Model Checking Hierarchical Probabilistic Systems*

Jun Sun[1], Songzheng Song[3] and Yang Liu[2]

[1] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] National University of Singapore
liuyang@comp.nus.edu.sg
[3] NUS Graduate School for Integrative Sciences and Engineering
songsongzheng@nus.edu.sg

Abstract. Probabilistic modeling is important for random distributed algorithms, bio-systems or decision processes. Probabilistic model checking is a systematic way of analyzing finite-state probabilistic models. Existing probabilistic model checkers have been designed for simple systems without hierarchy. In this paper, we extend the PAT toolkit to support probabilistic model checking of hierarchical complex systems. We propose to use PCSP#, a combination of Hoare's CSP with data and probability, to model such systems. In addition to temporal logic, we allow complex safety properties to be specified by non-probabilistic PCSP# model. Validity of the properties (with probability) is established by refinement checking. Furthermore, we show that refinement checking can be applied to verify probabilistic systems against safety/co-safety temporal logic properties efficiently. We demonstrate the usability and scalability of the extended PAT checker via automated verification of benchmark systems and comparison with state-of-art probabilistic model checkers.

1 Introduction

Probabilistic systems are common in practice, e.g., randomized algorithms, unreliable system components, unpredictable environment, etc. Probabilistic model checking is a systematic way of analyzing finite-state probabilistic systems. Given a finite-state model of a probabilistic system and a property, a probabilistic model checker calculates the (range of) probability that the model satisfies the property. It has been proven useful in a variety of domains (see examples in [14]).

Designing and verifying probabilistic systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. Existing probabilistic model checkers have been designed for hierarchically simple systems. For instance, the popular PRISM checker [14] supports a simple state-based language, based on the Reactive Modules formalism of Alur and Henzinger [2]. The MRMC checker supports a rather simple input language too [16]. The input language of the LiQuor checker [10], named Probmela, is based on an extension of Promela supported by the SPIN model checker. None of the above checkers supports analysis of hierarchical complex probabilistic systems.

* This research was partially supported by a grant "SRG ISTD 2010 001" from Singapore University of Technology and Design.

In this work, we aim to develop a useful tool for verifying hierarchical complex probabilistic systems. Firstly, we propose a language called PCSP# for system modeling. PCSP# is an expressive language, combining Hoare's CSP [15], data structures, and probabilistic choices. It extends previous work on combining CSP with probabilistic choice [22] or on combining CSP with data structures [30]. PCSP# combines low-level programs, e.g., sequence programs defined in a simple imperative language or any C# program, with high-level specifications (with process constructs like parallel, choice, hiding, etc.), as well as probabilistic choices. It supports shared variables as well as abstract events, making it both state-based and event-based. Its underlying semantics is based on Markov Decision Processes (MDP) [6].

Secondly, we propose to verify complex safety properties by showing a refinement relationship (with probability) from a PCSP# model representing a system and a non-probabilistic model representing properties. Note that we assume that the property model is non-probabilistic. We view probability as a necessary devil forced upon us by the unreliability of the system or its environment. In contrast, properties which characterizes correct system behaviors are often irrelevant of the likelihood of some low-level failures. Refinement checking has been traditionally used to verify variants of CSP [26, 27]. It has been proven useful by the success of the FDR checker [27]. Verification of such properties are reduced to the problem of probabilistic model checking against deterministic finite automata, which has been previously solved (see for example [4]). Nonetheless, we present a slightly improved algorithm which is better suited for our setting. Alternatively, properties can be stated in form of state/event linear temporal logic (SE-LTL) [8]. An SE-LTL formula can be built from not only atomic state propositions but also events, making it a perfect property specification language for PCSP#, which is both state-based and event-based. A standard method for model checking SE-LTL formulae is the automata-based approach [4]. In this paper, we improve it by safety/co-safety recognition. That is, if an LTL formula or its negation is recognized as a safety property, then the model checking problem is reduced to a refinement checking problem and solved using our refinement checking algorithm. Though the worse-case complexity remains the same, we show that safety/co-safety recognition offers significantly memory/time saving in practice. Lastly, we extend the PAT model checker with all the techniques so as to offer a self-contained framework for probabilistic system modeling, simulation (using the built-in visualized simulator), and verification. In order to demonstrate the usability/scalability of our approach, we verify benchmark systems and compare the results with the PRISM checker [14].

Related work This work is related to methods and tools for probabilistic system modeling and verification. Existing probabilistic model checkers include at least PRISM [14], MRMC [16] and LiQuor [10]. PRISM is the most popular probabilistic model checker. It supports a variety of probabilistic models as well as property specification languages. The input of PRISM is a simple state-based language [2]. LiQuor is a probabilistic model checker for reactive systems [10]. MRMC is a command-line based model checker for a variety of probabilistic models and a rather simple input language. The extended PAT checker complements the existing checkers by 1) offering a language that is both state-based and event-based and is capable of modeling hierarchical systems; 2) supporting both SE-LTL model checking and probabilistic refinement checking and 3) offering a user-friendly environment for not only model checking but also simulation.

The language PCSP# is related to many works on integrating probabilistic behaviors into process algebras or programs, among which the most relevant are [22, 21, 9, 33]. In [22], an extension of CSP is proposed to incorporate probabilistic behaviors in the name of refinement checking. In [21], issues on integrating probability with Event-B has been discussed. In [9], issues on integrating probability with non-determinism have been addressed. Compared to [22, 21, 9, 33], this work focuses on developing a practical tool for systematic modeling and verification of probabilistic systems.

Our work on improving temporal logic model checking with safety recognition is related to work on categorizing safety and liveness. The work presented in [1] offers theoretical results for recognizing safety and liveness given a Büchi automaton. Others have also considered the problem of model checking safety LTL properties. In [28], a categorization of safety, liveness and fairness is discussed. Further, it showed that recognizing safety LTL properties is PSPACE-complete. Later, many theoretical results and algorithms have been presented in [17], which generalizes the earlier work presented in [28]. A forward direction version of the algorithm in [17] is evidenced in [12]. In [18], the author presented a translation of safety LTL formula to a finite state automaton which detects bad prefixes. Model checking safety properties expressed using past temporal operators has been considered in [13]. Our safety recognition is based on [1, 28]. Different from the above, we present methods/algorithms which improve model checking of not only safety properties but also a class of liveness properties; not only finite state systems but also probabilistic systems.

Organization The remainder of the paper is organized as follows. Section 2 presents relevant technical definitions. Section 3 introduces the syntax and semantics of PCSP#. Section 4 presents probabilistic verification of PCSP# models. In particular, Section 4.1 presents a refinement checking algorithm. Section 4.2 presents our approach for verifying SE-LTL formulae with safety recognition. Section 5 evaluates our methods. Section 6 concludes the paper with future research directions.

2 Preliminaries

LTS A labeled transition system (LTS) $\mathcal{L}$ is a tuple $(S, init, Act, T)$ where S is a finite set of states; and $init \in S$ is an initial state; Act is an alphabet; $T \subseteq S \times Act \times S$ is a labeled transition relation. A transition label can be either a visible event or an invisible one (which is referred to as τ). A τ-transition is a transition labeled with τ. For simplicity, we write $s \xrightarrow{e} s'$ to denote $(s, e, s') \in T$. If $s \xrightarrow{e} s'$, then we say that e is enabled at s. Let $s \leadsto s'$ to denote that s' can be reached from s via zero or more τ-transitions; we write $s \overset{e}{\leadsto} s'$ to denote there exists s_0 and s_1 such that $s \leadsto s_0 \xrightarrow{e} s_1 \leadsto s'$. A path of $\mathcal{L}$ is a sequence of alternating states/events $\pi = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all i. The set of path of $\mathcal{L}$ is written as $paths(\mathcal{L})$. Given a path π, we can obtain a sequence of visible events by omitting states and τ-events. The sequence, written as $trace(\pi)$, is a trace of $\mathcal{L}$. The set of traces of $\mathcal{L}$ is written as $traces(\mathcal{L}) = \{trace(\pi) \mid \pi \in paths(\mathcal{L})\}$. An LTS is deterministic if and only if given any s and e, there exists only one s' such that $s \xrightarrow{e} s'$. An LTS is non-deterministic if and only if it is not deterministic. A non-deterministic LTS can be translated into a trace-equivalent deterministic LTS by determinization. Furthermore, non-deterministic LTSs containing τ-transitions can be translated into trace-equivalent deterministic LTSs without τ-transitions. The process is known as normalization [26].

Definition 1 (Normalization). *Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. The normalized LTS of $\mathcal{L}$ is $nl(\mathcal{L}) = (S', init', Act, T')$ where $S' \subseteq 2^S$ is a set of sets of states, $init' = \{s \mid init \rightsquigarrow s\}$ and T' is a transition relation satisfying the following condition: $(N, e, N') \in T'$ if and only if $N' = \{s' \mid \exists\, s : N.\ s \overset{e}{\rightsquigarrow} s'\}$.*

Normalization is to group states which can be reached via the same trace. Given two LTSs $\mathcal{L}_0$ and $\mathcal{L}_1$, it is often useful to check whether $traces(\mathcal{L}_0)$ is a subset of $traces(\mathcal{L}_1)$ (or equivalently $\mathcal{L}_0$ trace-refines $\mathcal{L}_1$). There are existing algorithms and tools for trace inclusion check [26]. The idea is to construct the product of $\mathcal{L}_0$ and $nl(\mathcal{L}_1)$ and then search for a state of the form (s, s') such that s enables more visible events than s' does. In the worse case, this algorithm is exponential in the number of states of $\mathcal{L}_1$. It is nonetheless proven to be practical for real-world systems by the success of the FDR checker [27].

SE-LTL LTL was introduced to specify the properties of executions of a system [24]. It is built up from a set of *propositions* using standard Boolean operators ($\neg$, $\wedge$, $\vee$) and X (next), U (until), R (release), $\Diamond$ (eventually) and $\Box$ (always). It has been adopted for specifying properties in many systems. In [8], LTL is extended to build up from not only state propositions but also events. The extended LTL is referred to as SE-LTL. The simplicity of writing formulas concerning events as in the above example is not purely a matter of aesthetics. It may yield gains in time and space [8].

LTL (and SE-LTL) formulae can be categorized into either safety or liveness. Informally speaking, safety properties stipulate that "bad things" do not happen during system execution. A finite execution is sufficient evidence to the violation of a safety property. In contrast, liveness properties stipulate that "good things" do happen eventually. A counterexample to a liveness property is an infinite system execution (which forms a loop if the system has finitely many states). In this paper, we adopt the definition of safety and liveness in [1]. For instance, $\Box(a \Rightarrow \Box b)$ and $\Diamond a \Rightarrow \Box b$ are safety properties; $\Box a \Rightarrow \Diamond b$ is a liveness property, whose negation, however, is a safety property. A liveness property whose negation is safety is referred to as co-safety, e.g., $\Diamond a$ is co-safety. We remark that a formula may be neither safety nor liveness, e.g., $\Box\Diamond a \wedge \Box b$. It has been shown in [28] that recognizing whether an LTL formula is safety is PSPACE-compele. A number of methods have been proposed to identify subsets of safety. For instance, *syntactic LTL safety formulae* (which is constituted by $\wedge$, $\vee$, $\Box$, U, X, and propositions or negations of propositions) can be recognized efficiently. A number of methods have been proposed to translate safety LTL to finite state automata [17, 18].

It has been proved in [32] that for every LTL formula ϕ, there exists an equivalent Büchi Automaton. There are many sophisticated algorithms on translating LTL to an equivalent Büchi automaton [11, 29]. In addition, it is possible to tell whether an LTL formula represents safety by examining its equivalent Büchi automaton. For instance, it has been proved in [1] that a (reduced) Büchi automaton specifies a safety property if and only if making all of its states accepting does not change its language. Based on this result, *a Büchi automaton representing a safety property can be viewed as an LTS for simplicity*. The reason is that all of its infinite traces must be accepting and therefore the acceptance condition can be ignored.

3 Hierarchical Modeling

In this section, we present PCSP#, which is designed for modeling and verifying probabilistic systems. We remark that the LiQuor checker, which is based on Probmela, makes a step towards an expressive useful modeling language. Nonetheless, Probmela is not capable of modeling fully hierarchical systems.

Syntax PCSP# extends the CSP# language [30] with probabilistic choices. CSP# integrates low-level programs with high-level compositional specification. It is capable of modeling systems with not only complicated data structures (which are manipulated by the low-level programs) but also hierarchical systems with complex control flows (which are specified by the high-level specification). Compared with PCSP [22], PCSP# supports explicit complex data structures/operations.

A PCSP# model is a 3-tuple $(Var, init, P)$ where Var is a set of global variables (with bounded domains) and channels; $init$ is the initial values of Var; P is a process. A variable can be either of simple types like boolean, integer, arrays of integers or any user-defined data type (which must be defined in an external C# library). The process P is an extension of Hoare's classic CSP. Part of its syntax is defined as follows.

$$\begin{array}{lll}
P ::= & Stop \mid Skip & \text{– primitives} \\
\mid & e \rightarrow P & \text{– event prefixing} \\
\mid & a\{program\} \rightarrow P & \text{– data operation prefix} \\
\mid & P \Box Q \mid P \sqcap Q \mid \textbf{if } b \textbf{ then } P \textbf{ else } Q & \text{– choices} \\
\mid & P;\ Q & \text{– sequence} \\
\mid & P \parallel Q \mid P \mid\mid\mid Q & \text{– concurrency} \\
\mid & P \setminus X & \text{– hiding} \\
\mid & Q & \text{– process referencing} \\
\mid & \textbf{pcase}\ \{d_0 : P_0;\ d_1 : P_1;\ \cdots;\ d_k : P_k\} & \text{– probabilistic multi-choices}
\end{array}$$

where P, P_i and Q range over processes, e is a simple event, a is the name of a sequential program; b is a Boolean expression, d_i is a rational number and $d_0 + d_1 + \cdots + d_k = 1$. Process $Stop$ does nothing. Process $Skip$ terminates. Process $e \rightarrow P$ engages in event e first and then behaves as P. Combined with parallel composition, event e may serve as a multi-party synchronization barrier. Process $a\{program\} \rightarrow P$ generates an event a, executes a sequential program $program$ at the same time, and then behaves as P. External C# data operations can be invoked in $program$.

A variety of choices are supported, e.g., $P \Box Q$ for external choice; $P \sqcap$ for internal non-determinism and **if** b **then** P **else** Q for conditional branching. 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 channels or variables, then it is written as $P \mid\mid\mid Q$. Process $P \setminus X$ hides occurrence of any event in X. Recursion is supported through process referencing. Lastly, probabilistic choice is written in the form of **pcase** $\{d_0 : P_0;\ d_1 : P_1;\ \cdots;\ d_k : P_k\}$. Intuitively, it means that with d_i probability, the system behaves as P_i. It is required that $d_0 + d_1 + \cdots + d_k = 1$.

Example 1 (Pacemaker). A pacemaker is an electronic implanted device which functions to regulate the heart beat by electrically stimulating the heart to contract and thus to pump blood throughout the body. Common pacemakers are designed to correct bradycardia, i.e., slow heart beats. A pacemaker mainly performs two functions, i.e., sensing and pacing. Sensing is to monitor the heart's natural electrical activity, helping the pacemaker to gather information on the heart beats and react accordingly. Pacing is when a pacemaker sends electrical stimuli, i.e., tiny electrical signals, to heart through a pacing lead, which starts a heart beat. A pacemaker can operate in many different modes, according to the implanted patient's heart problem. The following is a high-level abstraction of the simplest mode of pacemaker, i.e., the AAT mode.

```
**var** $count = 0;$
$AAT$      $= (Heart \parallel Pacing) \setminus \{missingPulseA, missingPulseV\}$
$Heart$    $=$ **pcase** $\{$
               $[pA] : missingPulseA \rightarrow pulseV \rightarrow Heart$
               $[pV] : pulseA \rightarrow missingPulseV \rightarrow Heart$
               $[1 - pA - pV] : pulseA \rightarrow pulseV \rightarrow Heart$
           $\};$
$Pacing$   $= pulseA \rightarrow Pacing \ \Box\ pulseV \rightarrow Pacing$
           $\Box\ missingPulseA \rightarrow add\{count++\} \rightarrow$ **pcase** $\{$
                  $[99.54] : pulseB\{count--\} \rightarrow Pacing$
                  $[0.46] : Pacing$
           $\}$
           $\Box\ missingPulseV \rightarrow add\{count++\} \rightarrow$ **pcase** $\{$
                  $[99.54] : pulseW\{count--\} \rightarrow Pacing$
                  $[0.46] : Pacing$
           $\};$
```

Variable *count* is an integer (with a default bound) which records the number of skipped pulses. A (mode of the) pacemaker is typically modeled in the following form: $Heart \parallel Pacing$ where *Heart* models normal or abnormal heart condition; *Pacing* models how the pacemaker functions. In this particular mode, process *Heart* generates two events *pulseA* (i.e., atrium does a pulse) and *pulseV* (i.e., ventricle does a pulse), periodically for a normal heart or with one of them missing once a while for an abnormal heart. In the latter case, event *missingPulseA* or *missingPulseV* is generated. Constant pA is the (patient-dependent) probability of *pulseA* missing; pV is the probability of *pulseV* missing. Process *Pacing* synchronizes with process *Heart*. If event *missingPulseA* (denoting the missing of event *pulseA*) is monitored, variable *count* is incremented by one. Notice that the event *add* is associated with the simple program of updating *count*. In general, it can be associated with any state update function. It is in this way that state update is introduced in an event-based language. Ideally, the pacemaker helps the heart to beat by generating event $pulseB$. Once $pulseB$ is generated, $count$ is decremented by one. Similarly, it generates $pulseW$ when $pulseV$ is missing. Note that it has been reported that pacemaker may malfunction for certain rate (exactly 0.46%) [20]. This is reflected in the model again using **pcase**. If a $pulseB$ or $pulseW$ is skipped, $count$ is not decremented.

At the top level, the pacemaker system is a choice of different modes. Each mode is often a parallel composition of multiple components. Each component may have internally hierarchies due to complicated sensing and pacing behaviors. We skip the details (refer to [5]) and remark that our modeling language is more suitable for such systems than those supported by existing probabilistic model checkers. $\square$

Semantics The semantic model of CSP# (without **pcase**) is LTS. In this paper, we assume that all variables have finite domain and the set of reachable process expressions are finite so that the LTS has finitely states. A state in the LTS is a tuple of the form (V, P) where V is the valuation of the variables and P is a process expression. Given a CSP# model $\mathcal{M}$, its LTS can be generated systematically following its structural operational semantics (also known as firing rules [30]). A firing rule for CSP# is of the form $(V, P) \xrightarrow{e} (V', P')$. Based on the LTS, different semantic objects can be defined. For instance, the traces of $\mathcal{M}$ are defined to be the traces of the LTS.

The underlying semantics of PCSP# is Markov Decision Process (MDP), which is expressive enough to capture systems with probabilistic choices as well as nondeterminism and concurrency. Given a set of states S, a distribution is a function $\mu : S \rightarrow [0, 1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S. An MDP is a 3-tuple $\mathcal{M} = (S, init, Pr)$ where S is a set of system states; $init \in S$ is the initial system configuration[4]; $Pr : S \times Act \times Distr(S)$ is a transition relation[5]. A transition of the system is written as $s \xrightarrow{e} \mu$ where μ is a distribution, or equivalently $s \xrightarrow{e} \{(s_1, d_1), (s_2, d_2), \cdots\}$ where $s \in S$ and $s_i \in S$ for all i; and $d_i : [0, 1]$ is the probability of reaching s_i given the distribution. A path of $\mathcal{M}$ is a sequence of alternating states, events and distributions $\pi = \langle s_0, e_0, \mu_0, s_1, e_1, \mu_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} \mu_i$ and $\mu_i(s_{i+1}) > 0$ for all i. The probability of exhibiting π by $\mathcal{M}$, denoted as $\mathcal{P}_{\mathcal{M}}(\pi)$, is $\mu_0(s_1) * \mu_1(s_2) * \cdots$. Given a path π, we define $trace(\pi)$ to be the sequence of visible events in π. Let $paths(\mathcal{M})$ denote all paths of $\mathcal{M}$. In an abuse of notation, let $s \in \pi$ denote that π visits state s.

In the following, *we assume that MDPs are deadlock-free* following common practice. A deadlocking model can be made deadlock-free by adding a special self loop to the deadlock states, without affecting the result of probabilistic verification. Intuitively speaking, given a system configuration, firstly an event and a distribution is selected nondeterministically by the scheduler, and then one of successor states is reached according to the probability distribution. A scheduler is a function decides which event and distribution to choose based on the execution history (in the form of a path). A Markov Chain [4] can be defined given an MDP $\mathcal{M}$ and a scheduler δ, denoted as $\mathcal{M}_\delta$. Intuitively, a Markov Chain is an MDP where only one event and

[4] This is a simplified definition. In general, there can be an initial distribution.

[5] This is slightly different from the classic definition of MDP.

Rule for any process constructs in CSP#:

$$\frac{(V, P) \xrightarrow{e} (V', P') \text{ is a firing rule of CSP\#}}{(V, P) \xrightarrow{e} \mu \text{ such that } \mu((V', P')) = 1}$$

Rule for **pcase**

$$\frac{}{(V, \textbf{pcase}\,\{d_0 : P_0;\ \cdots;\ d_k : P_k\}) \xrightarrow{\tau} \mu \text{ such that } \mu((V, P_i)) = d_i \text{ for all } i}$$

Fig. 1. Firing Rules

distribution is enabled at every state. For simplicity, we write $\mathcal{P}^{\delta}_{\mathcal{M}}(\pi)$ to denote the probability of exhibiting a path π in $\mathcal{M}$ with scheduler δ. The probability of exhibiting a set of path $X \subseteq paths(\mathcal{M}_\delta)$ is the accumulated probability of each path, i.e., $\mathcal{P}^{\delta}_{\mathcal{M}}(X) = \Sigma_{x \in X}\mathcal{P}^{\delta}_{\mathcal{M}}(x)$. It is often useful to find out the probability of reaching a set of states. Note that with different scheduling, the probability may be different. The measurement of interest is thus the maximum and minimum probability. Given a set of target states G, the maximum probability of reaching any state in G is defined as

$$\mathcal{P}^{max}_{\mathcal{M}}(G) = \sup_\delta Pr^{\delta}_{\mathcal{M}}(\{\pi \mid \exists\, s \in G.\ s \in \pi\})$$

Note that the supremum ranges over all, potentially infinitely many, schedulers. Accordingly, the minimum is written as $\mathcal{P}^{min}_{\mathcal{M}}(G)$. Similarly, we define the maximum probability of exhibiting a trace in a set Tr by $\mathcal{M}$.

$$\mathcal{P}^{max}_{\mathcal{M}}(Tr) = \sup_\delta Pr^{\delta}_{\mathcal{M}}(\{\pi \mid trace(\pi) \in Tr\})$$

Accordingly, the minimum is written as $\mathcal{P}^{min}_{\mathcal{M}}(Tr)$.

Next, we define firing rules for PCSP#. Figure 1 presents all the necessary rules. The first rule states that if by CSP# firing rules, $(V, P) \xrightarrow{e} (V', P')$, then configuration (V, P) can perform e and result in one distribution which maps configuration (V', P') to 1. The second rule is for **pcase**. The result distribution associates d_k probability with (V, P_k) for all k. Notice that V remains unmodified during the transition. We remark that only τ-transitions can be associated with probability other than 1. Following these two rules, an MDP can be generated from a model systematically.

4 Probabilistic Refinement Checking

Refinement checking has been traditionally used to verify CSP [15]. Different from temporal-logic based model checking, refinement checking works by taking a model (often in the same language) as a property. The property is verified by showing a refinement relationship from the system model to the property model. There are different refinement relationships designed for proving different properties. In the following, we focus on trace refinement and remark that our approach can be extended to stable failures refinement or failures/divergence refinement. For instance, one way of verifying the pacemaker is to check whether the pacemaker model (present in Example 1) trace-refines the following model (without variables) which models a 'fine' heart.

$$\begin{aligned} OKHrt = {} & pulseA \rightarrow pulseV \rightarrow OKHrt \,\Box\, pulseA \rightarrow pulseW \rightarrow OKHrt \,\Box \\ & pulseB \rightarrow pulseV \rightarrow OKHrt \,\Box\, pulseB \rightarrow pulseW \rightarrow OKHrt \end{aligned}$$

In theory, it is possible to encode the property model as temporal logic formulae (as temporal logic is typically more expressive than LTS) and then apply temporal-logic based model checking to verify the property. It is, however, impractical. For instance, LTL model checking is exponential in the size of the formulae and therefore it cannot handle formulae which encode non-trivial property model. In short, refinement checking allows users to verify a different class of properties from temporal logic formulae.

4.1 Refinement Checking PCSP#

Because of probabilistic choices, refinement checking in our setting is not simply to verify whether traces of a PCSP# model is subset of those of another. Instead, it is 'how likely' the system behaves as specified by the property model (in the presence of unreliability of system components). Because we assume the property model is non-probabilistic, the problem is thus to calculate the probability of an MDP (i.e., the semantics of PCSP# model) trace-refines an LTS (i.e., the semantics of a non-probabilistic PCSP# model).

Definition 2 (Refinement Probability). *Let $\mathcal{M}$ be an MDP and $\mathcal{L}$ be an LTS. The maximum probability of $\mathcal{M}$ trace-refines $\mathcal{L}$ is defined by $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L}) = \mathcal{P}^{max}_{\mathcal{M}}(traces(\mathcal{L}))$. The minimum is defined by $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L}) = \mathcal{P}^{min}_{\mathcal{M}}(traces(\mathcal{L}))$.* □

Intuitively, the probability of $\mathcal{M}$ refines $\mathcal{L}$ is the sum of the probability of $\mathcal{M}$ exhibiting every trace of $\mathcal{L}$. The probability may vary due to different scheduling. One way of calculating the maximum/minimum probability [4] is to (1) build a deterministic LTS $\mathcal{L}^{-1}$ which complements $\mathcal{L}$ (such that $traces(\mathcal{L}^{-1}) = \Sigma^* \setminus traces(\mathcal{L})$); (2) compute the product of $\mathcal{M}$ and $\mathcal{L}^{-1}$; 3) calculate the maximum/minimum probability of paths of the product.

In the following, we present a slightly improved algorithm which avoids the construction of $\mathcal{L}^{-1}$. Note that for a complicated language like PCSP#, computing $\mathcal{L}^{-1}$ is highly nontrivial. The algorithm is inspired by the refinement checking algorithm in FDR. Firstly, we normalize $\mathcal{L}$ using the standard powerset construction. Next, we compute the synchronous product of $\mathcal{M}$ and $nl(\mathcal{L})$, written as $\mathcal{M} \times nl(\mathcal{L})$. It can be shown that the product is an MDP.

Definition 3 (Product MDP). *Let $\mathcal{M} = (S_{\mathcal{M}}, init_{\mathcal{M}}, Act, Pr_{\mathcal{M}})$ be an MDP and $\mathcal{L} = (S_{\mathcal{L}}, init_{\mathcal{L}}, Act, T_{\mathcal{L}})$ be a deterministic LTS without τ-transitions. The product is the MDP $\mathcal{M} \times \mathcal{L} = (S_{\mathcal{M}} \times S_{\mathcal{L}}, (init_{\mathcal{M}}, init_{\mathcal{L}}), Act, Pr)$ such that Pr is the least transition relation which satisfies the following conditions.*

– *If $s_m \xrightarrow{\tau} \mu$ in $\mathcal{M}$, then $(s_m, s_l) \xrightarrow{\tau} \mu'$ in $\mathcal{M} \times \mathcal{L}$ for all $s_l \in S_{\mathcal{L}}$ such that $\mu'((s'_m, s_l)) = \mu(s'_m)$ for all $s'_m \in S_{\mathcal{M}}$.*

– *If* $s_m \xrightarrow{e} \mu$ *in* $\mathcal{M}$ *and* $s_l \xrightarrow{e} s'_l$ *in* $\mathcal{L}$*, then* $(s_m, s_l) \xrightarrow{e} \mu'$ *in* $\mathcal{M} \times \mathcal{L}$ *such that* $\mu'((s'_m, s'_l)) = \mu(s'_m)$ *for all* $s'_m \in S_\mathcal{M}$.

In the product, there are two kinds of transitions, i.e., τ-transitions from $\mathcal{M}$ with the same probability or transitions labeled with a visible event with probability 1. Note that τ-transitions are not synchronized, whereas visible events must be jointly performed by $\mathcal{M}$ and $\mathcal{L}$. Let $G \subseteq S_\mathcal{M} \times S_\mathcal{L}$ be the least set of states satisfying the following condition: for every pair $(s, s') \in G$, $s' = \varnothing$. Intuitively, $(s, s') \in G$ if and only if a trace of $\mathcal{M}$ leading to s is not possible in $\mathcal{L}$. The following theorem states our main result on refinement checking.

Theorem 1. *Let* $\mathcal{M}$ *be an MDP;* $\mathcal{L}$ *be an LTS;* $\mathcal{D} = \mathcal{M} \times nl(\mathcal{L})$. $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L}) = 1 - \mathcal{P}^{min}_\mathcal{D}(G)$ *and* $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L}) = 1 - \mathcal{P}^{max}_\mathcal{D}(G)$.

Proof Let δ be any scheduler for $\mathcal{M}$. Note that δ can be extended to be a scheduler for $\mathcal{D}$ straightforwardly. For simplicity, we use δ to denote both of them. Let $X \subseteq paths(\mathcal{M})$. The following shows that the equivalence holds with any scheduler.

$$\begin{array}{ll}
\mathcal{P}^\delta_M(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \in traces(\mathcal{L})\}) & \\
\equiv 1 - \mathcal{P}^\delta_M(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \notin traces(\mathcal{L})\}) & \text{– by def.} \\
\equiv 1 - \mathcal{P}^\delta_\mathcal{D}(\{\pi \in paths(\mathcal{D}) \mid trace(\pi) \notin traces(\mathcal{L})\}) & \text{– (1)} \\
\equiv 1 - \mathcal{P}^\delta_\mathcal{D}(G) & \text{– (2)}
\end{array}$$

(1) is true because for every path of $\mathcal{M}$, there is a path of $\mathcal{D}$ with the same probability (as $\mathcal{L}$ is non-probabilistic) and the same trace; and vice versa. (2) is true because by [26], a path of $\mathcal{D}$ such that $trace(\pi) \notin traces(\mathcal{L})$ if and only if it visits some state in G. It can be shown then $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L})$, which is $\mathcal{P}^{max}_M(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \in traces(\mathcal{L})\})$, is $1 - \mathcal{P}^{min}_\mathcal{D}(G)$ and $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L})$ is $1 - \mathcal{P}^{max}_\mathcal{D}(G)$. □

Intuitively, the theorem holds because, with any scheduler, the probability of $\mathcal{M}$ *not* refining $\mathcal{L}$ is exactly the probability of reaching G in $\mathcal{M} \times nl(\mathcal{L})$. As a result, refinement checking is reduced to reachability probability in $\mathcal{D}$. There are known approaches to compute $\mathcal{P}^{max}_\mathcal{M}(G)$ and $\mathcal{P}^{min}_\mathcal{M}(G)$, e.g., using an iterative approximation method or by solving linear programs [4].

4.2 SE-LTL Probabilistic Model Checking as Refinement Checking

Another way of specifying properties is through temporal logic. In this section, we examine the problem of model checking PCSP# models against SE-LTL formulae. SE-LTL is an effective property language for PCSP# as it can be constituted by state propositions as well as events. In the pacemaker example, an SE-LTL formula could be stated as follows: $(\Box count \leq 10) \land \Box(missingPulseA \Rightarrow X\ pulseB)$ which states $count$ must be always less than 10 and event $missingPulseA$ must lead to an occurrence of event $pulseB$ next. Given an MDP $\mathcal{M}$ and an SE-LTL formula ϕ, let $\mathcal{P}^{max}_\mathcal{M}(\phi)$ (and $\mathcal{P}^{min}_\mathcal{M}(\phi)$) denote the maximum (and minimum) probability of $\mathcal{M}$ satisfying ϕ.

A standard LTL probabilistic model checking method is the automata-theoretic approach [4]. Firstly, a deterministic Rabin automaton, which is equivalent to the property, is built. The product of the automaton and the system model is then computed.

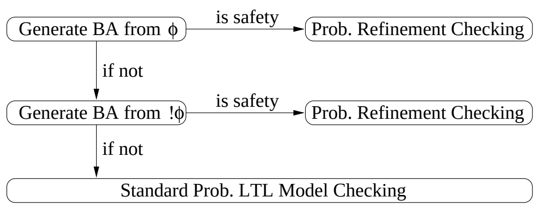
Generate BA from ϕ
is safety
Prob. Refinement Checking
if not
Generate BA from !ϕ
is safety
Prob. Refinement Checking
if not
Standard Prob. LTL Model Checking

Fig. 2. Workflow

Thirdly, *end components* (which is similar to strongly connected components) in the product which satisfy the Rabin acceptance condition are identified. Lastly, the probability of reaching any state of the end components are calculated, which is exactly the probability of the model satisfying the property. This method is computationally expensive due to multiple reasons. Firstly, the construction of the deterministic Rabin automaton is expensive. Given a Büchi automaton $\mathcal{B}$, its equivalent deterministic Rabin automaton, in the worse case, is of size $2^{\mathcal{O}(n \log n)}$ where n is the size of $\mathcal{B}$. Secondly, identifying the end components is expensive. The worse case complexity is bounded by $\#S \times (\#S + \#T)$ where $\#S$ is the number system states and $\#T$ is the number of the system transitions. In this section, we show that by recognizing safety properties, we can improve probabilistic model checking of certain class of SE-LTL formulae by avoiding constructing the Rabin automaton or computing the end components.

Given a formula ϕ, we check whether ϕ is a safety property using the following approach. Firstly, we check whether it is a *syntactic LTL safety formula* [28]. If it is not, we generate an equivalent Büchi automaton using an existing approach [11], and then check whether all states of the Büchi automaton are accepting. If positive, by the result proved in [1], ϕ is a safety property. If we cannot conclude that ϕ is safety, we assume that it is not. This is a sound but not complete method for recognizing safety. In practice, we found that it is effective in recognizing most of the commonly used safety properties, including for example $\Box(a \Rightarrow \Box b)$ and $\Diamond a \Rightarrow \Box b$.

Next, we adopt the workflow shown in Figure 2 to improve probabilistic model checking. Let ϕ be an SE-LTL formula and $\mathcal{B}$ be the equivalent Büchi automaton. If ϕ is a safety property, then $\mathcal{B}$ can be simply treated as an LTS, as discussed in Section 2. The problem of model checking a system model $\mathcal{M}$ against ϕ is thus reduced to calculate the probability of $\mathcal{M}$ refines the LTS $\mathcal{B}$. If ϕ cannot be determined as a safety property, then we check whether ϕ is a co-safety property. A Büchi automaton $\mathcal{B}'$, equivalent to $\neg\,\phi$, is generated. If $\mathcal{B}'$ is a safety property, the problem of model checking ϕ is thus reduced to calculate the probability of $\mathcal{M}$ refines the LTS $\mathcal{B}'$.

Theorem 2. *Let $\mathcal{M}$ be an MDP; ϕ be an SE-LTL formula; $\mathcal{B}$ be the Büchi automaton equivalent to ϕ. Let $\mathcal{B}^{-1}$ be the Büchi automaton equivalent to $\neg\,\phi$. If ϕ is safety, then $\mathcal{P}^{max}_{\mathcal{M}}(\phi) = \mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{B})$ and $\mathcal{P}^{min}_{\mathcal{M}}(\phi) = \mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{B})$; If ϕ is co-safety, then $\mathcal{P}^{max}_{\mathcal{M}}(\phi) = 1 - \mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{B}^{-1})$ and $\mathcal{P}^{min}_{\mathcal{M}}(\phi) = 1 - \mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{B}^{-1})$.* $\Box$

The proof of the theorem is sketched as follows. If ϕ is a safety property, any trace of $\mathcal{B}$ is accepting. It can be shown that any trace of $\mathcal{M}$ which is not a trace of $\mathcal{B}$ is a counterexample to ϕ. Therefore, the probability of $\mathcal{M}$ exhibiting a trace of $\mathcal{B}$ (i.e., the probability of $\mathcal{M}$ trace-refines $\mathcal{B}$) is the probability of $\mathcal{M}$ satisfying ϕ. Next, the theorem states that the probability of $\mathcal{M}$ not-refining $\mathcal{B}$ is the probability of $\mathcal{M}$ executing a finite prefix of any traces which is not possible for $\mathcal{B}$. Similarly, we can prove the result for co-safety properties.

By the theorem, probabilistic model checking of safety LTL formula or co-safety LTL formula is reduced to probabilistic refinement checking, which is considerably more efficient as we avoid constructing the deterministic Rabin automaton or identifying end components. This is confirmed by the experiments conducted in Section 5.

5 Case Studies

Our methods have been implemented in the PAT[6] model checker [31]. PAT is a self-contained framework for system modeling, simulation and verification. It supports a layered system design so that new modeling languages and new model checking algorithms/techniques can be easily incorporated. In this paper, we extend PAT with a module to support PCSP#, integrating the existing CSP# language with probabilistic choices. Furthermore, we extend the library of model checking algorithms in PAT with probabilistic refinement checking and probabilistic SE-LTL model checking with safety recognition. We evaluate our implementation using benchmark systems. We compare our results with PRISM version 3.3.1. In order to perform a fair comparison, we use existing PRISM models; re-model them using the extended CSP# language and re-verify them using PAT. It should be noticed that our language is capable of specifying hierarchical systems which are beyond PRISM. Working with existing PRISM models, which are not hierarchical, is not justified to show our advantage. Nonetheless, we show that even for those systems, PCSP# offers an intuitive and compact representation and PAT offers comparable performance. The following models are adopted for comparison.

- Model ME describes a probabilistic solution to N-process mutual exclusion problem, which is based on [25].
- Model RC is a shared coin protocol of the randomized consensus algorithm, which is based on [3]. Note that N is the number of coins and K is a parameter used to generate suitable probability.
- Model DP is the probabilistic N-dining philosophers under fairness, based on [19].
- Model CS is the IEEE 802.3 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol, which is based on [23]. Note that N is the number of stations and K is the exponential backoff limit.

The models (and others) with configurable parameters are embedded in the latest version of PAT. In the following, we discuss three aspects of the comparison.

Comparison on modeling The simplicity of writing models is not purely a matter of aesthetics. It may yield gains in time and space. Table 1 presents the size of the models (in number lines of codes) as well as the number of global states. The size of all models are reduced. Note that with different parameters, the PRISM models vary in sizes,

[6] Available at http://pat.comp.nus.edu.sg

System	PAT			PRISM		
	LOC	#States	Deadlock Check(s)	LOC	#States	Deadlock Check(s)
ME (N=5)	22	5489	0.351	36	308800	0.410
ME (N=8)	22	86966	7.279	39	390068480	1.203
RC (N=4, K=4)	24	6835	0.218	25	43136	0.110
RC (N=10, K=6)	24	997403	56.072	31	7598460928	3.250
DP (N=5)	20	32766	2.413	30	93068	0.156
DP (N=6)	20	260100	25.775	31	917424	0.672
CS (N=2, K=4)	115	9165	0.337	119	7958	0.266
CS (N=3, K=2)	94	49101	2.243	122	36850	0.772

Table 1. Experiments on modeling

whereas the size of the PAT models remain constant. The state counts for PAT models are significantly smaller than those of the PRIMS models. The state counts are reported by PAT and PRISM when checking deadlock-freeness of both models. One of the reasons why PAT may have much less states is that global variables in the PRISM models, which are used to track local state of each processes, are removed (and become part of the process definition). The processes then become fully symmetric (as expected in the original protocol), which then triggered an internal state reduction based on symmetry reduction in PAT.

Performance of refinement checking In general, refinement checking and temporal logic verification are good at different classes of properties. For instance, using temporal logic formulae to capture the process $OKHrt$ (shown in Section 4) would result in a large formula which in turn result in in-efficient verification. Our experiments, however, show that even for those properties designed for temporal-logic based verification, probabilistic refinement checking offers comparable performance. Given any safety property of the above mentioned models, we build a property model and verify the property by refinement checking. Table 2 presents the experiment results. The experiment data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 2GB RAM. We use the iterative method in calculating the probability and set termination threshold as relative difference 1.0E-6 (exactly same as PRISM). PAT performs worse than PRISM for ME, comparable for RC and better for DP. The main reason that PAT outperforms PRISM for the DP model is that PAT has less states and its refinement checking algorithm has less computation than temporal logic-based model checking. Note that because the models are designed to satisfy the properties, the result probability is all 1.

Performance improvement using safety recognition Lastly, we show that safety recognition improves probabilistic LTL model checking and allows PAT to outperform PRISM in many cases. Safety recognition in PAT is based on syntax analysis or simple heuristics based on the generated Büchi automata. The computational overhead is negligible. Table 3 presents the experiment results on verifying the models against safety, co-safety and properties which are neither. Column $PAT(w)$ ($PAT(w/o)$) shows the time taken with (without) safety recognition. If the property is neither safety or co-safety, safety

System	Property	Result(Pmax)	PAT (s)	PRISM (s)
ME (N=5)	mutual exclusion	1	0.359	0.282
ME (N=8)	mutual exclusion	1	9.831	1.234
ME (N=10)	mutual exclusion	1	81.192	3.127
RC (N=4,K=4)	consensus	1	0.218	0.328
RC (N=6,K=6)	consensus	1	2.813	2.543
RC (N=8,K=8)	consensus	1	19.642	14.584
DP (N=5)	once eat, never hungry	1	3.333	37.769
DP (N=6)	once eat, never hungry	1	53.062	389.334

Table 2. Experiments on refinement checking

recognition becomes computational overhead. The cost is however negligible as evidenced in the table. For safety or co-safety properties, PAT performs better with safety recognition. In comparison with PRISM, PAT outperforms PRISM (for almost all properties) for some models, e.g., ME and RC. This is mainly because the PAT models have much less states, because of the difference in modeling. For some other models (e.g., DP and CS), safety recognition allows PAT to outperform PRISM.

In general, PRISM handles more states per time unit than PAT. This is suggested by the experiment results presented in Table 1, which shows the time for verifying deadlock-freeness. Apart from the fact that PRISM has been optimized for many years, 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, whereas PAT is based on explicit state representation currently. Symbolic methods like BDD are known to handle more states [7]. Applying BDD techniques to hierarchical complex languages like PCSP# is highly non-trivial. It remains as one of our ongoing work. The experiment results are not to be taken as the limit of PAT. The fact that PAT handles less states per time unit does not imply that PAT is always slower than PRISM, as evidenced in the experiments. The main reason is that 1) a system modeled using PRISM may have more states than its model in PCSP# due to its language limitation; 2) safety/co-safety recognition which avoid much computation in probabilistic model checking.

6 Conclusion

The main contribution of this work is the extended PAT model checker which offers a self-contained framework for modeling and checking of hierarchical complex probabilistic systems. Compared to existing probabilistic model checkers, PAT offers an expressive modeling language and an alternative way of probabilistic system verification, i.e., refinement checking. In addition, PAT improves LTL probabilistic model checking by supporting SE-LTL and safety recognition.

As for future research directions, we will explore methods for checking refinement relationship between probabilistic PCSP# models. Furthermore, We are investigating how to combine zone abstraction for real-time systems with probabilistic system behaviors so that we can support real-time probabilistic systems. In addition, in order

System	Property	Result(Pmax)	PAT (w)	PRISM	PAT (w/o)
ME (N=5)	co-safety	1	2.356	231.189	27.411
ME (N=8)	co-safety	1	94.204	-	8901.295
ME (N=10)	co-safety	1	1076.217	-	-
RC (N=4,K=4)	co-safety(1)	0.99935	0.379	21.954	12.150
RC (N=4,K=4)	neither	0.54282	6.106	45.612	6.087
RC (N=4,K=4)	co-safety(2)	0.15604	6.703	35.144	7.868
RC (N=6,K=6)	co-safety(1)	1	5.854	1755.984	585.706
RC (N=6,K=6)	neither	0.53228	457.815	-	442.008
RC (N=6,K=6)	co-safety(2)	0.12493	355.027	-	453.362
RC (N=8,K=8)	co-safety(1)	1	52.906	-	-
RC (N=8,K=8)	neither	0.52537	10179.796	-	10107.268
RC (N=8,K=8)	co-safety(2)	0.10138	5923.086	-	9420.430
DP (N=5)	safety	1	1.162	37.769	10.006
DP (N=6)	safety	1	9.760	389.334	164.423
DP (N=5)	co-safety	1	1.039	38.347	544.307
DP (N=6)	co-safety	1	9.091	384.231	-
CS (N=2, K=4)	co-safety(1)	1	0.615	0.921	0.736
CS (N=2, K=4)	co-safety(2)	0.99902	0.933	2.314	1.034
CS (N=3, K=2)	co-safety(1)	1	6.118	1.733	6.707
CS (N=3, K=2)	co-safety(2)	0.85962	6.284	7.233	7.484

Table 3. Experiments on LTL checking

to tackle the state space explosion problem, optimization techniques like partial order reduction and symmetry reduction will be incorporated.

References

1. B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.
2. R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
3. J. Aspnes and M. Herlihy. Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms*, 15(1):441–460, 1990.
4. C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
5. S. S. Barold, R. X. Stroopbandt, and A. F. Sinnaeve. *Cardiac Pacemakers Step by Step: an Illustrated Guide*. Blachwell Publishing, 2004.
6. R. Bellman. A Markovian Decision Process. *Journal of Mathematics of Mechanics*, 6, 1957.
7. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.
8. S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *IFM*, volume 2999 of *LNCS*, pages 128–147. Springer, 2004.
9. Y. Chen and J. W. Sanders. Unifying Probability with Nondeterminism. In *FM*, volume 5850 of *LNCS*, pages 467–482. Springer, 2009.
10. F. Ciesinski and C. Baier. LiQuor: A Tool for Qualitative and Quantitative Linear Time Analysis of Reactive Systems. In *QEST*, pages 131–132. IEEE Computer Society, 2006.

11. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *CAV*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.
12. M. Geilen. On the Construction of Monitors for Temporal Logic Properties. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.
13. K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *TACAS*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
14. A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
15. C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
16. J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. In *QEST*, pages 167–176. IEEE Computer Society, 2009.
17. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
18. T. Latvala. Efficient Model Checking of Safety Properties. In *SPIN*, volume 2648 of *LNCS*, pages 74–88. Springer, 2003.
19. D. Lehmann and M. Rabin. On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract). In *POPL*, pages 133–138. ACM, 1981.
20. W. H. Maisel, M. Moynahan, B. D. Zuckerman, T. P. Gross, O. H. Tovar, D. Tillman, and D. B. Schultz. Pacemaker and ICD Generator Malfunctions. *The Journal of American Medical Association*, 295(16):1901–1906, 2006.
21. C. Morgan, T. S. Hoang, and J. Abrial. The Challenge of Probabilistic *Event B* - Extended Abstract. In *ZB*, volume 3455 of *LNCS*, pages 162–171. Springer, 2005.
22. C. Morgan, A. McIver, K. Seidel, and J. W. Sanders. Refinement-Oriented Probability for CSP. *Formal Asp. Comput.*, 8(6):617–647, 1996.
23. X. Nicollin, J. Sifakis, and S. Yovine. Compiling Real-time Specifications into Extended Automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, 1992.
24. A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE, 1977.
25. A. Pnueli and L. Zuck. Verification of Multiprocess Probabilistic Protocols. *Distributed Computing*, 1(1):53–72, 1986.
26. A. W. Roscoe. Model-checking CSP. pages 353–378, 1994.
27. A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *TACAS*, volume 1019 of *LNCS*, pages 133–152. Springer, 1995.
28. A. P. Sistla. Safety, Liveness and Fairness in Temporal Logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.
29. F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *CAV*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.
30. J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE*, pages 127–135. IEEE Computer Society, 2009.
31. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, volume 5643 of *LNCS*, pages 709–714. Springer, 2009.
32. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986.
33. H. Zhu, S. Qin, J. He, and J. Bowen. PTSC: Probability, Time and Shared-Variable Concurrency. *International Journal on Innovations in Systems and Software Engineering*, 5(4):271–294, 2009.