PRTS: An Approach for Model Checking Probabilistic Real-time Hierarchical Systems[⋆]

Jun Sun[1], Yang Liu[2], Songzheng Song[3], Jin Song Dong[2] and Xiaohong Li[4]

[1] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] National University of Singapore
{liuyang,dongjs}@comp.nus.edu.sg
[3] NUS Graduate School for Integrative Sciences and Engineering
songsongzheng@nus.edu.sg
[4] School of Computer Science and Technology, Tianjin University
xiaohongli@tju.edu.cn

Abstract. Model Checking real-life systems is always difficult since such systems usually have quantitative timing factors and work in unreliable environment. The combination of real-time and probability in hierarchical systems presents a unique challenge to system modeling and analysis. In this work, we develop an automated approach for verifying probabilistic, real-time, hierarchical systems. Firstly, a modeling language called PRTS is defined, which combines data structures, real-time and probability. Next, a zone-based method is used to build a finite-state abstraction of PRTS models so that probabilistic model checking could be used to calculate the probability of a system satisfying certain property. We implemented our approach in the PAT model checker and conducted experiments with real-life case studies.

1 Introduction

With the development of computing and sensing technology, information process and control software are integrated into everyday objects and activities. Design and development of control software for real-life systems are notoriously difficult problems, because such systems often have complex data components or complicated hierarchical control flows. Furthermore, control software often interacts with physical environment and therefore depends on quantitative timing. In addition, probability exhibits itself commonly in the form of statistical estimates regarding the environment in which control software is embedded. Requiring a system always to function perfectly within any environment is often overwhelming. Standard model checking may produce ‘unlikely’ counterexamples which may not be helpful.

Example 1 (A motivating example). Multi-lift systems heavily rely on control software. A multi-lift system consists of a hierarchy of components, e.g., the system contains multiple lifts, floors, users, etc.; a lift contains a panel of buttons, a door and a lift

⋆ This research was partially supported by research grant “SRG ISTD 2010 001” from Singapore University of Technology and Design.

controller; a lift controller may contain multiple control units. It is complex in control logic as behavior of different components must be coordinated through a software controller. Ideally, the system shall be formally verified to satisfy desirable properties. For instance, one of the properties is: *if a user has requested to travel in certain direction, a lift should* not *pass by, i.e., traveling in the same direction without letting the user in*. However, this property is not satisfied. Typically, once a user presses a button on the external panel at certain floor, the controller assigns the request to the '*nearest*' lift. If the '*nearest*' lift is not the first reaching the floor in the same traveling direction, the property is violated. One counterexample that could be returned by a standard model checker is that the lift is held by some user for a long time so that other lifts pass by the floor in the same direction first. Designing a multi-lift system which always satisfies the property is extremely challenging. One way is to re-assign all external requests every time a lift travels to a different floor. Due to high complexity, many existing lift systems do not support re-assigning requests. The question is then: *what is the probability of violating the property, with typical randomized arrival of user requests from different floors or from the button panels inside the lifts*? If the probability is sufficiently low, then the design may be considered as acceptable. Further, can we prove that choosing the 'nearest' lift is actually better than assigning an external request to a random lift?

The above example illustrates two challenges for applying model checking in real-life systems. Firstly, an expressive modeling language supporting features like real-time, hierarchy, concurrency, data structures as well as probability, may be required to model complex systems. Secondly, the models should be efficiently model checkable for widely used properties, such as reachability checking and Linear Temporal Logic(LTL) checking. One line of work on modeling complicated systems is based on integrated formal specification languages [10, 23]. These proposals suffer from one limitation, i.e., there are few supporting tools for system simulation or verification. Existing model checkers are limited because they do not support one or many of the required system features. For instance, SPIN [17] supports complex data operations and concurrency, but not real-time or probability. UPPAAL [7] supports real-time, concurrency and recently data operations as well as probability (in the extension named UPPAAL-PRO), but lacks support for hierarchical control flow and is limited to maximal probabilistic reachability checking. PRISM [15] is popular in verifying systems having concurrency, probability and the combination of real-time and probability in its latest version [19]. However, it does not support hierarchical systems, but rather networks of flat finite state systems. In addition, most of the tools support only simple data operations, which could be insufficient in modeling systems which have complicated structures and complex data operations, such as the multi-lift system.

Contribution Compared to our previous work [28, 27], the contributions of this work are threefold. First, we develop an expressive modeling language called PRTS, combining language features from [28, 27]. PRTS is a combination of data structures, hierarchy, real-time, probability, concurrency, etc, and it is carefully designed in order to be expressive and also model checkable for different properties. Second, a fully automated method is used to generate abstractions from PRTS models. We show that the infinite states caused by real-time transitions could be reduced to finitely zones, which are then subject to probabilistic model checking. The abstraction technique proposed in [27] is extended to PRTS and shown to be probability preserving. Third, we implement a dedicated model checker as a part of the PAT model checker [26], which supports editing, simulating and verifying PRTS models. The tool has been applied to the multi-lift system and benchmark systems.

Organization The paper is structured as follows. Section 2 recalls background. Section 3 introduces the proposed modeling language PRTS. Section 4 defines its operational semantics. Section 5 describes zone-based abstraction technique, which leads to the model checking approach in Section 6. The evaluation is reported in Section 7. Section 8 surveys related work. Section 9 concludes the paper and discusses future work.

2 Basic Concepts

In this section, we recall some basic concepts and definitions of model checking techniques [5] that will be used throughout the rest of the paper. When modeling probabilistic systems (particularly, discrete-time stochastic control processes), MDP is one of the most widely used models. An MDP is a directed graph whose transitions are labeled with events or probabilities. The following notations are used to denote different transition labels. $\mathbb{R}_+$ denotes the set of non-negative real numbers; $\epsilon \in \mathbb{R}_+$ denotes the event of idling for exactly ϵ time units; τ denotes an unobservable event; Act denotes the set of observable events such that $\tau \notin Act$; Act_τ denotes $Act \cup \{\tau\}$. Given a countable set of states S, a distribution is a function $\mu : S \to [0, 1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. μ is a *trivial* distribution or is trivial if and only if there exists a state $s \in S$ such that $\mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S. Formally,

Definition 1. *An MDP is a tuple* $\mathcal{D} = (S, init, Act, Pr)$ *where* S *is a set of states;* $init \in S$ *is the initial state;* $Pr : S \times (Act_\tau \cup \mathbb{R}_+) \times Distr(S)$ *is a transition relation.*

An MDP $\mathcal{D}$ is *finite* if and only if S and $Distr(S)$ are finite. For simplicity, a transition is written as: $s \xrightarrow{x} \mu$ such that $s \in S$; $x \in Act_\tau \cup \mathbb{R}_+$ and $\mu \in Distr(S)$. If μ is trivial, i.e., $\mu(s') = 1$, then we write $s \xrightarrow{x} s'$. There are three kinds of transitions. A time-transition is labeled with a real-valued constant $\epsilon \in \mathbb{R}_+$. An observable transition is labeled with an event in Act. An un-observable transition is labeled with τ. Throughout the paper, *MDPs are assumed to be deadlock-free following the standard practice*. A deadlocking MDP can be made deadlock-free by adding self loops labeled with τ and probability 1 to the deadlocking states, without affecting the result of probabilistic verification.

A state of $\mathcal{D}$ may have multiple outgoing distributions, possibly associated with different events. A scheduler is a function deciding which event and distribution to choose. A Markov Chain [5] can be defined given an MDP $\mathcal{D}$ and a scheduler δ, which is denoted as $\mathcal{D}^\delta$. A Markov Chain is an MDP where only one event and distribution is available at every state. Intuitively speaking, given a state s, firstly an enabled event and a distribution are selected by the scheduler, and then one of the successor states is reached according to the probability distribution. A rooted run of $\mathcal{D}^\delta$ is an alternating sequence of states and events $\pi = \langle s_0, x_0, s_1, x_1, \cdots \rangle$ such that $s_0 = init$. The sequence $\langle x_0, x_1, \cdots \rangle$, denoted as $trace(\pi)$, is a trace of $\mathcal{D}^\delta$. Let $runs(\mathcal{D}^\delta)$ denote the set of rooted runs of $\mathcal{D}$. Let $traces(\mathcal{D}^\delta)$ denote the set of traces of $\mathcal{D}^\delta$. Given $\mathcal{D}^\delta$ and $s_i \in D$, let μ_i be the (only) distribution at s_i. The probability of exhibiting π in $\mathcal{D}^\delta$, denoted as $\mathcal{P}_{\mathcal{D}^\delta}(\pi)$, is $\mu_0(s_1) * \mu_1(s_2) * \cdots$.

It is often useful to find out the probability of $\mathcal{D}$ satisfying a property ϕ. Note that with different schedulers, the result may be different. For instance, if ϕ is reachability of a state s, then s may be reached by different scheduling with different probability. The measurement of interest is thus the maximum and minimum probability of satisfying ϕ. The maximum probability is defined as follows.

$$\mathcal{P}_{\mathcal{D}}^{max}(\phi) = \sup_\delta \mathcal{P}_{\mathcal{D}}(\{\pi \in runs(\mathcal{D}^\delta) \mid \pi \text{ satisfies } \phi\})$$

Note that the supremum ranges over all, potentially infinitely many, schedulers. Intuitively, it is the maximum of probability of satisfying ϕ with any scheduler. The minimum is defined as: $\mathcal{P}_{\mathcal{D}}^{min}(\phi) = \inf_\delta \mathcal{P}_{\mathcal{D}}(\{\pi \in runs(\mathcal{D}^\delta) \mid \pi \text{ satisfies } \phi\})$ which yields the best lower bound that can be guaranteed for the probability of satisfying ϕ. For different classes of properties, there are different methods to calculate the maximum and minimum probability, e.g., reachability by solving a linear program or graph-based iterative methods; LTL checking by identifying end components and then calculating reachability probability [5].

3 Syntax of PRTS

The choice of modeling language is an important factor in the success of the entire system analysis or development. The language should cover several facets of the requirements and the model should reflect exactly (up to abstraction of irrelevant details) a system. In this work, we draw upon existing approaches [16, 21, 2, 27] and create the single notation PRTS. In the following, we briefly introduce the syntax of a core subset of PRTS. Interested readers can refer to PAT user manual for a complete list of constructs and detailed explanation.

A PRTS model (hereafter model) is a 3-tuple (Var, σ_i, P) where Var is a finite set of finite-domain global variables; σ_i is the initial valuation of Var and P is a process which captures the control logic of the system. A process is defined in form of $Proc(\overline{para}) = PExpr$ where $Proc$ is a process name; $\overline{para}$ is a vector of parameters and $PExpr$ is a process expression. A rich set of process constructs are defined to capture different features of various systems, as shown in the following.

$$\begin{aligned}P = {} & Stop \mid Skip \mid e \rightarrow P \mid P \,\Box\, Q \mid P \sqcap Q \mid P;\ Q \mid P \parallel Q \mid P \,|||\, Q \\ & \mid\ P \setminus \{X\} \mid if\ b\ then\ P\ else\ Q \mid a\{program\} \rightarrow P \mid Wait[d] \\ & \mid\ P\ timeout[d]\ Q \mid P\ interrupt[d]\ Q \mid P\ deadline[d] \mid P\ within[d] \\ & \mid\ pcase\{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\} \mid ref(Q)\end{aligned}$$

Hierarchical control flow A number of the constructs are adapted from the classic CSP [16] to support modeling of hierarchical systems. Process $Stop$ and $Skip$ are process primitives, which denote inaction and termination respectively. Process $e \rightarrow P$ engages in an abstract event e first and then behaves as process P. Event e may serve as a multi-party synchronization barrier if combined with parallel composition $\parallel$. A variety of choices are supported, e.g., $P \Box Q$ for unconditional choice; and $if\ b\ then\ P\ else\ Q$ for conditional branching in which b is a boolean expression composed by process parameters and variables in Var. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization. If P and Q only communicate through variables [5], then it is written as $P \mid\mid\mid Q$. Process $P \setminus \{X\}$ hides occurrence of any event in $\{X\}$. Recursion is supported by referencing a process name with concrete parameters. The semantics of the constructs is defined in [29].

Data structures and operations Different from CSP, a PRTS model is equipped with a set of variables Var. Variables can be of simple types like Boolean or integer or arrays of simple types. In order to support arbitrary complex data structures and operations, user-defined data types are allowed. A user-defined data type must be defined externally (e.g., as a C# library), and imported in a model. The detailed explanation of the interface methods and examples of creating/using C# library can be found in PAT user manual. Note that in order to guarantee that model checking is terminating, each data object must have only finitely many different values and *all data operations must be terminating*, both of which are users' responsibility. Furthermore, users are recommended to apply standard programming techniques like using assertions to ensure correctness of the data operations. Data operations are invoked through process expression $a\{program\} \rightarrow P$, which generates an event a and atomically executes program $program$ at the same time, and then behaves as P. In other words, $program$ is a *transaction*. Variable updates are allowed in $program$. In order to prevent data race, event a with an attached program will not to be synchronized by multiple processes.

Real-time A number of timed process constructs are supported in PRTS to cover common timed behavioral patterns. Process $Wait[d]$ idles for exactly d time units, where d an integer constant. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since the process is *activated*). Otherwise, Q takes over control after exactly d time units. Process $P\ interrupt[d]\ Q$ behaves exactly as P (which may engage in multiple observable events) until d time units elapse, and then Q takes over control. PRTS extends Timed CSP [25] with additional timed process constructs. Process $P\ deadline[d]$ constrains P to terminate before d time units. Process $P\ within[d]$ requires that P must perform an observable event within d time units. Constant d associated with the timed process constructs are referred as the *parameter of the timed process construct*. Note that real-time systems modeled in PRTS can be fully hierarchical, whereas Timed Automata based languages (e.g., the one supported by Uppaal) often have the form of a network of flat Timed Automata.

Probability In order to randomized behaviors (i.e., unreliable environment or cognitive aspects of user behaviors), probabilistic choices are introduced as follows.

$$pcase\ \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\}$$

where pr_i is a positive integer constant to express the probability weight. Intuitively, it means that with $\frac{pr_i}{pr_0+pr_1+\cdots+pr_k}$ probability, the system behaves as P_i. Note that the

[5] Or synchronous/asynchronous channels. The details are skipped for simplicity.

```
1. #define NoOfFloors 2;
2. #define NoOfLifts 2;
3. #import "PAT.Lib.Lift";
4. var<LiftControl> ctrl = new LiftControl(NoOfFloors,NoOfLifts);
5. Users() = pcase {
6.           1 : extreq.0.1{ctrl.Assign_External_Up_Request(0)} -> Skip
7.           1 : intreq.0.0.1{ctrl.Add_Internal_Request(0,0)} -> Skip
8.           1 : intreq.1.0.1{ctrl.Add_Internal_Request(1,0)} -> Skip
9.           1 : extreq.1.0{ctrl.Assign_External_Down_Request(1)} -> Skip
10.          1 : intreq.0.1.1{ctrl.Add_Internal_Request(0,1)} -> Skip
11.          1 : intreq.1.1.1{ctrl.Add_Internal_Request(1,1)} -> Skip
12.      } within[1]; Users();
13. Lift(i, level, direction) = ...;
14. System = (||| x:{0..NoOfLifts-1} @ Lift(x, 0, 1)) ||| Users();
```

Fig. 1. A lift system model

```
public void Assign_External_Up_Request(int level) {
    ...
    int minimumDistance = int.MaxValue;
    int chosenLift = -1;
    for (int i = 0; i < LiftStatus.Length; i++) {
        int distance;
        if (LiftStatus[i] >= 0) {
            if (LiftStatus[i] <= level) {
                distance = level - LiftStatus[i];
            } else {
                distance = NoOfFloors - LiftStatus[i] + NoOfFloors - level;
            }
        } else {
            distance = LiftStatus[i] * -1 + level;
        }
        if (distance < minimumDistance) {
            chosenLift = i;
            minimumDistance = distance;
        }
    }
    ExternalRequestsUp[level] = chosenLift;
}
```

Fig. 2. A data operation example

sum of all the probabilities in one *pcase* is guaranteed to be 1. Process P_i can be any process and thus PRTS supports fully hierarchical probabilistic systems.

Example 2. We use the lift system example to illustrate modeling with PRTS. The model (in ASCII format as supported in PAT) is shown in Figure 1. Line 1 and 2 define two constants which denote the number of floors and lifts respectively. Line 3 imports a C# library, which defines a data type *LiftControl* encapsulating all data components and operations of the lift system. Note that it is a design decision whether to maintain the data externally in the C# library or in the model itself. A *LiftControl* object contains multiple data structures, e.g., an integer array for user requests from external button panels, a two dimensional array for requests for internal button panels, etc. Interested readers can refer to PAT (version 3.0 or later, open with PAT's C# editor and compiler) for its details. The *LiftControl* class also defines multiple data operations. For instance, one of them is shown in Figure 2 which assigns an external request for traveling upwards to a lift. The idea is to assign a request to a lift which can reach the requesting floor by traveling the minimum number of floors (without changing direction except at the top or bottom floor). Note that $level$ denotes the requesting floor and $LiftStatus$ is an array maintaining status of the lifts, i.e., $ListStatus[i] = -2$ means that i-th lift is at level 2 traveling downwards. In Figure 1, line 4 of the lift model creates a $LiftControl$ object named $ctrl$. Line 5 to 12 defines a process $Users()$, which models behavior of the users. In this (overly) simplified model, user requests are assumed to arrive periodically with uniform probabilistic distribution[6]. There are 6 different requests with 2 floors and 2 lifts (two of which are external requests). Each is given $\frac{1}{6}$ probability, as modeled using $pcase$ at line 5-12 in Figure 1. For instance, event $extreq.0.1$ models an external request at 0-floor for traveling upwards. The event is associated with a program which invokes the method for assigning requests to lifts through object $ctrl$. Note that user behaviors are subject to real-time constraint, i.e., a request is requested within 1 second, modeled using $within[1]$. At line 13, process $Lift$ which is composed of sub-processes models an individual lift. We skip its details for the sake of space. At the top level, the system is the interleaving of users and lifts at line 14.

4 Operational Semantics

The semantics of a PRTS model is an MDP, due to its mixture of nondeterminism and probabilistic choices. In order to define the operational semantics, we define the notion of a configuration to capture the global system state during the execution, referred as *concrete configurations*. This terminology distinguishes the notion from the abstract configurations which will be introduced in Section 5.

Definition 2. *A concrete system configuration is a tuple $c = (\sigma, P)$ where σ is a variable valuation and P is a process.*

Given a model, the probabilistic transition relation of its MDP semantics can be defined by associating a set of firing rules with every process construct, which are also known as *concrete firing rules*. In the following, the rules for process $Wait[d]$, $P\ timeout[d]\ Q$ and $pcase$ are exemplified in Figure 3. The rest are similarly defined (available in [29]). The top two rules capture behaviors of process $Wait[d]$. The first rule states that through a time-transition, a process may idle for any amount of time as long as it is less than or equal to d time units. Note that no variable update is not possible in time-transitions. The second rule states that the process terminates immediately after d becomes 0. The next four rules capture semantics of process $P\ timeout[d]\ Q$. If an observable event e can be performed by P, then $P\ timeout[d]\ Q$ becomes P' (the first rule). That is, once an observable event is engaged before d time units, time-out never occurs. If d is 0, Q may take over control and the whole process becomes Q via a τ-transition (the second rule). Note that it is possible that an observable event occurs when d is 0. Only that when d is 0, time-transition is not allowed before the τ-transition. If an unobservable transition is generated by P, the $timeout$ operator remains (the third rule). If P may idle for less than or equal to d time units, so is $P\ timeout[d]\ Q$. All above

[6] A realistic user model can be obtained by mining data of actual lift systems.

$$\frac{\epsilon \leq d}{(\sigma,\ \mathit{Wait}[d]) \xrightarrow{\epsilon} (\sigma,\ \mathit{Wait}[d-\epsilon])} \qquad \frac{}{(\sigma,\ \mathit{Wait}[0]) \xrightarrow{\tau} (\sigma, \mathit{Skip})}$$

$$\frac{(\sigma, P) \xrightarrow{e} (\sigma', P'),\ e \in \mathit{Act}}{(\sigma, P\ \mathit{timeout}[d]\ Q) \xrightarrow{e} (\sigma', P')} \qquad \frac{}{(\sigma, P\ \mathit{timeout}[0]\ Q) \xrightarrow{\tau} (\sigma, Q)}$$

$$\frac{(\sigma, P) \xrightarrow{\tau} (\sigma', P')}{(\sigma, P\ \mathit{timeout}[d]\ Q) \xrightarrow{\tau} (\sigma', P'\ \mathit{timeout}[d]\ Q)}$$

$$\frac{(\sigma, P) \xrightarrow{\epsilon} (\sigma, P'), \epsilon \leq d}{(\sigma, P\ \mathit{timeout}[d]\ Q) \xrightarrow{\epsilon} (\sigma, P'\ \mathit{timeout}[d-\epsilon]\ Q)}$$

$$\frac{}{\begin{array}{l}(\sigma, \mathit{pcase}\ \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\}) \xrightarrow{\tau} \mu \\ \quad \text{s.t. } \mu((\sigma, P_i)) = \frac{pr_i}{pr_0 + pr_1 + \cdots + pr_k} \text{ for all } i \in [0,\ k]\end{array}}\ [\ \mathit{pcase}\ ]$$

Fig. 3. Concrete firing rules

transitions result in trivial distributions. The resultant distribution of the $pcase$ process is defined such that the probability of becoming P_i is pr_i. Note that neither variable valuation nor time change. Rule $pcase$ is the only rule which produces a nontrivial distribution. *We remark that different from Probabilistic Timed Automata(PTA) [14, 21], probability and time are separated in PRTS, i.e., a transition can be either time-consuming or has trivial probability but never both.*

Definition 3. *Let $M = (Var, \sigma_{init}, P)$ be a model. $\mathcal{D}_M$ is an MDP $(S, init, Act, Pr)$ such that S is a set of concrete system configurations; $init = (\sigma_{init}, P)$; and $Pr : S \times (Act_\tau \cup \mathbb{R}_+) \times Distr(S)$ is defined by the firing rules.*

$\mathcal{D}_M$ is referred to as the *concrete semantics of M*. Because PRTS has a dense-time semantics, $\mathcal{D}_M$ has infinitely many states. In order to apply model checking techniques, a finite-state abstract MDP is required.

5 Abstraction

In this section, we present a fully automated approach to generate a finite-state abstract MDP from a model. Without loss of generality, *we assume that every process reachable from the initial configuration is finite-state (as defined in [24])*. As a result, in a process which has finitely many process constructs, the only source of infinity is timing, or equivalently, the infinitely many possible values for parameters of timed process constructs. For instance, given process $Wait[1]$, there are infinitely many processes that can be reached by a time-transition, e.g., $Wait[0.9]$, $Wait[0.99]$, $Wait[0.999]$, etc. One observation is that for certain properties, the exact value of the parameters is not important, i.e., they can be grouped into equivalent classes. This leads to the idea of using a constraint to capture the value of the parameters. In the following, we summarize *dynamic zone abstraction* [27] and prove that it can be applied to PRTS models without changing the results of probabilistic properties.

In order to distinguish parameters associated with different process constructs, the first step of the abstraction is to associate timed process constructs with clocks. Constraints on the clocks are then used to capture values of the respective parameters. For simplicity of presentation, we assume that each process construct is associated with a unique clock[7]. For instance, let $P\ timeout[d]_c\ Q$ denote that process $P\ timeout[d]\ Q$ is associated with clock c. $P\ timeout[d]_c\ Q$ with a constraint $c \leq 5$ represents any process $P\ timeout[d']\ Q$ with $d' \leq 5$. This gives the notion of abstract system configurations, which compose the abstract MDP.

Definition 4. *Given a concrete system configuration (σ, P), the corresponding abstract system configuration is a triple (σ, P_T, D) such that P_T is a process obtained by associating P with a set of clocks; and D is a zone over the clocks.*

There are usually multiple timed process constructs in a process P. Nonetheless, at one moment not all of the timed constructs are activated, i.e., only some of them are ready to take over control and perform a transition. We write $cl(P)$ to denote the set of clocks activated in P and $X = 0$ where X is a set of clocks to denote the conjunction of $c = 0$ for all $c \in X$.

A zone D is the conjunction of multiple primitive constraints over a set of clocks. A primitive constraint is of the form $t \sim d$ or $t_i - t_j \sim d$ where t, t_i, t_j are clocks, d is a constant and $\sim$ is either, $\geq$, $=$ or $\leq$[8]. Intuitively, a zone is the maximal set of clock valuations satisfying the constraint. A zone is empty if and only if the constraint is unsatisfiable. An abstraction configuration (σ, P_T, D) is valid if and only if D is not empty. The following zone operations are relevant. Let D denote a zone. $D^\uparrow$ denotes the zone obtained by delaying arbitrary amount of time. Note that all clocks proceed at the same rate. For instance, let c be a clock, $(c \leq 5)^\uparrow$ is $c \leq \infty$. Given a set of clocks X, $D[X]$ denotes the set of valuations of clocks in X which satisfy D. Zones can be equivalently represented as Difference Bound Matrices(DBMs) and zone operations can be translated into DBMs manipulation [12, 8].

In order to define the abstract MDP, we define abstract firing rules. To distinguish from concrete transitions, an abstract transition is written in the form: $(\sigma, P_T, D) \overset{e}{\leadsto} (\sigma', P'_T, D')$. Figure 4 shows the abstract rules for process $Wait[d]$, $P\ timeout[d]\ Q$ and $pcase$ as examples. Given process P which is associated with clocks, $idle(P)$ is defined to be the maximum zone such that P can idle before performing an event-transition. For instance, $idle(P\ deadline[5]_c) = idle(P) \wedge c \leq 5$, i.e., $P\ deadline[5]_c$ can idle as long as P can idle and the reading of c is no bigger than 5. Refer to [29]

[7] For practice, clocks are renamed dynamically so that they are shared by processes which are activated at the same time. Refer to details in [27].

[8] In our setting, the clock constraints are always closed.

for the detailed definition of $idle(P)$ and the rest of the abstract firing rules. Rule ade in Figure 4 states that process $Wait[d]$ idles for exactly d time units and then engages in event τ and the process transforms to $Skip$. Note that the zone of the target configuration is $D^{\uparrow} \wedge c = d$. Intuitively, it means that the transition occurs sometime in the future (captured by $D^{\uparrow}$) when c reads d (captured by $c = d$). It should be clear that this is 'equivalent' to the concrete firing rules. Rule $ato1$, $ato2$ and $ato3$ capture the abstract semantics of $P\ timeout[d]\ Q$. Depending on when the first event of P takes place and whether it is observable, process $P\ timeout[d]\ Q$ behaves differently in three ways. Rule $ato1$ states that if P generates a τ-transition, the $timeout$ construct remains. Furthermore, the target zone $D' \wedge c \leq d$ constrains that the transition must take place no later than d time units. In contrast, rule $ato2$ states that if P generates an observable transition, then the $timeout$ construct is removed. Similarly, it is constrained that the transition must occur no later than d time units. Rule $ato3$ captures the case when timeout occurs. Namely, timeout occurs if and only if the reading of c is exactly d and, further, P must be able to idle until c reads d. Rule $apcase$ captures the abstract semantics of $pcase$. *Note that this τ-transition is instantaneous.*

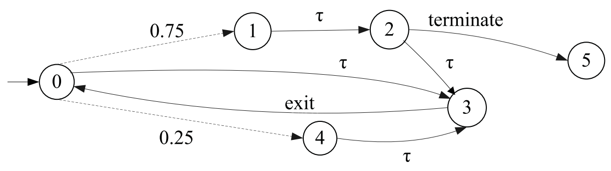
Definition 5. *Let $M = (Var, \sigma_{init}, Proc)$ be a model. $\mathcal{D}_M^a = (S_a, init_a, Act, Pr_a)$ is the abstract MDP such that S_a is a set of valid abstract system configurations; $init_a = (\sigma_{init}, Proc, D_{init})$ is the initial abstract configuration where D_{init} is $cl(Proc) = 0$; and Pr_a is the smallest transition relation such that: for all $s \in S_a$, if $s \overset{a}{\rightsquigarrow} \mu$, then $(s, a, \mu') \in Pr_a$ such that: if $\mu((\sigma, P, D)) > 0$, then $\mu'((\sigma, P, D')) = \mu((\sigma, P, D))$ where $D' = D[cl(Q)] \wedge cl(Q) - cl(P) = 0$.*

Informally, for any (σ, P_T, D) obtained by applying an abstract firing rule, D' is obtained by firstly pruning all clocks which are not in $cl(Q)$ and then setting clocks associated with newly activated processes (i.e., $cl(Q) - cl(P)$) to be 0. The construct of $\mathcal{D}_M^a$ is illustrated in the following example.

Example 3. Assume a model $M = (\varnothing, \varnothing, P)$ such that process P is defined as follows.

$$P = (pcase\ \{1 : Wait[2]_{c_0};\ 3 : Wait[5]_{c_1}\})\ timeout[3]_{c_2}\ exit \rightarrow P$$

The abstract MDP is shown as follows.



A transition is labeled with an event (with a skipped probability 1) or a probability less than 1 (with a skipped event τ). Note that all transitions with resulting in a non-trivial distribution is labeled with τ, whereas all transitions labeled with an event other than τ has probability 1. The initial configuration state 0 is $(\varnothing, P, c_2 = 0)$ where clock c_2 is associated with $timeout[3]$ in P. Applying rule $ato3$, we get the transition from state 0 to state 3. Note that clock c_2 is pruned after the transition because it is no longer associated with any process constructs. Applying rule $apcase$, we

$$\frac{}{(\sigma, \mathit{Wait}[d]_c, D) \overset{\tau}{\rightsquigarrow} (\sigma, \mathit{Skip}, D^{\uparrow} \wedge c = d)} \; [\, \mathit{ade} \,]$$

$$\frac{(\sigma, P, D) \overset{\tau}{\rightsquigarrow} (\sigma', P', D')}{(\sigma, P \; \mathit{timeout}[d]_c \; Q, D) \overset{\tau}{\rightsquigarrow} (\sigma', P' \; \mathit{timeout}[d]_c \; Q, D' \wedge c \leq d)} \; [\, \mathit{ato}1 \,]$$

$$\frac{(\sigma, P, D) \overset{e}{\rightsquigarrow} (\sigma', P', D'), e \neq \tau}{(\sigma, P \; \mathit{timeout}[d]_c \; Q, D) \overset{e}{\rightsquigarrow} (\sigma', P', D' \wedge c \leq d)} \; [\, \mathit{ato}2 \,]$$

$$\frac{}{(\sigma, P \; \mathit{timeout}[d]_c \; Q, D) \overset{\tau}{\rightsquigarrow} (\sigma, Q, c = d \wedge \mathit{idle}(P))} \; [\, \mathit{ato}3 \,]$$

$$\frac{}{\begin{array}{l}(\sigma, \mathit{pcase}\; \{pr_0 : P_0;\; pr_1 : P_1;\; \cdots;\; pr_k : P_k\}, D) \overset{\tau}{\rightsquigarrow} \mu \\ \quad \text{s.t. } \mu((\sigma, P_i, D)) = \frac{pr_i}{pr_0 + pr_1 + \cdots + pr_k} \text{ for } i \in [0, k]\end{array}} \; [\, \mathit{apcase} \,]$$

Fig. 4. Abstract firing rules

get the transitions from state 0 to state 1 and 4, which belong to the same distribution. Note that clock c_2 is not pruned during both transitions. State 4 is as follows: $(\varnothing, \mathit{Wait}[5]_{c_1} \; \mathit{timeout}[3]_{c_2} \; \mathit{exit} \rightarrow P, c_2 = 0 \wedge c_1 = 0)$. By rule apcase, the τ-transition is instantaneous and thus $c_2 = 0$. Note that $c_2 = c_1$ since c_1 starts when $c_2 = 0$. Two rules can be applied to state 4, i.e., $\mathit{ato}3$ so that timeout occurs or otherwise $\mathit{ato}2$. Applying rule $\mathit{ato}3$, we obtain the transition from state 4 to state 3. Applying rule $\mathit{ato}2$, we obtain the following zone: $c_1 = 5 \wedge c_2 \leq 3 \wedge c_1 = c_2$. It can be shown that the zone is empty and hence the transition is infeasible. Other transitions are similarly obtained. Note that the event $\mathit{terminate}$ is generated by the process Skip which is in term generated from $\mathit{Wait}[2]$ (rule ade). □

6 Verification

We show that the abstraction model can be verified with standard probabilistic model checking techniques. Given a model M, $\mathcal{D}^a_M$ must be finite so as to be model checkable.

Theorem 1. *$\mathcal{D}^a_M$ is finite for any model M.* □

A proof sketch is as follows. The number of states in $\mathcal{D}^a_M$ is bounded by the number of 1) variable valuations, 2) process expressions and 3) zones. By assumption, 1) is finite. Because clocks are associated with timed process constructs and we assume that every reachable process P is finite, $cl(P)$ is finite. It can be shown that by reusing clocks, finitely many clocks are sufficient. Combined with our assumption, 2) is finite. By previous work on zone abstraction [9], 3) is finite[9].

[9] Zone normalization is not necessary as all clocks are bounded from above. Refer to [29].

Furthermore, the abstract semantics $\mathcal{D}_M^a$ must be 'sufficiently' equivalent to the concrete semantics $\mathcal{D}_M$ so that verification results based on $\mathcal{D}_M^a$ apply to $\mathcal{D}_M$. In the following, we show that our abstraction is probability preserving with respect to one popular class of properties: (untimed) LTL-X (i.e., LTL without the 'next' operator)[10]. Assume that ϕ is an LTL-X formula, constituted by temporal operators, logic operators and atomic propositions on variables. Given a run π of an MDP and ϕ, satisfaction of ϕ by π is defined in the standard way. Let $\mathcal{P}_{\mathcal{D}}^{max}(\phi)$ be the maximum probability of MDP $\mathcal{D}$ satisfying ϕ; $\mathcal{P}_{\mathcal{D}}^{min}(\phi)$ be the minimum probability satisfying ϕ. The following establishes that it is sound and complete to model-check LTL-X against $\mathcal{D}_M^a$.

Theorem 2. *Let M be a model. $\mathcal{P}_{\mathcal{D}_M^a}^{max}(\phi) = \mathcal{P}_{\mathcal{D}_M}^{max}(\phi)$ and $\mathcal{P}_{\mathcal{D}_M^a}^{min}(\phi) = \mathcal{P}_{\mathcal{D}_M}^{min}(\phi)$.* $\square$

A proof sketch is as follows. Refer to [29] for a complete proof. This theorem is proved by showing that: for every run ex of $\mathcal{D}_M$, there is a run ex' of $\mathcal{D}_M^a$(and vice versa) such that (1) ex and ex' are stutter equivalent in terms of variable valuations; (2) ex and ex' have the same probability. Intuitively, (1) is true because variable valuations do not change through time transitions and all that our abstract does is to encapsulate time transitions (while preserving event transitions). (2) is true because time-transitions always have probability 1. We remark that it is known that forward analysis of PTA [21] is not accurate (e.g., the maximum probability returned is an over-approximation) because zone graphes generated from Timed Automata do not satisfy (pre)-stability. We show in [29] that dynamic zone abstraction (for Stateful Timed CSP [27] and ergo PRTS) generates zone graphes which are time-abstract bi-similar to the concrete transition systems and satisfy (pre-)stability. As a result, forward analysis of $\mathcal{D}_M^a$ is accurate.

We adopt the automata-based approach [5] to check LTL-X properties. Firstly, a deterministic Rabin automaton equivalent to a given LTL-X formula is built. The product of the automaton and the abstract MDP is then computed. Thirdly, *end components* in the product which satisfy the Rabin acceptance condition are identified. Lastly, the probability of reaching any state of the end components is calculated, which equals the probability of satisfying the property.

7 Implementation and Evaluation

System modeling, simulation and verification in PRTS have been supported (as a module) in PAT[11]. PAT has user-friendly editor, simulator and verifier and works under different operating systems. After inputting PRTS model in the editor, users could simulate the system behaviors step by step or generate the whole state space if the number of states is under some constraint. For verification, besides the LTL-X checking, PRTS also supports reachability checking, and refinement checking (i.e., calculating the probability of a probabilistic system exhibiting any behaviors of a non-probabilistic specification).

To answer the question on our motivating example, we verify the lift model and compare two ways of assigning external requests. One is to assign the request to a *random* lift. The other is that an external request is always assigned to the 'nearest' lift.

[10] The next operator is omitted because its semantics for real-times systems can be confusing.

[11] http://www.patroot.com

System	Random		Nearest	
	Result(pmax)	Time(s)	Result(pmax)	Time(s)
lift=2; floor=2; user=2	0.21875	3.262	0.13889	2.385
lift=2; floor=2; user=3	0.47656	38.785	0.34722	18.061
lift=2; floor=2; user=4	0.6792	224.708	0.53781	78.484
lift=2; floor=2; user=5	0.81372	945.853	0.68403	223.036
lift=2; floor=3; user=2	0.2551	12.172	0.18	6.757
lift=2; floor=3; user=3	0.54009	364.588	0.427	119.810
lift=2; floor=3; user=4	0.74396	11479.966	0.6335	1956.041
lift=2; floor=4; user=2	0.27	27.888	0.19898	13.693
lift=3; floor=2; user=2	0.22917	208.481	0.10938	88.549
lift=3; floor=2; user=3	OOM	OOM	0.27344	3093.969

Table 1. Experiments: Lift System

For simplicity, we assume external requests are never re-assigned. A lift works as follows. It firstly checks whether it should serve the current floor. If positive, it opens its door and then repeats from the beginning later. If negative, it checks whether it should continue traveling in the same direction (if there are internal requests or assigned external requests on the way) or change direction (if there are internal or assigned external requests on the other direction) or simply idle (otherwise). Note that it is constrained (using $within$) to react regularly. The property that a lift should not pass by without serving a user's external request is verified through probabilistic reachability analysis, i.e., what is the maximal probability of reaching a state such that a lift is passing by a requested floor in the requested direction. Table 1 summarizes the experiment results, where OOM means out of memory. The experiment testbed is a PC running Windows Server 2008 64 Bit with Intel Xeon 4-Core CPU$\times$2 and 32 GB memory. Details about our experiments are available at $http://www.comp.nus.edu.sg/\tilde{}pat/icfem/prts$.

The parameters of the model denote the number of *lifts*, the number of *floors* and number of *user requests* respectively. We limit the number of user requests so as to check how the probability varies as well as to avoid state space explosion. It is inf when there is no limit. Column $Random$ and $Nearest$ shows the *maximum* probability of violating the property with $random$ assignment and '$nearest$' assignment respectively. Note that it can be shown that the minimum probability is always 0 (i.e., there exists a scheduler which guarantees satisfaction of the property). The following conclusion can be made. Firstly, it takes at least two external requests, two lifts and two floors to constitute a bad behavior, e.g., one lift is at top floor (and later going down to serve a request), while a request for going down at the top floor is assigned to the other lift. Secondly, the more user requests, the higher the probability is. Intuitively, this means that with more requests, it is more likely that the bad behavior occurs. Similarly, the probability is higher with more floors. Lastly, 'nearest' assignment performs better than random assignment as expected, i.e., the maximum probability of exhibiting a bad behavior with the former is always lower than with the latter in all cases.

The statistics on memory consumption is skipped as PAT only generates an estimated memory usage for each verification run because memory usage is managed by

System	Property	Result	PAT(s)	PRISM(s)
ME (N=5)	LTL	1	9.031	7.413
ME (N=8)	LTL	1	185.051	149.448
RC (N=4,K=4)	LTL	0.99935	4.287	33.091
RC (N=6,K=6)	LTL	1	146.089	2311.388
CS (N=2, K=4)	LTL	0.99902	9.362	1.014
CS (N=3, K=2)	LTL	0.85962	212.712	7.628

Table 2. Experiments: PAT vs PRISM

.NET framework. In average, our current implementation processes 11K states per second (or millions in one hour) in these experiments, which is less than other explicit-state model checkers like SPIN. This is expected given the complexity in handling PRTS. State space explosion occurs when there are more than 3 lifts and more than 4 floors. This, however, should not be taken as the limit of PAT, as many optimization techniques are yet to incorporated.

Next, the PRTS checker is compared with state-of-the-art probabilistic model checker PRISM on verifying benchmark systems based on MDP. The results are summarized in Table 2. We use existing PRISM models; re-model them using PRTS and verify them. The models are a mutual exclusion protocol (ME), a randomized consensus algorithm(RC), and the CSMA/CD protocol (CS). We use the iterative method in calculating the probability and set termination threshold as relative difference 1.0E-6 (same as PRISM). Our implementation is better for CS, slightly slower than ME and significantly slower for RC. The main reason that PAT could outperform PRISM in some cases is that models in the PRTS have much fewer states than their respective in PRISM - due to difference in modeling language design. In general, PRISM handles more states per time unit than PAT. The main reason is the complexity in handling hierarchical models. Note that though these models have simple structures, there is overhead for maintaining underlying data structures designed for hierarchical systems. PRISM is based on MTBDD or sparse matrix or a hybrid approach, whereas PAT is based on explicit state representation currently. Symbolic methods like BDD are known to handle more states. Applying BDD techniques to hierarchical complex languages like PRTS is highly non-trivial. It remains as one of our ongoing work.

8 Related Work

There are several modeling methods and model checking algorithms for real-time probabilistic systems. Alur, Courcoubetis and Dill presented a model-checking algorithm for probabilistic real-time systems to verify TCTL formulae of probabilistic real-time systems [1]. Their specification is limited to deterministic Timed Automata, and its use of continuous probability distributions (a highly expressive modeling mechanism) does not permit the model to be automatically verified against logics which include bounds on probability. Remotely related is the line of work on Continuous-Time Markov Chains (CTMC) [4]. Different from CTMC, our work is based on discrete probability distributions. A method based on MTBDD for analyzing the stochastic and timing properties of systems was proposed in [3]. Properties are expressed in a subset of PCTL. The method was not based on real-time but in the realm of discrete time. Similar work using discrete time includes [13, 20].

Research on combining quantitative timing and probability has been mostly based on Probabilistic Timed Automata (PTA) [14, 21]. PTA extends Timed Automata [2] with nondeterministic choices, and discrete probability distributions which are defined over a finite set of edges. It is a modeling formalism for describing formally both nondeterministic and probabilistic aspects of real-time systems. Based on PTA, symbolic verification techniques [22] are developed using MTBDDs. In [6], Beauquier proposed another model of probabilistic Timed Automata. The model in [6] differs from PTA in that it allows different enabling conditions for edges related to a certain action and it uses Büchi conditions as accepting conditions. In [18], probabilistic timed program (PTP) is proposed to model real-time probabilistic software (e.g., SystemC). PTP is an extension of PTA with discrete variables. PTA and PTP are closely related to PRTS with some noticeable differences. Firstly, time transitions and probabilistic transitions are separated in PRTS. Secondly, (Stateful) Timed CSP is equivalent to closed Timed Automata (with τ-transitions) [24] and therefore strictly less expressive, which implies that PRTS is less expressive than PTA. Lastly, different from PRTS, models based on PTA or PTP often have a simple structure, e.g., a network of automata with no hierarchy.

Verification of real-time probabilistic systems often uses a combined approach, i.e., combination of real-time verifiers with probabilistic verifiers [11]. Our approach is a combination of real-time zone abstraction with MDP, which has no extra cost of linking different model checkers. This work is related to our previous works [27, 28] with the following new contribution: the two languages proposed in [27, 28] are combined to form PRTS and dynamic zone abstraction is seamlessly combined with probabilistic model checking to verify PRTS models.

9 Conclusion

We proposed a modeling language PRTS which is capable of specifying hierarchical complex systems with quantitative real-time features as well as probabilistic components. We show that dynamic zone abstraction results in probabilistic preserving finite-state abstractions, which are then subject to probabilistic model checking. In addition, we have extended our PAT model checker to support this kind of systems so that the techniques are easily accessible. As for future work, we are investigating state space reduction techniques such as symmetry reduction, bi-simulation reduction in the setting of PRTS. We are also exploring other classes of properties such as timed property.

References

1. R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for Probabilistic Real-time Systems. In *ICALP*, pages 115–126, 1991.
2. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

3. C. Baier, E. M. Clarke, V. H. Garmhausen, M. Z. Kwiatkowska, and M. Ryan. Symbolic Model Checking for Probabilistic Processes. In *ICALP*, pages 430–440, 1997.
4. C. Baier, B. R. Haverkort, H. Hermanns, and J. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.
5. C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
6. D. Beauquier. On Probabilistic Timed Automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003.
7. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, Wang Yi, and M. Hendriks. UPPAAL 4.0. In *QEST*, pages 125–126. IEEE, 2006.
8. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *CAV*, pages 341–353, 1999.
9. J. Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.
10. A. Butterfield, A. Sherif, and J. Woodcock. Slotted-Circus. In *IFM*, pages 75–97, 2007.
11. C. Daws, M. Kwiatkowska, and G. Norman. Automatic Verification of the IEEE 1394 Root Contention Protocol with KRONOS and PRISM. *International Journal on Software Tools for Technology Transfer*, 5(2-3):221–236, 2004.
12. D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.
13. V. H. Garmhausen, S. V. Aguiar Campos, and E. M. Clarke. ProbVerus: Probabilistic Symbolic Model Checking. In *ARTS*, pages 96–110, 1999.
14. H. Gregersen and H. E. Jensen. *Formal Design of Reliable Real Time Systems*. PhD thesis, 1995.
15. A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444, 2006.
16. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
17. G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
18. M. Kwiatkowska, G. Norman, and D. Parker. A Framework for Verification of Software with Time and Probabilities. In *FORMATS*, LNCS. Springer, 2010. To appear.
19. M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *CAV'11*, LNCS. Springer, 2011. To appear.
20. M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance Analysis of Probabilistic Timed Automata using Digital Clocks. *FMSD*, 29:33–78, 2006.
21. M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic Verification of Real-time Systems with Discrete Probability Distributions. *Theoretical Computer Science*, 282(1):101–150, 2002.
22. M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic Model Checking for Probabilistic Timed Automata. *Information and Computation*, 205(7):1027–1077, 2007.
23. B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *ICSE*, pages 95–104, 1998.
24. J. Ouaknine and J. Worrell. Timed CSP = Closed Timed Safety Automata. *Electrical Notes Theoretical Computer Science*, 68(2), 2002.
25. S. Schneider. *Concurrent and Real-time Systems*. John Wiley and Sons, 2000.
26. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, volume 5643 of *LNCS*, pages 709–714. Springer.
27. J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *ICFEM*, pages 581–600, 2009.
28. J. Sun, S. Z. Song, and Y. Liu. Model Checking Hierarchical Probabilistic Systems. In *ICFEM 2010*, volume 6447 of *LNCS*, pages 388–403. Springer, 2010.
29. J. Sun, S. Z. Song, Y. Liu, and J. S. Dong. PRTS: Specification and Model Checking. Technical report, 2010. `http://www.comp.nus.edu.sg/~pat/preport.pdf`.