ye Wang

Department of Computer Science
University of Toronto, Toronto, ON. Canada, M5S 3G4

Hui Lan

Department of Computer Science
University of Toronto, Toronto, ON. Canada, M5S 3G4

Abstract

Model checking has made significant advancement in the past two decades. In this report, we will focus on probabilistic symbolic model checking. We will describe three probabilistic system models, namely, discrete-time Markov chain, Markov decision process, and continuous-time Markov chain. With each model, we will examine its corresponding specification language and model checking algorithm. Then we will use the probabilistic symbolic model checking tool, PRISM, to verify properties of a simple web server with possible hardware and software failures.

Key words: probabilistic model checking

1 Introduction

Model checking has been recognized as a state-of-the-art technique for verifying the correctness of hardware, communication protocols, and lately, mission critical software modules. Conventional [1] model checking uses boolean values to model states and transitions. For example, property A is true in state b; at state c, the system can make a transition to state d, e, or f. Time can be incorporated into the model to verify real time systems. Both discrete real time and continuous real time can be represented. When the truth value of a

[1] In this report, we will use *conventional* to address model checking that does not explicitly involve probability

property or transition is not know for sure, 3-valued model checking can be used to resolve the problem, at a cost of performance [13].

However, in real world, not all systems are boolean. Many systems contain probabilistic factors. For example, 99% of the time, the system will respond to a request; at most 5% of the processes will fail. To model such systems directly, probabilistic model checking techniques are developed.

Many system models have been studied intensively. In this report, we will describe three of them and their corresponding specification languages, namely, discrete-time Markov chain (DTMC) and probabilistic computation tree logic (PCTL), Markov decision process (MDP) and pCTL, continuous time Markov chain (CTMC) and continuous stochastic logic (CSL). The characteristics of the models are listed in the table below.

Model	Random	Continuous-Time	Specification Language
DTMC			PCTL
MDP	$\surd$		pCTL
CTMC	$\surd$	$\surd$	CSL

The rest of the report is organized as follow. In section 2, 3, 4, we will describe the probabilistic models, introduce their associated specification languages and symbolic model checking methods. In section 5 we will briefly describe the working of PRISM. In section 6, we will talk about our experience of PRISM while modeling a simple web server. In the conclusion, we will compare probabilistic model checking with conventional ones.

2 Discrete-Time Markov Chain and PCTL

2.1 Discrete-Time Markov Chain

Discrete time Markov chain is used to model systems that determine transitions based on predefined probabilistic distribution.

Definition 1 *A* discrete time Markov chain *(DTMC) D is a tupe* $(S, s_i, \mathbf{P}, L)$ *where*

- S *is a finite set of states*
- s_i *is the initial state*
- $\mathbf{P} : S \times S \to [0, 1]$ *is a probability matrix such that* $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ *for all* $s \in S$*, and*

- *$L : S \to 2^{AP}$ is a labeling function that labels each state in S with the atomic propositions that are true in that state.*

Figure 1 is a DTMC representing a probabilistic system. In each state, the system makes a transition according to the probability distribution defined by $\mathbf{P}$. For example, at state s_1, the system has 60% chance of moving to s_2 and 40% chance of moving to s_4. A *path* or a *execution sequence* through a DTMC is a (finite or infinite) sequence of states $\sigma = s_0 s_1 s_2 \cdots$, with $\mathbf{P}(s, s') > 0$ for all $i \geq 0$. s_0 is the starting state. The state s_n in σ is denoted $\sigma[n]$, and the prefix $s_0 \cdots s_n$ is denoted $\sigma{\uparrow}n$. For each DTMC D and state s, a *probability measure* on the set of paths from s, is defined as below.

Definition 2 *[2] (Probability measure) Let $paths^D_{s_0}$ denote the set of paths of D starting at s_0. Define:*

- *For any sequence $s_0 \cdots s_n$*

$$\mu^D_{s_0}(\{\sigma \in paths^D_{s_0} \mid \sigma{\uparrow}n = s_0 \cdots s_n\}) = \mathbf{P}(s_0, s_1) \times \cdots \times \mathbf{P}(s_{n-1}, s_n)$$

  Intuitively, the measure [2] of a finite path $\sigma = s_0 \cdots s_n$ is the product of the probability of each transition in σ. It can also be regarded as the sum of the measures of all the paths whose prefix is $\sigma = s_0 \cdots s_n$.
- *For $n = 0$*

$$\mu^D_{s_0}(\{\sigma \in paths^D_{s_0} \mid \sigma{\uparrow}n = s_0 \cdots s_n\}) = 1$$

  Intuitively, if there is no transition, the measure is 1.
- *For any countable set $\{X_i\}_{i \in I}$ of disjoint subsets of $paths^D_{s_0}$*

$$\mu^D_{s_0}(\bigcup_{i \in I} X_i) = \sum_{i \in I} \mu^D_{s_0}(X_i)$$

  Two subsets of $paths^D_{s_0}$ are disjoint if they do not contain the same path. For example, in figure 1, $\{s_1(s_2s_3)^m s_4^w\}$ and $\{s_1(s_2s_3)^n s_4^w\}$ are disjoint if $n \neq m$. This tells us that probability measure is addable, and is well defined (the sum is in the range $[0, 1]$). The well-definedness can be proved using induction.
- *If X is a subset of $paths^D_{s_0}$, then the measure of the complement set $paths^D_{s_0} \setminus X$ is defined as:*

$$\mu^D_{s_0}(paths^D_{s_0} \setminus X) = 1 - \mu^D_{s_0}(X)$$

[2] In this report, we will use *probability measure* and *measure* interchangeably

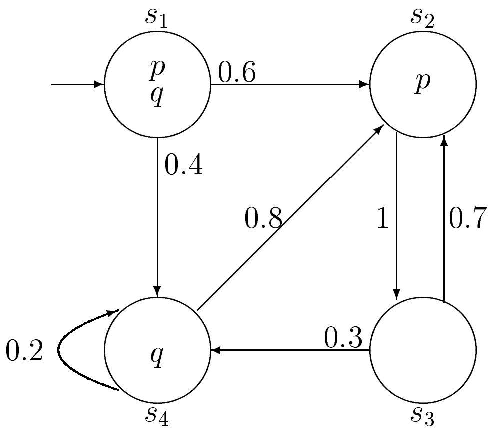

Fig. 1. A discrete time Markov chain modeling a probabilistic system

2.2 PCTL

PCTL stands for Probabilistic CTL. It was first introduced in [2]. The original PCTL was used to model discrete time Markov chain. Researchers have extended PCTL to include features for modeling nondeterminism and continuous-time, for example, in [10]. In this report, we will use the original definition.

Definition 3 *[2] (PCTL Syntax) The syntax of PCTL formulas is defined inductively as follows:*

- *Each atomic proposition a is a PCTL formula*
- *If f_1 and f_2 are PCTL formulas, then so are $\neg f_1$ and $(f_1 \wedge f_2)$*
- *If f_1 and f_2 are PCTL formulas, t is a nonnegative integer or ∞, and p is a real number with $0 \leq p \leq 1$, then $f_1 U^{\leq t}_{\geq p} f_2$ and $f_1 U^{\leq t}_{> p} f_2$ are PCTL formulas*

There are two types of PCTL formulas: state formulas and path formulas. If a state formula f is true in state s in a DTMC D, we say

$$s \models_D f$$

If a path σ satisfy the path formula f in D, we say

$$\sigma \models\!\equiv_D f$$

Definition 4 *[2] (PCTL Semantics) The relations $\models_D$ and $\models\!\equiv_D$ are inductively defined as follows:*

$$
\begin{array}{ll}
s \models_D a & \textit{iff } a \in L(s) \\
s \models_D \neg f & \textit{iff not } s \models_D f \\
s \models_D f_1 \wedge f_2 & \textit{iff } s \models_D f_1 \textit{ and } s \models_D f_2 \\
\sigma \models_D f_1 U^{\leq t} f_2 & \textit{iff there exists an } i \leq t \textit{ such that} \\
 & \sigma[i] \models_D f_2 \textit{ and} \\
 & \sigma[j] \models_D f_1, \textit{ for all } j : 0 \leq j < i. \\
s \models_D f_1 U^{\leq t}_{\geq p} f_2 & \textit{iff } \mu^D_s(\{\sigma \mid \sigma[0] = s \wedge \sigma \models_D f_1 U^{\leq t} f_2\}) \geq p. \\
s \models_D f_1 U^{\leq t}_{> p} f_2 & \textit{iff } \mu^D_s(\{\sigma \mid \sigma[0] = s \wedge \sigma \models_D f_1 U^{\leq t} f_2\}) > p.
\end{array}
$$

U is the *strong until* operator. $f_1 U^{\leq t}_{\geq p} f_2$ represents with probability at least p, f_2 will become true within t time unite, and f_1 will remain true until then.

The above semantics form an adequate set for PCTL. Other formulas can be represented as follows:

$$f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$$

$$f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$$

$$f_1 u^{\leq t}_{\geq p} f_2 \equiv \neg(\neg f_2 U^{\leq t}_{> 1-p} \neg(f_1 \vee f_2))$$

$$f_1 u^{\leq t}_{> p} f_2 \equiv \neg(\neg f_2 U^{\leq t}_{\geq 1-p} \neg(f_1 \vee f_2))$$

$$G^{\leq t}_{\geq p} f \equiv f \ u^{\leq t}_{\geq p} false$$

$$F^{\leq t}_{\geq p} f \equiv true \ U^{\leq t}_{\geq p} f$$

where u is the *weak until* operator. The formula $f_1 u^{\leq t}_{\geq p} f_2$ means that with at least probability p that either

> f_1 will remain true for at least t time units

or

> f_2 will become true within t time units and f_1 will be true from now on until f_2 becomes true

In fact, CTL can be regarded as a subset of PCTL.

$$A[f_1 U \ f_2] \equiv f_1 U^{\leq \infty}_{\geq 1} f_2$$

$$E[f_1 U \ f_2] \equiv f_1 U^{\leq \infty}_{> 0} f_2$$

$$AGf \equiv fu_{\geq 1}^{\leq\infty} false$$

$$AFf \equiv true\ U_{\geq 1}^{\leq\infty} f$$

$$EGf \equiv fu_{>0}^{\leq\infty} false$$

$$EFf \equiv true\ U_{>0}^{\leq\infty} f$$

2.3 Model Checking PCTL

[5] described a symbolic model checking algorithm for CTL that is based on fixed point calculation. The idea behind PCTL model checking is very similar. Instead of running the loop until a fixed point is reached, the number of iteration is bounded by the time bound. During the execution, the probability measure is accumulated. When the algorithm returns, the result is checked against the specification to determine if a property is satisfied. Here we will introduce the algorithm for the *strong until* operator. The algorithms for other operators can be devised likewise.

Definition 5 *[2] For a state $s \in S$ in a structure D and an integer $t \geq 0$, we define the function $\rho(t, s)$ to be the measure for the set of paths σ in $paths_s^D$ for which $\sigma \models_D f_1 U^{\leq t} f_2$. If $t < 0$, $\rho(t, s) = 0$.*

Theorem 6 *[2] If $t \geq 0$ then $\rho(t, s)$ satisfies the following recurrence equation:*

$$\begin{aligned}\rho(t,s) = &\ \textit{if } f_2 \in label(s) \textit{ then } 1\\ &\quad \textit{else if } f_1 \notin label(s) \textit{ then } 0\\ &\quad\quad \textit{else } \textstyle\sum_{s' \in S} \mathbf{P}(s,s') \times \rho(t-1,s')\end{aligned}$$

where $label(s)$ is the set of subformulas that are true in state s.

Algorithm 1
for $i := 0$ to t do
    for all $s \in S$ do
        if $f_2 \in label(s)$ then $\rho(i, s) := 1$
        else begin
            $\rho(i, s) := 0$;
            if $s_1 \in label(s)$ then for all $s' \in S$ do
                $\rho(i, s) := \rho(i, s) + \mathbf{P}(s, s') \times \rho(i-1, s')$
        end

The algorithm is straight forward. During the ith iteration, all the states that satisfy f_1 and reach a state that satisfies f_2 within i steps are identified, and their probability measures calculated.

When $t = \infty$, the above algorithm will not work. Instead, we first calculate the measure of $\sigma \models_D f_1 U^{\leq\infty} f_2$, denoted $\rho(\infty, s)$. Using a breadth first algorithm, we can identify below states:

- R: the set of success states. They are states labeled with f_2, or states for which the probability measure for eventually reaching a success state without passing a failure state is 1.
- Q: the set of failure states. They are states labeled by neither f_1 nor f_2, or states from which no success state is reachable.

We can then calculate $\rho(\infty, s)$ using the below formula:

$$
\begin{aligned}
\rho(\infty, s) = {} & \text{if } s \in R \text{ then } 1 \\
& \quad \text{else if } s \in Q \text{ then } 0 \\
& \qquad \text{else } \textstyle\sum_{s' \in S} \mathbf{P}(s, s') \times \rho(\infty, s')
\end{aligned}
$$

Theorem 7 *[2] $\rho(\infty, s')$ is the probabilistic measure for the set of paths σ in $paths_s^D$ for which $\sigma \models_D f_1 U^{\leq\infty} f_2$.*

In general, the complexity of the model checking algorithm is linear in the size of the formula, polynomial in the size of the state space, and linear in the time bound. More efficient algorithms can be used for formulas with special properties, such as $p = 0$ or $p = 1$.

3 Markov Decision Process and pCTL

Markov decision process, also called probabilistic-nondeterministic system (PNS), adds nondeterminism to DTMC to better model real world systems. In a DTMC, when making a transition, the system selects the next state according to a probability distribution. In a MDP, when making a transition, the system first asks an advisory for a probability distribution; the advisory randomly selects such a distribution from the (possibly infinite) set of available ones and returns it to the system; the system then selects a transition according to that distribution. The advisory's choice is nondeterministic and independent from past history.

3.1 Discrete Markov Decision Process

Definition 8 *[3] A Markov decision process is a quadruple $\Pi = (S, s_i, \tau, L)$, where:*

- *S is the denumerable or finite state space of the system*
- *$s_i \in S$ is the initial state*
- *τ is a function that associates with each $s \in S$ the set $\tau(s) = \{p_1^s, p_2^s, \cdots, p_{k_s}^s\}$, where k_s is the number of distribution choices at state s, p_i^s is a probability distribution*
- *L is a labeling function*

The probability measure of MDP is defined similar to DTMC. However, it is no longer possible to have one probability measure for a path due to the nondeterminism. Instead, there is a *maximal probability* $\mu_s^+(\Delta)$, and a *minimal probability* $\mu_s^-(\Delta)$. Δ is a set of paths starting at s. $\mu_s^+(\Delta)$ (resp. $\mu_s^-(\Delta)$) represents the probability that the system follows a path in Δ provided that the nondeterministic choices are as favorable (resp. unfavorable) as possible. In other words, $\mu_s^+(\Delta)$ is the least upper bound of measures of paths in Δ; $\mu_s^-(\Delta)$ the greatest lower bound. μ^+ and μ^- are not generally additive.

Lemma 9 *[3] Let Δ_1 and Δ_2 be set of paths starting from state s in a MDP, and $\Delta_1 \bigcap \Delta_2 = \emptyset$, then*
$\mu_s^-(\Delta_1 \bigcup \Delta_2) \geq \mu_s^-(\Delta_1) + \mu_s^-(\Delta_2)$
$\mu_s^+(\Delta_1 \bigcup \Delta_2) \geq \mu_s^+(\Delta_1) + \mu_s^+(\Delta_2)$

Lemma 10 *[3] Let Ω_s represent all the possible paths starting from s. $\Delta \subseteq \Omega_s$. Then*
$\mu_s^-(\Delta) = 1 - \mu_s^+(\Omega_s \setminus \Delta)$

3.2 pCTL

pCTL first appeared in [3]. It is used to model Markov decision process. It has similar semantics to CTL and PCTL. The only difference is how the probability measure is defined.

$s \models P_{\geq a}\phi$ iff $\mu_s^-(\{\omega \in \Omega_s \mid \omega \models \phi\}) \geq a$
$s \models P_{\leq a}\phi$ iff $\mu_s^+(\{\omega \in \Omega_s \mid \omega \models \phi\}) \leq a$

where $a \in [0, 1]$.

Intuitively, when checking the probability is at least a, we want to find the most unfavorable scenario, μ_s^-; and vice versa.

3.3 Model Checking pCTL

pCTL symbolic model checking is quite similar to PCTL. Since the probability measures are defined differently, we need to calculate them differently to account for the nondeterminism. Here, again, we will only talk about the *strong until* operator. Algorithms for other operators can be derived likewise.

To calculate the probability measure of $\phi = \gamma U \psi$, let S^{ψ} denote the states in which ψ hold. Let $S^{\phi}_{>0}$ denote all the states that satisfy ϕ, and $S'_p = S^{\phi}_{>0} - S^{\psi}$, we have:

$$\mu_s^- \phi = \min_{1 \le i \le k_s} [\sum_{t \in S'_p} p_i^s(t) \mu_t^- \phi + \sum_{t \in S^{\psi}} p_i^s(t)]$$

The formula is the same as the one for PCTL, except among all the possible results, we always pick the smallest one.

μ_s^+ is defined in a similar fashion.

$$\mu_s^+ \phi = \max_{1 \le i \le k_s} [\sum_{t \in S'_p} p_i^s(t) \mu_t^+ \phi + \sum_{t \in S^{\psi}} p_i^s(t)]$$

The complexity of the model checking algorithm is the same as PCTL, namely, linear in the size of the formula, polynomial in the size of the state space, and linear in the time bound of the formula.

4 Continuous-Time Markov Chain and CSL

In real world, time is continuous. Continuous time can be modeled using discrete-time system by slicing the time into shortest interval needed. Such approach, however, faces the state-explosion problem. To model continuous time directly, we need to use a continuous-time model, such as continuous-time Markov chain.

4.1 Continuous-Time Markov Chain

Continuous-time Markov chain has similar structure as discrete Markov chain. However, in a system where action can happen at any point in time, race conditions may exist. For example, while the system is making a transition between state s_2 and s_3, another transition to s_4 may occur before the first one

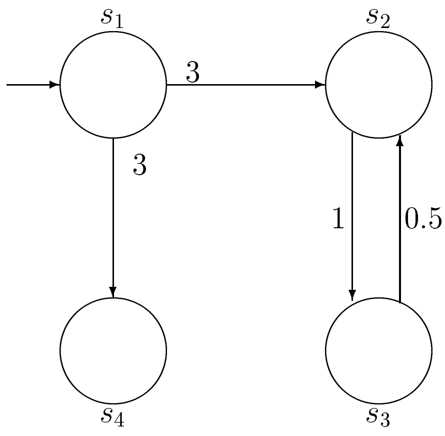

Fig. 2. A continuous time Markov chain modeling a probabilistic system

finishes. Note that there is no nondeterminism in CTMC. To reflect relation between continuous-time and probability, the transition probability is labeled differently in CTMC.

Definition 11 *[4]Let AP be a fixed, finite set of atomic propositions. A (labeled)* continuous-time Markov chain *(CTMC) is a tuple $M = (S, \mathbf{Q}, L)$ where*

- *S is a finite set of states*
- $\mathbf{Q} : S \times S \to R_{\geq 0}$ *the* generator matrix
- *L is the labeling function*

We use the example in [4] to demonstrate the working of CTMC, as shown in figure 2. In the model,

$$\mathbf{Q} = \begin{pmatrix} 0 & 3 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

The probability of moving from state s to s' within t time units (for positive t) is $1 - e^{-\mathbf{Q}(s,s')\cdot t}$. If s does not make any transition to other states, it is called an *absorbing state.* If $\mathbf{Q}(s, s') > 0$ for more than one s', a race condition is assumed to exist. Let $\mathbf{E}(s) = \sum_{s' \in S} \mathbf{Q}(s, s')$. It is not hard to see that the probability of leaving s within t time units is $1 - e^{-\mathbf{E}(s)\cdot t}$. Because of the possible race condition, we define a transition "wins", if it finishes before all the other transitions (each transition has a different start time and delay). Therefore, the probability of making a transition from state s to s' by a single transition, denoted by $\mathbf{P}(s, s')$, is $\mathbf{P}(s, s') = \mathbf{Q}(s, s')/\mathbf{E}(s)$. If s is an absorbing state, $\mathbf{E}(s) = 0$, $\mathbf{P}(s, s') = 0$.

A *path* in a CTMC is a (finite or infinite) sequence $s_0, t_0, s_1, t_1, \cdots$, denoted by

$$\sigma = s_0 \stackrel{t_0}{\rightarrow} s_1 \stackrel{t_1}{\rightarrow} s_2 \stackrel{t_2}{\rightarrow} \cdots$$

where $i \in N$, $s_i \in S$, $t_i \in R^+$, and $\mathbf{Q}(s_i, s_{i+1}) > 0$, if σ is infinite. If σ is finite, the last state, s_l is required to be absorbing. and $\mathbf{Q}(s_i, s_{i+1}) > 0$ for all $i < l$. $\sigma[i] = s_i$ denotes the ith state of σ; $\delta(t) = \sigma[i]$ denotes the time spent in state s_i; $\delta(t) = \sigma[i]$, where i is the smallest index of the state that takes t time units to reach, i.e. $t \leq \sum_{0 \leq j \leq i} t_j$; $\sigma(t) = s_0$ for $t < t_0$.

The probability measure of a set of paths in CTMC with prefix $\sigma = s_0 s_1 \cdots s_k$ is defined recursively by:

- $Pr(C(s_0)) = 1$
- $Pr(C(s_0, \cdots, s_k, I', s')) = Pr(C(s_0, \cdots, s_k)) \cdot \mathbf{P}(s_k, s') \cdot (e^{-\mathbf{E}(s_k) \cdot a} - e^{-\mathbf{E}(s_k) \cdot b})$ for $k > 0$

where I' is a nonempty interval in R^+, $a = inf I'$ (greatest lower bound) and $b = sup I'$ (least upper bound). If $b = \infty$ and $\mathbf{E}(s_k) > 0$, $e^{-\mathbf{E}(s_k) \cdot b} = 0$.

4.2 CSL

CSL first appeared in [6] for modeling continuous-time Markov chains. Later, it was extended by [7]. Below is the extended version.

Definition 12 *[7] For $a \in AP$, $p \in [0, 1]$ and $\bowtie \in \{\leq, <, \geq, >\}$, the state-formulas of* **CSL** *are defined by the grammar*

$$\Phi ::= true \mid a \mid \Phi \wedge \Phi \mid \neg \Phi \mid \mathcal{S}_{\bowtie p}(\Phi) \mid P_{\bowtie p}(\phi)$$

where $t \in R^+$, path-formulas are defined by

$$\phi ::= X\Phi | \Phi U \Phi \mid \Phi U^{\leq t} \Phi$$

$\mathcal{S}$ is the *steady state* operator, with the semantics:

$$s \models \mathcal{S}_{\bowtie p}(\Phi) \textit{ iff } \pi_{Sat(\Phi)}(s) \in I_{\bowtie p}$$

$I_{\bowtie p}$ is an interval, $[0, p)$, $[0, p]$, $(p, 1]$, or $[p, 1]$, depending on the value of $\bowtie$ and p. $Sat(\Phi) = \{s \in S \mid s \models \Phi\}$.

Let $S' \subseteq S$.

$$\pi_{S'}(s) = \lim_{t \to \infty} Pr\{\sigma \in Path(s) \mid \sigma(t) \in S'\}$$

The limit exists as a result of S being finite [8]. Intuitively, $\mathcal{S}_{\bowtie p}$ is satisfied if the probability measure of all paths, that start at state s and satisfy Φ within the time bound, is in $I_{\bowtie p}$.

Steady state operator is introduced for convenience. All other operators have the same semantics as PCTL and pCTL.

4.3 CSL Model Checking

To compute the probability for steady state formula, we need to introduce *bottom strongly connected component* (bscc). A subgraph B is bscc iff B is a strongly connected component and for all $s \in B$, the reachable states from s are also in B. Let $\diamond a_B$ be an atomic proposition such that $a_B \in L(s)$ iff $s \in B$. Let B be a bscc of S and $s, s' \in B$. Then $\pi_{s'} = Prob(s, \diamond a_B) \cdot \pi_B(s')$ where

$$\pi_B(s') = \begin{cases} 1 & \text{if } B = \{s'\} \\ \frac{\pi'_B(s')/\mathbf{E}(s')}{\sum_{s \in B} \pi'_B(s)/\mathbf{E}(s)} & \text{otherwise} \end{cases}$$

and $\pi'_B(s'')$ satisfies

$$\pi'_B(s'') = \textstyle\sum_{s \in B} \mathbf{P}(s, s'') \cdot \pi'_B(s) \text{ such that } \sum_{s \in B} \pi'_B(s) = 1.$$

Computing $P_{\bowtie p}(\phi)$ is similar to PCTL and pCTL when no time is involved. When time is involved, let $F(s, t)$ represent the probability of s satisfies $\Phi_1 U^{\leq t} \Phi_2$.

$$F(s,t) = \begin{cases} 1 & \text{if } s \models \Phi_2 \\ \sum_{s' \in S} \mathbf{Q}(s, s') \cdot \int_0^t e^{-\mathbf{E}(s) \cdot x} \cdot F(s', t - x) dx & \text{if } s \models \Phi_1 \wedge \neg \Phi 2 \\ 0 & otherwise \end{cases}$$

Note that due to the presence of the integral and exponent, it is unlikely we can find an exact answer. Numeric methods have been used to compute the approximate value.

Fig. 3. The Structure of PRISM

The complexity of CSL model checking is linear in the size of the formula and polynomial in the size of the state space. At the same time, the complexity also depends on the numeric method employed.

5 PRISM

5.1 Overview

PRISM is a probabilistic model checker developed at the University of Birmingham. This tool needs two input files, a system description file using prism language (a simple, state-based specification language) and a properties file either using PCTL [3] or using CSL. From the system description file, PRISM constructs one of the following models based on the model keyword (probabilistic, nondeterministic or stochastic) in that file:

(1) discrete-time Markov chain(DTMC),
(2) Markov decision process(MDP), or
(3) continuous-time Markov chain(CTMC).

The figure 3 illustrates the structure of the tool.

5.2 Implementation

PRISM uses the CUDD package to obtain BDD/MTBDD-based representation of probabilistic models. For PCTL model checking over DTMC or MDP models, PRISM implements the algorithm described in [2][3]. For CSL and CMTCs, the algorithm described in [6][4] is used. Graph traversal is implemented with BDD fixpoints, but numerical computation can be performed using one of three different model checking engines: symbolic MTBDD-based for both the model and the vector; sparse matrix model and full vectors; and hybrid which combines symbolic model and full vector.

PRISM performs model checking for the properties specified using PCTL or CSL and determines whether or not they are satisfied. Due to the way the model checking algorithms work, the tool actually checks whether each property is satisfied in *all* states of the model.

[3] Here, PCTL includes pCTL

Fig. 4. Abstraction of the Web server system

6 Verifying a Web server system with PRISM

6.1 System Description

We consider a Web server system with both hardware and software (application) failures described in [11]. The Web server software fails at rate γ_p, running on a machine (node) that fails independently at the rate γ_m. An automatic failure detection mechanism based on polling is installed. Assume that the mean time to detect server process failure is δ_p^{-1} and the mean time to detect machine failure is δ_m^{-1}. Furthermore, when the machine is detected to have failed, the server process is started on another machine, if available. The mean restart time of a machine is τ_m^{-1}. When only the server process is detected to have failed, it is automatically restarted on the same machine. The mean restart time of the server software is τ_p^{-1}. Typically, $\tau_p > \tau_m$. There is a small probability $1 - C$ that the process restart in the same machine is unsuccessful, in which case it is restarted on another machine, if available. Such a scheme of automatic restart after failures is also called *cold replication*. The Web server is considered available when the server process as well as the machine it is running on are up. We assume that no further failure can occur after a failure of either the process or the machine until it has been dealt with.

Here we define the system parameters in the following table:

γ_p	γ_m	δ_p	δ_m	τ_p	τ_m	$1-C$
0.001	0.0001	12	30	20	12	0.02

6.2 Observations

The above Web server system can be illustrated in figure 4. The left part is the system abstraction. The simple Web server system consists of two machines (primary server and spare server) with both application and hardware failures. detect_and_restart inspects application and/or hardware failure and repairs (restarts) them.

The right part is a rough state transition graph for applications and hardwares on both machines. Either applications or hardwares have three distinct states, OK, broken down, and restart. Their transition conditions (COND1, COND2_S/H and COND3) are described below:

(1) COND1 is just a failure rate. For applications, COND1 can be regarded as γ_p. For hardwares, COND1 can be regarded as γ_m.
(2) COND3 represents the mean time to restart the failed application and/or hardware. For applications, COND3 can be regarded as τ_p; for hardwares, τ_m.
(3) COND2_H can be regarded as the mean time for the detect_and_restart mechanism to detect the hardware failure. COND2_S is more complex, it is the mean time for the detect_and_restart to detect software failure plus
    (a) The software on the other machine is neithr working nor restarting, and
    (b) The hardware on the same machine is OK.

Therefore, to model the system correctly, we need to bear in mind the following important characteristics of the Web server system:

(1) The software status depends on the hardware status. That is, the software will crash immediately when the hardware is down and the software cannot restart before the hardware is working properly.
(2) There are chances that the hardware of both machines are down. In this case, the software cannot restart until one of them is restarted. If both of them recover from hardware failures at the same time, the choice of running software on which machine is nondeterministic.
(3) If only software fails on some machine, software will restart on the same machine with a small probability$(1 - C)$ of not being able to restart successfully. If the software restart is unsuccessful, the software on the other machine will restart if that machine is working, with the same small probability of not being able to restart successfully.

6.3 Model Construction

We describe the whole system with three modules: `primary`, `spare`, and `detect_and_restart`. `primary` and `spare` are two machines on which the Web server applications are installed. `detect_and_restart` module represents the automatic failure detection and fixing mechanism. It synchronizes with the `primary` and `spare` modules. See Appendix A for the full version of the corresponding Web server model described in prism language.

We use s1(s2), h1(h2), sr1(sr2), and hr1(hr2) to represent the primary(spare) software status, primary(spare) hardware status, primary(spare) software restart, primary(spare) hardware restart, respectively. The following table shows the meaning of these variables when they are in different values:

variable	*value* = 0	*value* = 1
s1	broken down	OK
s2	broken down	OK
h1	broken down	OK
h2	broken down	OK
sr1	not restarting	restarting
sr2	not restarting	restarting
hr1	not restarting	restarting
hr2	not restarting	restarting

Initially, primary machine and the Web server application on the primary machine are working. The spare machine is working too. But the Web server application on the spare machine is not working.

```
s1 : [0..1] init 1;
h1 : [0..1] init 1;

s2 : [0..1] init 0;
h2 : [0..1] init 1;
```

The restart values of both hardware and software in either machine are zero at the beginning.

```
sr1 : [0..1] init 0;
hr1 : [0..1] init 0;

sr2 : [0..1] init 0;
hr2 : [0..1] init 0;
```

Because the primary and spare machines have very similar behaviors, for discussion convenience, we just consider the primary machine. We'll mention the spare machine when necessary. The hardware and software on the primary machine suffer from failures with some rate. Note that when the hardware on which the software is running crashes, software cannot survive.

```
[] (s1 = 1) -> gamma_p : (s1' = 0);
[] (h1 = 1) -> gamma_m : (h1'= 0)&(s1' = 0);
```

The `primary` module synchronizes with the `detect_and_restart` module to inspect failures and recover from failures.

```
module primary
...

        [inspect_p_soft] (sr1 = 0)&(s1 = 0)&(h1 = 1)
                         &!((s2 = 1 )&(h2 = 1 ))&(sr2 = 0)
                         -> 1: (sr1' = 1);
        [restart_p_soft] (sr1 = 1)&(s1 = 0)
                         -> 0.98:(s1' = 1)&(sr1' = 0)
                            + 0.02:(sr1' = 0);

        [inspect_p_hard] (hr1 = 0)&(h1 = 0)
                         -> 1:(hr1' = 1);
        [restart_p_hard] (hr1 = 1)&(h1 = 0)
                         -> 1:(h1' = 1)&(hr1' = 0);
...
endmodule

module detect_and_restart
...

        [inspect_p_soft] (f=0) -> 12.0:(f'=1);
        [inspect_p_hard] (f=0) -> 30.0:(f'=1);

...

        [restart_p_soft] (f=1) -> 20.0:(f'=0);
        [restart_p_hard] (f=1) -> 12.0:(f'=0);

...
endmodule
```

Here we use the *synchronization* mechanism of PRISM. We label commands in different modules with action names. This can force two or more modules to make transitions simultaneously. As a common technique, we make one action (like [inspect_p_hard] (hr1 = 0)&(h1 = 0) $\rightarrow$ 1:(hr1' = 1);) *passive*, with rate 1 and one action (like [inspect_p_hard] (f=0) $\rightarrow$ 30.0:(f'=1);) *active*, which actually defines the rate for the synchronized transition. Note that in

```
        [inspect_p_soft] (sr1 = 0)&(s1 = 0)&(h1 = 1)
                         &!((s2 = 1 )&(h2 = 1 ))&(sr2 = 0)
                         -> 1: (sr1' = 1);
```

We guarantees that the software restart signal(sr1) cannot be set until the software is broken down(s1=0), the hardware on which it is running is OK(h1=1), the software on the spare is neither running correctly(!((s2 = 1)&(h2 = 1))) nor restarting(sr2 = 0). This guarantees the following characteristics:

(1) The software will not restart on a broken machine,
(2) The software will restart on the same machine if the hardware is working,
(3) The software does not need to restart if the software on the spare machine is working properly, and
(4) The software will not restart on the primary machine if the software on the spare machine is restarting.

6.4 Model Checking

We have model checked the following properties:

(1) Availability. For at least 99% of the time, the Web server is operational.
$S \geq 0.99[(s1 = 1 \wedge h1 = 1) \vee (s2 = 1 \wedge s2 = 1)]$

(2) Instantaneous availability. The probability that the Web server is operational at a given point of time (100 hours) is at least 0.99.
$P > 0.99[\Diamond^{[100,100]}(s1 = 1 \wedge h1 = 1) \vee (s2 = 1 \wedge s2 = 1)]$

(3) Liveness. The Web Server will eventually be operational.
$P \geq 1[\Diamond(s1 = 1 \wedge h1 = 1) \vee (s2 = 1 \wedge s2 = 1)]$

(4) The probability that Web server will go down in 1000 hours is less than 0.01.
$P < 0.01[\Diamond^{\leq 1000}(s1 = 0) \wedge (s2 = 0)]$

(5) If primary software is down, it will eventually recover.
$s1 = 0 \rightarrow P \geq 1[\Diamond s1 = 1 \wedge h1 = 1]$

(6) The probability that the Web server is not working in the 1, 10,100, 1000, 10000 time unit is less than 0.001.
  (a) $P < 0.001[\Diamond^{[1,1]} s1 = 0 \wedge s2 = 0]$
  (b) $P < 0.001[\Diamond^{[10,10]} s1 = 0 \wedge s2 = 0]$
  (c) $P < 0.001[\Diamond^{[100,100]} s1 = 0 \wedge s2 = 0]$
  (d) $P < 0.001[\Diamond^{[1000,1000]} s1 = 0 \wedge s2 = 0]$
  (e) $P < 0.001[\Diamond^{[10000,10000]} s1 = 0 \wedge s2 = 0]$

(7) There's at least a 99% chance that the Web server will stabilize such that it is up to more than 90% in the long run.
$P > 0.99[\Diamond S > 0.90[(s1 = 1 \wedge h1 = 1) \vee (s2 = 1 \wedge s2 = 1)]]$

(8) When the software of one machine is down, the probability that the Web server will be available within 0.6 time units is at least 0.99. This is an extension of property (5).
   (a) $s1 = 0 \wedge h1 = 1 \rightarrow P > 0.99[(s1 = 0 \wedge h1 = 1) \cup^{\leq 0.6} (s1 = 1 \wedge h1 = 1) \vee (s2 = 1 \wedge s2 = 1)]$
   (b) $s2 = 0 \wedge h2 = 1 \rightarrow P > 0.99[(s2 = 0 \wedge h2 = 1) \cup^{\leq 0.6} (s1 = 1 \wedge h1 = 1) \vee (s2 = 1 \wedge s2 = 1)]$
(9) If the primary hardware crashes, the probability that it will recover in 0.41 time units is greater than 0.9.
   $h1 = 0 \rightarrow P > 0.9[\diamond^{\leq 0.41} h1 = 1]$

The above properties are verified to be valid [4]. For the corresponding properties specification file, refer to Appendix B.

6.5 Model Statistics

The output of the model checking is given in the Appendix C. Here we explain some selected data. The time taken to construct the models using the Hybrid engine is 0.048 secs. There are two stages. Firstly, the system description is parsed and converted to the MTBDD representing it. Secondly, reachability is performed to identify non-reachable states and the MTBDD is filtered accordingly. Reachability is performed using a BDD based on the fixpoint algorithm. The number of iterations for this process is 9.

The table below shows statistics the Web server model we have built. The first section gives information about the CTMC representing the model: the number of states, the number of transitions(NNZ), the number of nodes and minterms. The second part refers to represent this CTMC using MTBDD engine: the time used to construct and the amount of memory needed to store it. The third part refers to represent this CTMC using Sparse engine: the time used to build the model and the memory needed. The last part refers to represent this CTMC using Hybrid(MTBDD + Sparse) engine: the time used

[4] Abbreviations:

- $\diamond \Phi = true \cup \Phi$ – eventually
- $\diamond^{\leq t} \Phi = true \cup^{\leq t} \Phi$ – within t time units

to build the model and the memory needed.

Model				MTBDD		Sparse		Hybrid	
states	NNZ	nodes	minterms	secs	kb	secs	kb	secs	kb
44	132	244	132	0.049	13.9	0.047	2.8	0.048	10.7

Steady State Probability Computation The table below gives the times taken to compute the steady state probabilities using the Power method. The number of iterations required and the time taken using the MTBDD, sparse and hybrid engines is as follows.

Time per iterations(secs):			Iterations
MTBDD	Sparse	Hybrid	
0.001669	0.000002	0.000015	10000

The time, memory and iterations taken using the Hybrid engine to check each of properties described in the section 6.4 is given in the following table:

property #	time(secs)	memory(kb)	iterations
1	0.03	10.7	244
2	0.04	11.0	79
3	0.07	9.7	12
4	0.116	-	10000
5	0.05	-	16
6a	0.05	11.0	233
6b	0.015	11.0	824
6c	0.107	11.0	6786
6d	0.156	11.0	10000
6e	0.156	11.0	10000
7	0.012	10.7	244
8a	0.002	5.2	45
8b	0.002	6.5	45
9	0.002	8.1	76

7 Evaluation

7.1 Lessons Learned

Throughout modeling the Web server system, the lessons we've learned can be summarized as follows.

(1) **Use as less variables as possible to modeling the system**. Because each command specifies a state in the model system and its associated transition. We have to know all the relevant states in order to model the system correctly. This is fine for a simple system like we modeled(with at most 4 variables in each module, each variable has only 2 values: 0 and 1). However, if the system becomes more complex and therefore needs more variables, it would be difficult for us to trace every meaningful state and its relevant transition without writing down a table. In this case, we have to think over all the possible combinations.

To cut down the number of variable and hence reduce the difficulty in modeling, it's better to write the properties specification first. Then we try our best to use the variables in the properties specification file only to model the system.

(2) **Combining global variables with synchronization is not good**. At first, we used a *global* variable `flag`, with an initial value of 0, as well as the embedded synchronization mechanism of PRISM, to model the behavior of the Web server system. If $flag = 0$, primary server is on(will be on) service. Otherwise, spare server is on(will be on) service. The `flag` variable changes its value only when hardware failures happen.

In this way, we were unable to verify the properties to be valid for a long time. After examining the PRISM manual [15] more carefully, we noticed that if we use the synchronization mechanism (labeling two or more commands in different modules with the same transition name) , we should avoid use the global variables at the same time. Combining the global variables and synchronization mechanism will result in a not well defined system. We think this is a weak point of PRISM.

(3) **To use synchronization, make one command passive and the other active**. See the following example:

```
module primary
...
      [inspect_p_soft] (sr1 = 0)&(s1 = 0)&(h1 = 1)
                       &!((s2 = 1 )&(h2 = 1 ))&(sr2 = 0)
```

```
                          -> 1: (sr1' = 1);
...
endmodule
...
module detect_and_restart
...

        [inspect_p_soft] (f=0) -> 12.0:(f'=1);
...
endmodule
```

The product of the two rates does not always meaningfully represent the rate of a synchronized transition. Here we make one transition passive with rate 1, and another active, with a rate of 12. By this technology, the product of the two rates (in this case, 1×12) can be regarded as meaningful [15].

(4) **Constructing a corresponding continuous-time Markov Chain(C TMC) manually is *not* necessary**. At the beginning, we tried to draw a CTMC manually. Then we used one variable s with a value ranged from 0 to N (number of states in the CTMC). Each state is a meaningful combination of different values of variables. Later, after examining more examples, we realized that PRISM can construct a CTMC automatically according to the system description.

7.2 What Is PRISM Good For And What It Is Bad For?

PRISM(**Pr**obabilist**i**c **S**ymbolic **M**odel Checker), as the name indicates, can be used to check probabilistic systems. It supports three models(discrete-time Markov chains, continuous-time Markov chains, Markov decision processes) and two specification languages(PCTL and CSL). The PRISM modeling language is a simple, state-based language. Therefore, it is easy to learn.

However, at this point, we think PRISM has several weak points. One example is that PRISM does not support LTL and/or $PCTL^*$. If PRISM supports LTL and/or $PCTL^*$, the properties specification would become more expressive. Another improvement is that PRISM should provide counterexamples for model checking, like nusmv. Counterexamples are very useful since they help us to model the target system correctly. Also, we think PRISM should support abstraction mechanism, which is successful in non-probabilistic model checking [14]. Lastly, the output of PRISM can be refined. It is highly desirable if PRISM can draw a state transition graph automatically like Xspin. Also, PRISM can consider adding a simulation function.

7.3 Modeling Decisions

Since the Web server system is a simple, we tried to model the system as accurately as possible. Here is a list of functionality that we chose to model:

(1) Software and hardware will fail at some rate, independently.
(2) Software and hardware failure will be detected and repaired, at some rate.
(3) If the hardware is down, the software on the same machine will unconditionally be down.
(4) Software cannot restart before the hardware is recovered.
(5) If hardwares on both machines are down, software will restart on the machine whose hardware recovers earlier.
(6) If both failed hardwares recover at the same time, software restart on which machine is nondeterministic.
(7) Software recovering on a machine suffers from a small probability $(1-C)$ of being unsuccessful.
(8) Unsuccessful software restarting will cause another machine to restart the software, if the hardware on that machine is in good condition.

One thing that we leave for future improvement is that if the software on the primary machine is down and the hardware on this machine is OK, software will restart on the primary machine at first time, not on the spare machine, even if the spare machine hardware is available. This is exactly what the *System Description* section requires. However, in our model, if the software on the primary machine is down, the software on the spare machine have chances to restart, if the hardware on that machine is available. However, this does not hurt the availability/performance of the Web server system.

7.4 Comments On Modeling Language

PRISM language is easy to learn and to use. First we declare the type of the system with one of the key words: probabilistic, nondeterministic and stochastic. Then we define several modules beginning with `module MODULE_NAME` and concluding with `endmodule`. In each module, we use local variables to represent states. The behavior of each model is described by a set of commands of the form: $[]g \rightarrow \lambda_1\mu_1 + \cdots + \lambda_n\mu_n$. Updates μ_i will take place if the guard g is true. The [] before g is for synchronization purpose. Also, synchronization can be done by referring to the variables of other modules or global variables. However, a module can not change the values of other modules except the values of global variables.

7.5 Scalability

We think RRISM is highly scalable. In [17], the model is represented as a CTMC which consists of a **M/Cox_2/1-queue** sequentially composed with a **M/M/1-queue**. **N** is used to denote the capacity of the queues. We can scale N from 5 to 8191 by just modifying one line:

```
...
const c = 5; // queue capacity
...
```

Another example that shows PRISM is highly scalable is in [16]. This case study is based on Pnueli and Zuck's randomized solution to the well known dining philosophers problem. The model is represented as an MDP. Let **N** denote the number of philosophers. We can scale **N** from 3 to 35 easily by adding another 32 modules by renaming.

```
...
module phil1 ... endmodule
// construct further modules through renaming

module phil2 = phil1 [p1=p2, p2=p3, p3=p1] endmodule
module phil3 = phil1 [p1=p3, p2=p1, p3=p2] endmodule
module phil4 ...
...
module phil35= phil1 [p1=p35, p2=p1,  p35=p34] endmodule
```

7.6 PRISM vs. Other Model Checkers

There are many model checkers available in the public domain. Each is designed with a specific goal in mind; therefore, some model checkers are more suitable for one task than others. Here is a brief comparison between PRISM and some other model checkers.

PRISM vs. Spin Spin takes an automata-theoretic approach to model checking and is highly efficient for checking linear time properties. However, Spin does not have a sense of time and is not built to model probabilistic systems. Using variables, time and probability can be simulated in Spin, although the programming may be tedious and time-consuming. PRISM, on the other hand, does not handle linear temporal properties, but can model probabilistic systems with ease. Both model checkers are highly scalable.

PRISM vs. NuSmv NuSmv implements symbolic model checking algorithms using BDDs. It can model discrete time; we can consider every transition in

NuSmv to take one time unit. While NuSmv can model check both CTL and LTL, there is no direct way to check probabilistic properties. At the same time, NuSmv uses a switching statements whose conditions are checked sequentially. We feel that this limits the expressive power of the modeling language.

PRISM vs. Uppaal Uppaal [18] is a conventional model checker that focuses on model checking real time systems. It can directly model both discrete and continuous time. Uppaal uses algorithms for timed-automata to optimize its performance. It also provides a Graphical User Interface that allows user to specify a system both graphically and textually. This greatly enhances its usability. However, Uppaal cannot check linear time properties.

PRISM vs. E $\vdash MC^2$ $E \vdash MC^2$ [19], also called Erlangen-Twente Markov Chain Model Checker, is developed at Institut für Informatik, Friedrich-Alexander-Universität, in Germany. Similar to PRISM, $E \vdash MC^2$ is a probabilistic model checker. It does not explicitly differentiate between CSL and PCTL, since any properties that can be expressed in PCTL can be expressed in CSL. $E \vdash MC^2$ does not use MTBDD, instead, it uses sparse matrices. $E \vdash MC^2$ is implemented using java, and is designed to have maximum portability and extensibility. To achieve this goal, $E \vdash MC^2$ does not have low level optimization. Therefore, $E \vdash MC^2$ is not suitable for model checking large systems.

8 Conclusion

In this report, we have described three probabilistic models and their corresponding model checking languages. Different from conventional model checking, with probability model checking, there is no need to worry about if the truth value of a proposition or transition is known. It can be represented by a confidence interval. Moreover, the complexity of the computation remains the same.

However, probabilistic symbolic model checking has limitations with linear time logic. PCTL* and pCTL* model checking algorithms, to our knowledge, all involve expensive conversion to branching time logic.

Despite the limitation, probabilistic model checking can be applied to a wide range of systems. We chose PRISM as the verification tool in our report. The modeling and checking of a probabilistic system using PRISM is straight forward. While PRISM is not the best model checker for every system, it provides great usability for checking branching time logic in probabilistic (discrete and continuous time) systems.

A Appendix: The System Model

Program .1. webserver.sm

```
/////////////////////////////////////////////////////////
//
// Web server with both hardware and software failures.
// Assumption: no further failures can occur after a
// failure of either the process or the machine until
// it has been dealt with.
//
// Created on Dec 27, 2003
// Last modified on Jan 1, 2004
//
// Some system parameters(time unit: hour):
//   * Prob(software restart unsucessfully): 0.02
//   * Software failure rate 0.001
//   * Hardware failure rate 0.0001
//   * Mean time to inspect software failure : 1/12
//   * Mean time to inspect hardware failure : 1/30
//   * Mean time to restart software: 1/20
//   * Mean time to restart hardware: 1/12
//
/////////////////////////////////////////////////////////

stochastic

rate  gamma_p = 0.001; // failure rate of the software
rate  gamma_m = 0.0001;// failure rate of the hardware

module primary

// Software status  1-working, 0-broken
       s1 : [0..1] init 1;
// Hardware status  1-working, 0-broken
       h1 : [0..1] init 1;
// Software's recovering?
       sr1 : [0..1] init 0;
// Hardware's recoviering?
       hr1 : [0..1] init 0;

// Software and hardware on the primary machine fail at
// some rate, respectively.
// Observations: (1).Software will fail immediately
//               after the hardware fails; and
//               (2).Software cannot restart until the
//               hardware is working.

       [] (s1 = 1) -> gamma_p : (s1' = 0);
       [] (h1 = 1) -> gamma_m : (h1'= 0)&(s1' = 0);
```

```
// If hardware is ok and the software is down, restart
// the software on the same machine.
// Observaton: (1).If software and hardware are ok on
//             one machine, the other machine doesn't
//             need to restart the software
// Assumption: when the software is restarting, the
//             hardware cannot fail

       [inspect_p_soft] (sr1 = 0)&(s1 = 0)&(h1 = 1)
                        &!((s2 = 1 )&(h2 = 1 ))&(sr2 = 0)
                        -> 1: (sr1' = 1);
       [restart_p_soft] (sr1 = 1)&(s1 = 0)
                        -> 0.98:(s1' = 1)&(sr1' = 0)
                           + 0.02:(sr1' = 0);

       [inspect_p_hard] (hr1 = 0)&(h1 = 0)
                        -> 1:(hr1' = 1);
       [restart_p_hard] (hr1 = 1)&(h1 = 0)
                        -> 1:(h1' = 1)&(hr1' = 0);

endmodule

// Similar to the primary module except that the
// initial software status is different.
module spare

// Software status  1-working, 0-broken
       s2 : [0..1] init 0;
// Hardware status  1-working, 0-broken
       h2 : [0..1] init 1;
// Software's recovering
       sr2 : [0..1] init 0;
// hardware's recovering
       hr2 : [0..1] init 0;

       [] (s2 = 1) -> gamma_p :(s2' = 0);
       [] (h2 = 1) -> gamma_m :(h2' = 0)&(s2' = 0);

       [inspect_s_soft] (sr2 = 0)&(s2 = 0)&(h2 = 1)
                        &!((s1 =1 )&(h1 =1))&(sr1 = 0)
                        -> 1: (sr2' = 1);
       [restart_s_soft] (sr2 = 1)&(s2 = 0)
                        -> 0.98: (s2' = 1)&(sr2' = 0)
                           + 0.02:(sr2' = 0);

       [inspect_s_hard] (hr2 = 0)&(h2 = 0)
                        -> 1: (hr2' = 1);
       [restart_s_hard] (hr2 = 1)&(h2 = 0)
                        -> 1: (h2' = 1)&(hr2'=0);

endmodule
```

```
// Failure detecting and repairing module, for
// synchronisation with the primary and spare module.
module detect_and_restart

       f:[0..1] init 0;

       [inspect_p_soft] (f=0) -> 12.0:(f'=1);
       [inspect_p_hard] (f=0) -> 30.0:(f'=1);
       [inspect_s_soft] (f=0) -> 12.0:(f'=1);
       [inspect_s_hard] (f=0) -> 30.0:(f'=1);

       [restart_p_soft] (f=1) -> 20.0:(f'=0);
       [restart_p_hard] (f=1) -> 12.0:(f'=0);
       [restart_s_soft] (f=1) -> 20.0:(f'=0);
       [restart_s_hard] (f=1) -> 12.0:(f'=0);

endmodule
```

B Appendix: Specification in CSL

Program .2. webserver.csl

```
// (1). Availability: for at least 99% of the time, the
//  system is operational on the long run .

S>=0.99[(s1 = 1 & h1 = 1)|(s2 = 1 & h2 = 1)]

// (2). Instantaneous availability: the probability that
// the system is operational at a given  point of time
// (100 hours) is at least 0.99.

P>0.99[true U[100,100] (s1 = 1 & h1 = 1)|(s2 = 1 & h2 = 1)]

// (3). The system will eventually be operational.

P>=1[true U (s1 = 1 & h1 = 1)|(s2 = 1 & h2 = 1)]

// (4). The probability that system will go down in 1000
// hours is less than 0.01.

P<0.01[true U<=1000 (s1 = 0)&(s2 = 0)]

// (5). If primary software is down, it will eventually
// recover

s1 = 0 -> P>=1[true U s1 = 1 & h1 = 1]

// (6). The probability that the whole system is not
```

```
// working in the 10,100, 1000, 10000 time unit is less
// than 0.001

P<0.001[true U[0.1,0.1] s1 = 0 & s2 = 0]
P<0.001[true U[1,1] s1 = 0 & s2 = 0]
P<0.001[true U[10,10] s1 = 0 & s2 = 0]
P<0.001[true U[100,100] s1 = 0 & s2 = 0]
P<0.001[true U[1000,1000] s1 = 0 & s2 = 0]
P<0.001[true U[10000,10000] s1 = 0 & s2 = 0]

// (7). There's at least a 99% chance that the system
// will stabilize such that it is up to more than 90% in
// the long run.

P>0.99[true U S>0.90[(s1 = 1 & h1 = 1)|(s2 = 1 & h2 = 1)]]

// (8). When the software of one machine is down, the
// probability that the system will be available within 0.6
// time units is at least 0.99. This is an extension of
// property (5).

s1 = 0 & h1 = 1 -> P>0.99[(s1 = 0 & h1 = 1) U<=0.6
                    (s1 =1 & h1 = 1)|(s2 = 1 & s2 = 1)]
s2 = 0 & h2 = 1 -> P>0.99[(s2 = 0 & h2 = 1) U<=0.6
                    (s1 =1 & h1 = 1)|(s2 = 1 & s2 = 1)]


// (9). If the primary hardware crashes, the probability
// that it will recover in 0.41 time units is greater than
// 0.99

h1 = 0 -> P>0.9[true U<=0.41 h1 = 1]
```

C Appendix: Output

Program .3. webserver.out

```
PRISM =====

Version: 1.3.1 Date: Fri Jan 02 13:49:47 EST 2004 Command
line: prism webserver.sm webserver.csl

Loading model file "webserver3.sm"...

Building model...

MTBDD variables used (9r, 9c): s1.0 s1'.0 h1.0 h1'.0
sr1.0 sr1'.0 hr1.0 hr1'.0 s2.0 s2'.0 h2.0 h2'.0 sr2.0
sr2'.0 hr2.0 hr2'.0 f.0 f'.0
```

```
Computing reachable states...

Reachability: 9 iterations in 0.00 seconds (average
0.000000, setup 0.00)

Model built in 0.048 seconds.

Type: Stochastic (CTMC) Modules: primary spare
detect_and_restart Variables: s1 h1 sr1 hr1 s2 h2 sr2 hr2
f

States: 44 Transitions: 132

Rate matrix: 244 nodes (7 terminal), 132 minterms ODD: 37
nodes

Loading properties file "webserver.csl"...

15 PCTL/CSL formulas:

(1) S>=0.99 [ (s1=1 & h1=1) | (s2=1 & h2=1) ]
(2) P>0.99  [ true U[100.0,100.0] (s1=1 & h1=1) |
    (s2=1 & h2=1) ]
(3) P>=1.0 [ true U (s1=1 & h1=1) | (s2=1 & h2=1) ]
(4) P<0.01 [ true U<=1000.0 (s1=0) & (s2=0) ]
(5) s1=0 => P>=1.0 [ true U s1=1 & h1=1 ]
(6) P<0.0010 [ true U[0.1,0.1] s1=0 & s2=0 ]
(7) P<0.0010 [ true U[1.0,1.0] s1=0 & s2=0 ]
(8) P<0.0010 [ true U[10.0,10.0] s1=0 & s2=0 ]
(9) P<0.0010 [ true U[100.0,100.0] s1=0 & s2=0 ]
(10) P<0.0010 [ true U[1000.0,1000.0] s1=0 & s2=0 ]
(11) P<0.0010 [ true U[10000.0,10000.0] s1=0 & s2=0 ]
(12) P>0.99 [ true U S>0.9 [ (s1=1 & h1=1)
     | (s2=1 & h2=1) ] ]
(13) s1=0 & h1=1 => P>0.99 [ (s1=0 & h1=1) U<=0.6 (s1=1 &
     h1=1) | (s2=1 & s2=1) ]
(14) s2=0 & h2=1 => P>0.99 [(s2=0 & h2=1) U<=0.6
    (s1=1 & h1=1) | (s2=1 & s2=1) ]
(15) h1=0 => P>0.9 [ true U<=0.41 h1=1 ]

Model checking property (1)...

(1) S>=0.99 [ (s1=1 & h1=1) | (s2=1 & h2=1) ]

Engine: Hybrid

CSL Steady State:

b = 11 states
```

```
Computing (B)SCCs...  SCCs: 1 BSCCs: 1

Computing steady state probabilities for BSCC 1

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [2 x 0.3 KB] TOTAL: [10.7 KB]

Starting iterations...

Jacobi: 244 iterations in 0.00 seconds (average 0.000000,
setup 0.00)

BSCC 1 Probability: 0.9998899344549256

All states are in a BSCC (so no reachability
probabilities computed)

Number of states satisfying steady-state operator: 44

Model checking completed in 0.03 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (2)...

(2) P>0.99 [ true U[100.0,100.0] (s1=1 & h1=1) | (s2=1 &
h2=1) ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 11 states

Number of non-absorbing states: 44 of 44 (100.0%)

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 100.000000 =
6000.200000 Fox-Glynn: left = 5648, right = 6786

Starting iterations...

Steady state detected at iteration 79

Iterative method: 79 iterations in 0.00 seconds (average
```

```
0.000000, setup 0.00)

Number of states satisfying time bounded until: 44

Model checking completed in 0.0040 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (3)...

(3) P>=1.0 [ true U (s1=1 & h1=1) | (s2=1 & h2=1) ]

Engine: Hybrid

CSL Until:

b1 = 44 states, b2 = 11 states

Warning: probability bound in formula is 1.0 so exact
probabilities may not be computed

Prob1: 12 iterations in 0.01 seconds (average 0.000833,
setup 0.00)

yes = 44, no = 0, maybe = 0

Number of states satisfying until: 44

Model checking completed in 0.0070 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (4)...

(4) P<0.01 [ true U<=1000.0 (s1=0) & (s2=0) ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 28 of 44 (63.6%)

Building hybrid MTBDD matrix... [nodes=299] [7.0 KB]
Adding sparse bits... [levels=3, bits=31] [1.3 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [9.7 KB]

Uniformisation: q.t = 60.002000 x 1000.000000 =
60002.000000 Fox-Glynn: left = 59715, right = 63303
```

```
Starting iterations...

Warning: Iterative method stopped early at 10000
iterations.

Iterative method: 10000 iterations in 0.12 seconds
(average 0.000012, setup 0.00)

Number of states satisfying time bounded until: 44

Model checking completed in 0.116 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (5)...

(5) s1=0 => P>=1.0 [ true U s1=1 & h1=1 ]

Engine: Hybrid

CSL Until:

b1 = 44 states, b2 = 6 states

Warning: probability bound in formula is 1.0 so exact
probabilities may not be computed

Prob1: 16 iterations in 0.00 seconds (average 0.000000,
setup 0.00)

yes = 44, no = 0, maybe = 0

Number of states satisfying until: 44

Model checking completed in 0.0050 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (6)...

(6) P<0.0010 [ true U[0.1,0.1] s1=0 & s2=0 ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 44 of 44 (100.0%)
```

```
Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 0.100000 = 6.000200
Fox-Glynn: left = 0, right = 178

Starting iterations...

Iterative method: 178 iterations in 0.01 seconds (average
0.000056, setup 0.00)

Number of states satisfying time bounded until: 28

Model checking completed in 0.0050 seconds.

Number of states satisfying PCTL/CSL formula: 28
(including initial state)

Model checking property (7)...

(7) P<0.0010 [ true U[1.0,1.0] s1=0 & s2=0 ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 44 of 44 (100.0%)

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 1.000000 = 60.002000
Fox-Glynn: left = 17, right = 233

Starting iterations...

Iterative method: 233 iterations in 0.00 seconds (average
0.000000, setup 0.00)

Number of states satisfying time bounded until: 44

Model checking completed in 0.0050 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)
```

```
Model checking property (8)...

(8) P<0.0010 [ true U[10.0,10.0] s1=0 & s2=0 ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 44 of 44 (100.0%)

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 10.000000 = 600.020000
Fox-Glynn: left = 461, right = 824

Starting iterations...

Iterative method: 824 iterations in 0.02 seconds (average
0.000012, setup 0.01)

Number of states satisfying time bounded until: 44

Model checking completed in 0.015 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (9)...

(9) P<0.0010 [ true U[100.0,100.0] s1=0 & s2=0 ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 44 of 44 (100.0%)

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 100.000000 =
6000.200000 Fox-Glynn: left = 5648, right = 6786
```

```
Starting iterations...

Iterative method: 6786 iterations in 0.11 seconds
(average 0.000015, setup 0.01)

Number of states satisfying time bounded until: 44

Model checking completed in 0.107 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (10)...

(10) P<0.0010 [ true U[1000.0,1000.0] s1=0 & s2=0 ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 44 of 44 (100.0%)

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 1000.000000 =
60002.000000 Fox-Glynn: left = 59715, right = 63303

Starting iterations...

Warning: Iterative method stopped early at 10000
iterations.

Iterative method: 10000 iterations in 0.16 seconds
(average 0.000016, setup 0.00)

Number of states satisfying time bounded until: 44

Model checking completed in 0.156 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (11)...

(11) P<0.0010 [ true U[10000.0,10000.0] s1=0 & s2=0 ]

Engine: Hybrid
```

```
CSL Time Bounded Until:

b1 = 44 states, b2 = 16 states

Number of non-absorbing states: 44 of 44 (100.0%)

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [11.0 KB]

Uniformisation: q.t = 60.002000 x 10000.000000 =
600020.000000 Fox-Glynn: left = 607323, right = 618660

Starting iterations...

Warning: Iterative method stopped early at 10000
iterations.

Iterative method: 10000 iterations in 0.15 seconds
(average 0.000015, setup 0.00)

Number of states satisfying time bounded until: 44

Model checking completed in 0.156 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (12)...

(12) P>0.99 [ true U S>0.9 [ (s1=1 & h1=1) | (s2=1 &
h2=1) ] ]

Engine: Hybrid

CSL Steady State:

b = 11 states

Computing (B)SCCs...  SCCs: 1 BSCCs: 1

Computing steady state probabilities for BSCC 1

Building hybrid MTBDD matrix... [nodes=337] [7.9 KB]
Adding sparse bits... [levels=3, bits=42] [1.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [2 x 0.3 KB] TOTAL: [10.7 KB]

Starting iterations...

Jacobi: 244 iterations in 0.00 seconds (average 0.000000,
```

```
setup 0.00)

BSCC 1 Probability: 0.9998899344549256

All states are in a BSCC (so no reachability
probabilities computed)

Number of states satisfying steady-state operator: 44

CSL Until:

b1 = 44 states, b2 = 44 states

Prob1: 4 iterations in 0.01 seconds (average 0.002500,
setup 0.00)

yes = 44, no = 0, maybe = 0

Number of states satisfying until: 44

Model checking completed in 0.012 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (13)...

(13) s1=0 & h1=1 => P>0.99 [ (s1=0 & h1=1) U<=0.6 (s1=1 &
h1=1) | (s2=1 & s2=1) ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 11 states, b2 = 21 states

Number of non-absorbing states: 7 of 44 (15.9%)

Building hybrid MTBDD matrix... [nodes=137] [3.2 KB]
Adding sparse bits... [levels=3, bits=15] [0.7 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [5.2 KB]

Uniformisation: q.t = 42.000100 x 0.600000 = 25.200060
Fox-Glynn: left = 6, right = 197

Starting iterations...

Steady state detected at iteration 45

Iterative method: 45 iterations in 0.00 seconds (average
0.000000, setup 0.00)
```

```
Number of states satisfying time bounded until: 28

Model checking completed in 0.0020 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (14)...

(14) s2=0 & h2=1 => P>0.99 [ (s2=0 & h2=1) U<=0.6 (s1=1 &
h1=1) | (s2=1 & s2=1) ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 11 states, b2 = 21 states

Number of non-absorbing states: 10 of 44 (22.7%)

Building hybrid MTBDD matrix... [nodes=193] [4.5 KB]
Adding sparse bits... [levels=3, bits=14] [0.6 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [6.5 KB]

Uniformisation: q.t = 42.001100 x 0.600000 = 25.200660
Fox-Glynn: left = 6, right = 197

Starting iterations...

Steady state detected at iteration 45

Iterative method: 45 iterations in 0.00 seconds (average
0.000000, setup 0.00)

Number of states satisfying time bounded until: 31

Model checking completed in 0.0020 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)

Model checking property (15)...

(15) h1=0 => P>0.9 [ true U<=0.41 h1=1 ]

Engine: Hybrid

CSL Time Bounded Until:

b1 = 44 states, b2 = 17 states
```

```
Number of non-absorbing states: 27 of 44 (61.4%)

Building hybrid MTBDD matrix... [nodes=232] [5.4 KB]
Adding sparse bits... [levels=3, bits=31] [1.3 KB]
Creating vector for diagonals... [0.3 KB] Allocating
iteration vectors... [3 x 0.3 KB] TOTAL: [8.1 KB]

Uniformisation: q.t = 60.002000 x 0.410000 = 24.600820
Fox-Glynn: left = 6, right = 197

Starting iterations...

Steady state detected at iteration 76

Iterative method: 76 iterations in 0.00 seconds (average
0.000000, setup 0.00)

Number of states satisfying time bounded until: 44

Model checking completed in 0.0020 seconds.

Number of states satisfying PCTL/CSL formula: 44 (all)
```

References

[1] M. Kviatkowska. Model Checking for Probability and Time: From Theory to Practice. Invited paper, in *Proc. 18th IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 351-360, IEEE Computer Society Press, June 2003.

[2] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512-535, 1994.

[3] A. Bianco and L. Alfaro. Model Checking of Probabilistic and Nondeterministic systems. In P. Thiagarajan, editor, *Proc. 15h Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499-513. Sprinter, 1995.

[4] C. Beier, J. 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 *LINCS*, pages 146-161. Sprinter, 1999.

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

[6] A. Aziz, K. Sansal, V. Singhal and R. Brayton. Verifying Continuous Time Markov Chains. In *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269-276. Springer, 1996.

[7] C. Baier, J. 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 *LINCS*, pages 146-161. Springer, 1999.

[8] W. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Priceton Univ. Press, 1994.

[9] M. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc. 1994.

[10] M. Kwiatkowska, G. Norman and J. Sproston. PCTL Model Checking of Symbolic Probabilistic Systems. *Technical Report CSR-03-2*, School of Computer Science, University of Birmingham, April 2003.

[11] S. Garg, Y. Huang, C.M. Kintala, K.S. Trivedi, and S. Yajnik. Performance and reliability evaluaion of passive replication schems in application level fault tolerance. In *Proc. 29th Annual Int. Symp. Fault Tolerant Computing(FTCS)*, Madison, Wisconsin, pp 15-18, June 15-18, 1999

[12] http://www.cs.bham.ac.uk/~dxp/prism/casestudies/index.html

[13] A. Gurfinkel and M. checkik. Multi-Valued Model Checking via Classical Model Checking. In *Proceedings of CONCUR'03*, LNCS2761, September 2003.

[14] D. Parker, Implementation of Symbolic Model Checking for Probabilistic Systems, Ph.D. Thesis, University of Birmingham, August 2002.

[15] Daver Parker, PRISM 1.3 Users' Guide, Feb., 2003.

[16] http://www.cs.bham.ac.uk/~dxp/prism/casestudies/phil.html

[17] http://www.cs.bham.ac.uk/~dxp/prism/casestudies/tandem.html

[18] http://www.uppaal.com/

[19] http://www7.informatik.uni-erlangen.de/etmcc/publications.html