Computer Science • 23(4) 2022 https://doi.org/10.7494/csci.2022.23.4.4588

Hichem Debbi

MODELING AND ANALYSIS OF PROBABILISTIC REAL-TIME SYSTEMS THROUGH INTEGRATING EVENT-B AND PROBABILISTIC MODEL CHECKING

Abstract *Event-B is a formal method that is used in the development of safety-critical systems; however, these systems may introduce uncertainty and also need to meet real-time requirements, which make the modeling and analysis of such systems a challenging task. While some works exist that try to extend Event-B with probability and over time, they fail to address both in a single framework. Besides, these works mainly addressed extending the language itself, not integrating extended Event-B with verification. In this paper, we aim to represent both probability and time in the Event-B language, and we will show how such a representation can be automatically translated into the probabilistic timed automata (PTA) that are described in the language of the PRISM probabilistic model checker. This transformation approach would allow us to analyze the probabilistic and time-bounded probabilistic reachability properties of probabilistic real-time systems through probabilistic timed CTL (PTCTL) logic.*

Keywords event-B, probabilistic event-B, real-time probabilistic model checking, PTA, PRISM

Citation Computer Science 23(4) 2022: 545–570

1. Introduction

Event-B [1] is a proof-based formal development framework that has been successfully used in many safety-critical systems. It is based on the abstraction-refinement principle, where the development starts with an abstract model that represents a high-level description of a system and then transforms its abstract model into a more detailed model through a number of refinement steps. However, systems are becoming more and more complex nowadays; this induces uncertainty, which requires dealing with this uncertainty (especially from a modeling perspective). Probability has been investigated in detail in other formal methods such as model checking for modeling stochastic systems and, thus, creating reliable probabilistic semantics, which resulted in the development of successful probabilistic model checkers such as PRISM [17] and MRMC [21]. Contrariwise, we can observe low attention for introducing probability into Event-B.

Extending Event-B to allow for the expression of probabilistic events is considered to be very important and a highly demanded task. For doing this, Abrial et al. [28] relied on probabilistic program semantics that were based on Kozen's original form [22], which was an adaptation of Dijkstra's guarded command language [11]. Other works have also tried to introduce probability into Event-B [4, 14, 33, 34].

Similar to works that have tried to extend Event-B for probabilistic reasoning, some other works also exist that have attempted to extend Event-B for timing constraints; timing constraints are very important for modeling a large class of communication protocols [20]. Some works have also tried to map such extensions to model checking [7].

Extending Event-B models to reason on probability and time has been addressed and applied to different case studies [4, 7, 20, 34]. Generally, probabilistic Event-B models have been transformed into Markov models for the purpose of verification, and timed Event-B models have been transformed into timed automata for the same purpose. However, some real-time systems need to incorporate both aspects (probability and time). To this end, we aim to model both aspects for the purpose of a formal evaluation of probabilistic timed properties; this can be achieved by transforming extended Event-B models into probabilistic time automata (PTA). As a result, these can be verified by using the PRISM probabilistic model checker. This approach has been applied on two different case studies that required the modeling of both probability and time. To our knowledge, incorporating both probability and real-time constraints into a single Event-B model has not been addressed before.

Transforming between different models and specification languages as well as incorporating different verification and specification frameworks would absolutely help to make great advancements in the software-development process. In this regard, many works have tried to transform Event-B models into formal verification frameworks, which have led to the proposition of software plugins that can be added to the Rodin platform, for instance [27, 32, 36]. Since Event-B is a constructive approach that goes through different steps of refinement (from an abstract level to a concert

one), some transformation approaches have addressed the abstract aspect of Event-B for the aim of verification [9], while others have addressed the concrete aspect [26].

This paper contains the following contributions: we propose an approach for extending Event-B models with both probability and time, which has not been addressed before in a single framework. This modeling approach allows for the smooth transformation of extended Event-B models with probability and time into probabilistic timed automata (PTA); thus, they can be easily verified against probabilistic timed properties in the PRISM probabilistic model checker. We believe that employing formal verification techniques through the software-development process with Event-B would help to verify our understanding of a system under construction and, thus, lead to more robustness through all of the steps of development. The approach has been applied in two case studies.

This paper is organized as follows. In Section 2, we introduce some related works. Some preliminaries and definitions are given in Section 3 – we introduce Markov models, and we define PTA, the PRISM language, probabilistic computation tree logic (PCTL), and probabilistic timed CTL (PTCTL) logic as well as the Event-B language. In Section 4, we introduce our approach for transforming extended Event-B models that are augmented with probability and time into the PRISM language (this transformation is guided by two case studies). Section 5 concludes the paper and presents some potential future work.

2. Related works

Many works have tried to introduce probabilistic assignments into Event-B. Tarasyuk et al. [33] studied how a probabilistic assessment of the reliability of control systems can be modeled in Event-B. The authors proposed a probabilistic choice operator that allows for an assessment of these systems' reliability. This new operator can replace nondeterministic choice statements in event actions and, thus, allow one to introduce probabilistic assignments. The refinement in this context was treated quantitatively in order to demonstrate that a refined system is at least as reliable as an abstract one. They aimed to integrate quantitative dependability assessment attributes such as reliability into their formal system development.

In an extended work [34], the author showed that an Event-B model that was augmented with stochastic information can be transformed into a continuous-time Markov chain (CTMC) through a case study of a formal modeling and verification of service-oriented systems where all events are augmented with transition rates instead of discrete probabilities. Then, this CTMC was expressed in the PRISM model checker, which enabled a quantitative evaluation of the quality of service through the probabilistic properties that are expressed in continuous stochastic logic (CSL).

Compared to previous works that only focused on probabilistic assignments, Aouadhi et al. [4] suggested expressing probabilities on the standard non-deterministic choices that appear in the Event-B language. In addition to probabilistic assignments, the choice between enabled events and event-parameter values were also considered.

Depending on the choice of the system under development, they proposed either complete probabilization (where all non-deterministic choices could be refined into probabilistic choices at the same time) or partial probabilization (where only some non-deterministic choices could turn into probabilistic choices). In contrast to the previous approaches, the aim was not to transform probabilistic Event-B modes into a Markov chain in order to verify probabilistic properties through model checking but rather to reason directly on fully probabilistic Event-B models by using the symbolic proof mechanism.

Hallerstede et al. [14] proposed a focus on extending Event-B to model probabilistic systems with the means for the qualitative modeling of probability. They stated that, in contrast to some modeling problems such as reliability and performance (which require numerical values), a class of problems like those that are found in communication protocols existed where exact numerical measures were not important. In order to target such a class of problems, they proposed refining non-deterministic assignments into qualitative probabilistic assignments.

Before addressing probabilistic information in Event-B, Hoang [18] developed the probabilistic B-Method (pB) as an extension of the B method in order to reason formally about probabilistic systems. This method included the new syntax and semantics of the probabilistic abstract machine notation (pAMN). They modified the B-Toolkit in order to support the extension from B to pB. Ndukwui et al. [29] relied on the semantics of this method and tried to investigate the automatic translation of probabilistic B machines into a PRISM model checker in order to investigate the presence of probabilistic counterexamples. They supposed that pB machines could be given as a Markov decision process (MDP) that could be expressed in PRISM (where the main aim was to define the reward-based properties).

For real-time constraints, Iliasov et al. [20] presented an approach for augmenting Event-B modeling with the verification of real-time properties, which can be achieved through extracting a process-based view from an Event-B model. Then, they introduced time constraints that allowed them to create a timed automata model; this could be used as an input for an Uppaal timed model checker [5] in order to evaluate real-time properties.

Cansell et al. [7] also tried to express the time constraints in an Event-B model by defining the concept of time in terms of set theory. Their work was motivated by a case study that was investigated by Abrial et al. [2], which concerned the IEEE 1994 tree identify protocol. They showed that a perfect modeling of such a protocol required time constraints.

Finding a common ground between B/Event-B and model checking has been also investigated in some works. Muller and Nakajima [36] extended the Rodin platform in order to allow for a behavioral analysis of the Event-B descriptions of concurrent systems by using the SPIN model-checker [19]. They built a plugin that provided interactive construction of the abstract model, then they transformed the given model into SPIN's modeling language (Promela) for safety verification. In a similar

work, Sena et al. [32] proposed a transformation approach from Event-B models into NuSMV [8]. They also provided comparative results between the NuXmv model checker [30] (an extension of NuSMV) and ProB [27].

3. Preliminaries and definitions

3.1. Markov models and Probabilistic Computation Tree Logic (PCTL)

Discrete-Time Markov Chain: a discrete-time Markov chain (DTMC) is a tuple $D = (S, s_{init}, P, L)$ such that S is a finite set of states, $s_{init} \in S$ is the initial state, and $P : S \times S \to [0, 1]$ represents a transition probability matrix where $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$. $L : S \to 2^{AP}$ is a labeling function that assigns the set $L(s)$ of the atomic propositions to each state $s \in S$.

A DTMC can be considered to be a probabilistic transition system that consists of states and the transitions between them. In DTMC, infinite path σ is a sequence of states and transitions $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2...$, where $P(s_i, s_{i+1}) > 0$ refers to the probability of a transition t_i for all $i \geq 0$. A finite path is a finite prefix of an infinite path. We define a set of paths that starts from a state s_0 by $Paths(s_0)$. The underlying σ-algebra is formed by the cylinder sets that are induced by the finite paths in $Paths(s_0)$. The probability of this *cylinder set* is as follows:

$$Pr(\{\sigma \in Paths(s_0) | s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2... \xrightarrow{t_{n-1}} s_n \text{ is a prefix of } \sigma\}) = \prod_{0 \leq i < n} P(s_i, s_{i+1}) \tag{1}$$

The probability of finite path $\sigma = s_0 s_1 ... s_n$ is defined as $\mathcal{P}(\sigma) = \prod_{0 \leq i < n} P(s_i, s_{i+1})$. The probability of the set of finite paths C is $\mathcal{P}(C) = \sum_{\sigma \in C} \mathcal{P}(\sigma)$.

Markov Decision Process: a *Markov decision process (MDP)* is a tuple $M = (S, s_{init}, Ac, P, L)$, where S is a finite set of states, $s_{init} \in S$ is the initial state, Ac is a set of actions, $P : S \times Ac \times S \to [0, 1]$ is a probability transition function such that, for each state $s \in S$ and an action $\alpha \in Ac$: $\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$, and $L : S \to 2^{AP}$ is a labeling function that assigns a subset of the finite set of atomic propositions AP to each state $s \in S$.

At each state s, the probability of moving to a successor state s' by taking an action α is given by $P(s, \alpha, s')$. We say that an action α is enabled in state s if and only if $\sum_{s' \in S} P(s, \alpha, s') = 1$; otherwise, action α is disabled. For each state $s \in S$, there is at least one action that is enabled. We denote the set of actions that is enabled from a state s as $Ac(s)$. If $|Ac(s)| = 1$ for each state s, then M can be considered to be a DTMC; thus, Ac can be omitted from the tuple.

For MDPs, computing the probabilities of paths must rely on the resolution of non-determinism (which is performed by an adversary A). An adversary resolves the non-determinism by taking one of the enabled actions $\alpha \in Ac(s)$ in each state, thus

resulting in a DTMC for which the probabilities of the paths is measurable. Since resolving the nondeterminism in an MDP results in a DTMC, the semantics of the PCTL properties over MDPs are similar to DTMCs.

Probabilistic Computation Tree Logic: probabilistic computation tree logic (PCTL) [15] has appeared as an extension of CTL for the specification of systems that exhibit stochastic behavior. We use PCTL to define the quantitative properties of DTMCs. PCTL state formulas are formed according to the following grammar:

$$\phi ::= true|a|\neg\phi|\phi_1 \wedge \phi_2|\mathbf{P}_{\sim p}(\psi) \tag{2}$$

where $a \in AP$ is an atomic proposition, ψ is a path formula, $\mathbf{P}$ is a probability threshold operator, $\sim \in \{<, \leq, >, \geq\}$ is a comparison operator, and p is a probability threshold. Path formulas ψ are formed according to the following grammar:

$$\psi ::= \phi_1 \mathbf{U} \phi_2|\phi_1 \mathbf{W} \phi_2|\phi_1 \mathbf{U}^{\leq n} \phi_2|\phi_1 \mathbf{W}^{\leq n} \phi_2 \tag{3}$$

where ϕ_1 and ϕ_2 are state formulas, and $n \in \mathbb{N}$. As in CTL, the temporal operators ($\mathbf{U}$ for strong untils, and $\mathbf{W}$ for weak [unless] untils and their bounded variants) are required to be immediately preceded by operator $\mathbf{P}$. The PCTL formula is a state formula where path formulas only occur inside operator $\mathbf{P}$. Operator $\mathbf{P}$ can be seen as a quantification operator for both the $\forall$ (universal quantification) and $\exists$ (existential quantification) operators, since the properties represent the quantitative requirements.

The semantics of a PCTL formula over a state s (or a path σ) in a DTMC model $D = (S, s_{init}, P, L)$ can be defined by a satisfaction relationship that is denoted as $\models$. The satisfaction of $\mathbf{P}_{\sim p}(\psi)$ on DTMC depends on the probability mass of a set of paths that satisfies ψ. This set is considered to be a countable union of cylinder sets so that its measurability is ensured.

The semantics of the PCTL state formulas for DTMC are defined as follows:

$s \models true \Leftrightarrow true$
$s \models a \Leftrightarrow a \in L(s)$
$s \models \neg\phi \Leftrightarrow s \not\models \phi$
$s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \wedge s \models \phi_2$
$s \models \mathbf{P}_{\sim p}(\psi) \Leftrightarrow \mathbf{P}(\{\sigma \in Paths(s) | \sigma \models \psi\}) \sim p.$

Given a path $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2...$ in D and an integer $j \geq 0$ (where $\sigma[j] = s_j$), the semantics of the PCTL path formulas for DTMC are defined as follows:

$\sigma \models \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq 0.\sigma\,[j] \models \phi_2 \wedge (\forall 0 \leq k < j.\sigma\,[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{W} \phi_2 \Leftrightarrow \sigma \models \phi_1 \mathbf{U} \phi_2 \vee (\forall k \geq 0.\sigma\,[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{U}^{\leq n} \phi_2 \Leftrightarrow \exists 0 \leq j \leq n.\sigma\,[j] \models \phi_2 \wedge (\forall 0 \leq k < j.\sigma\,[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{W}^{\leq n} \phi_2 \Leftrightarrow \sigma \models \phi_1 \mathbf{U}^{\leq n} \phi_2 \vee (\forall 0 \leq k \leq n.\sigma\,[k] \models \phi_1).$

3.2. Clocks and zones

We denote the domain of time (non-negative reals) as $\mathbb{R}_+$ and the set of natural numbers as $\mathbb{N}$. Let X be a finite set of variables called "clocks" that take values

from $\mathbb{R}_+$. We denote the clock-valuation function that assigns a value $v \in \mathbb{R}_+^X$ as $v(x)$, where $\mathbb{R}_+^X$ represents the set of all of the clock valuations of X. For any $v \in \mathbb{R}_+^X$ and $t \in \mathbb{R}_+$, $v + t$ denotes the clock valuation that is defined as $(v + t)(x) = v(x) + t$ for all $x \in X$.

Constraints: A constraint over X is an expression of the form $x_i \sim c$ or $x_i - x_j \sim c$, where $x \in X$, $1 \leq i \neq j \leq n$, $\sim \in \{<, \leq\}$, and $c \in \mathbb{N}$.

A clock valuation v satisfies a constraint $x_i - x_j \sim c$ iff $v(x_i) - v(x_j) \sim c$.

Zones: A zone of X (written as ζ) is a convex subset of the valuation space $\mathbb{R}_+^X$ described by a conjunction of the constraints. A ζ zone represents the set of valuations that satisfy the conjunction of the $n.(n+1)$ constraints, which is given by the following:

$$\bigwedge_{1 \leq i \neq j \leq n} x_i - x_j \sim i, jc_{i,j} \tag{4}$$

The set of zones (clock constraints) of X (denoted as $Z(X)$) is defined by the following syntax:

$$\zeta ::= x \leq d \mid c \leq x \mid x + c \leq y + d \mid \neg\zeta \mid \zeta \wedge \zeta \tag{5}$$

where $x, y \in X$, and $c, d \in \mathbb{N}$. We say that a clock valuation v satisfies a zone ζ (denoted as $v \triangleright \zeta$) if and only if ζ resolves to true after substituting each clock x with $v(x)$. Other constraints can be easily derived; for example, $\zeta_1 \wedge \zeta_2 = \neg(\neg\zeta_1 \vee \neg\zeta_2)$, $x > 1 \equiv \neg(x \leq 2)$, and equality can be written as a conjunction of constraints (for example, $x = 2 \equiv (x \geq 2 \wedge x < 3)$).

For the $\zeta, \zeta' \in Z(X)$ zones and subset of clocks $\chi \subseteq X$, we obtain the classical operations on the zones [16, 24] as follows:

$$\swarrow \zeta'\zeta \stackrel{def}{=} \{v | \exists t \geq 0.(v + t) \triangleright \zeta \wedge \forall t' \leq t.(v + t' \triangleright \zeta \vee \zeta')\}$$
$$[\chi := 0]\zeta \stackrel{def}{=} \{v | v[\chi := 0] \triangleright \zeta\}$$
$$\zeta[\chi := 0] \stackrel{def}{=} \{v[\chi := 0] | v \triangleright \zeta\}$$

3.3. Probabilistic Timed Automata (PTA)

While the formalism of clocks and zones is the same for classical timed automata [3], PTAs are extended with discrete probability distributions over the edges.

Probabilistic Timed Automata: a probabilistic timed automaton (PTA) is a tuple $(Loc, l_0, X, inv, A, prob, L)$ where: Loc is a finite set of locations with l_0 as the initial location. X is a finite set of clocks. $inv : Loc \longrightarrow Z(X)$ maps an invariant condition to each location. A is a finite set of actions, $prob \subseteq Loc \times Z(X) \times Dist(Loc \times 2^X)$ is the probabilistic edge relationship, and $L : Loc \longrightarrow AP$ is a labeling function.

A state of PTA is a pair $(l, v) \in L \times \mathbb{R}^X$ such that $v \triangleright inv(l)$. An edge of PTA is (l, g, a, p, l', Y), where l' is the destination location, Y is the set of clocks to be reset, (l, g, a, p) is a probabilistic edge of PTA where l is source location, g is a guard, a is an action, and p is the destination distribution. In l_0, all of the clocks are initialized to zero.

For any state (l, v), there is a non-deterministic choice between making a discrete transition and letting the time pass; the transition is enabled if $v \triangleright g$ and the probability of moving to destination location l' that results in resetting the Y set of clocks equals $p(l', Y)$. Letting the time pass in current location l is provided by invariant condition $inv(l)$, which is continuously satisfied as time elapses. Actually, a PTA can be considered to be an infinite-state MDP with the expression of time delays.

3.4. Probabilistic Timed CTL (PTCTL)

Probabilistic timed computation tree logic (PTCTL) [23] has appeared as an extension of CTL for the specification of probabilistic timed systems. This is derived from PCTL [15] (which is used to specify the properties of DTMCs and MDPs) and TCTL [16] (which is used for the specification of timed automata). We use PTCTL for defining the quantitative and timing properties of PTAs. Like TCTL, we use a set of clock variables for expressing the timing properties; this set is denoted as Z disjoint from X, where $\xi : Z \to \mathbb{R}$ is a formula clock valuation that assigns values to such clocks. PTCTL state formulas are formed according to the following grammar:

$$\phi ::= true|a|\zeta|z.\phi|\phi_1 \wedge \phi_2|\neg\phi|\mathbf{P}_{\sim p}(\varphi) \tag{6}$$

where $a \in AP$ is an atomic proposition, ζ is a zone over $X \cup Z$, $z.\phi$ is reset quantifier, φ is a path formula, $\mathbf{P}$ is a probability threshold operator, $\sim \in \{<, \leq, >, \geq\}$ is a comparison operator, and p is a probability threshold. The φ path formulas are formed according to the following grammar:

$$\varphi ::= \phi_1 \mathbf{U} \phi_2|\phi_1 \mathbf{U}^{\leq n} \phi_2 \tag{7}$$

where ϕ_1 and ϕ_2 are state formulas, and $n \in \mathbb{N}$. The temporal until operator $\mathbf{U}$ and its bounded variant are required to be immediately preceded by operator $\mathbf{P}$. The PTCTL formula is a state formula where path formulas only occur inside operator $\mathbf{P}$. Operator $\mathbf{P}$ can be seen as a quantification operator for both of the $\forall$ (universal quantification) and $\exists$ (existential quantification) operators, since the properties represent the quantitative requirements.

Before introducing the semantics of the PTCTL formula over PTA, we must first define the divergent adversaries. A divergent adversary must fulfill the following condition: for each state $s = (l, v)$, the probability of divergent paths under A is 1; i.e., $Pr_s^A\{\omega \in Paths^A(s)|\omega \text{ is divergant}\} = 1$. A path ω is divergent if, for any $t \in \mathbb{R}_+$, there exists $j \in \mathbb{N}$ such that $D_\omega(j) > t$, where $D_\omega(n)$ denotes the duration up to a state s_n.

Let (l, v) be a state and ξ be a formula clock valuation. We can now express the full semantics of PTCTL over PTA:

$(l, v), \xi \models a \Leftrightarrow a \in L(l, v)$
$(l, v), \xi \models \zeta \Leftrightarrow v, \xi \triangleright \zeta$
$(l, v), \xi \models z.\phi \Leftrightarrow (l, v), \xi[z := 0] \models \phi$

$(l,v),\xi \models \phi_1 \wedge \phi_2 \Leftrightarrow (l,v),\xi \models \phi_1$ and $(l,v),\xi \models \phi_2$
$(l,v),\xi \models \neg\phi \Leftrightarrow (l,v),\xi \models \phi$ is flase
$(l,v),\xi \models \mathbf{P}_{\sim p}(\varphi) \Leftrightarrow Pr^{A}_{(l,v)}\{\ \omega \in Paths^{A}(l,v)|\omega,\xi \models \varphi\} \sim p$ for all adversaries

With PTCTL, we can express properties such as, with a probability of at least 0.98, the packet is eventually delivered within five time units, which is expressed by using PTCTL as follows: $\mathbf{P}_{\geq 0.98}[(true\mathbf{U}PacketDelivered \wedge (z = 5)]$.

3.5. PRISM language

A model in PRISM consists of one module or several modules that interact with each other. The modules are specified using the PRISM language as a set of guarded commands. A module consists of variables that express the local state of this module. The behavior of a module is given by a set of guarded commands of the following form:

$[< action >] < guard > \longrightarrow < updates >$.

$< action >$ can be used as a synchronous action between different modules. In the case of no synchronization, label $< action >$ should not be named. $< guard >$ is a predicate over the variables of the system, and *updates* describe the probabilistic transitions that the module can make if the guard is true. These updates represent the new values of the variables of this module and are defined as follows:

$< prob >:< atomicupdate > +.....+ < prob >:< atomicupdate >$.

The state of the entire model is determined by the local states of all of the modules. Updates of two or more modules can be performed together through action synchronization. In addition to local variables, we have global variables that can be used along all of the modules for specifying guards, for instance; we can also use them for defining formulas, or they can be used in specifying properties. When representing CTMCs, $< prob >$ will refer to transition rates instead of discrete probabilities.

Formally, a PRISM program is defined as a tuple (V, M, C, A), where $V = V_{local} \cup V_{global}$ is a finite set of local and global variables, $M = (M_1, M_2, ...M_n)$ is a finite set of modules, C is a set of commands, and $A = A_{syn} \cup A_{asyn}$ is a finite set of synchronizing and non-synchronizing actions (where synchronizing actions help obtain the parallel composition of modules $[M1||M2...]$).

A PTA in PRISM is represented by one module or a parallel composition of different modules that can be synchronized. A PTA is considered to be a probabilistic automaton with real-valued clocks or a timed automaton with discrete probabilistic choices. For the cases of discrete transitions, these are enabled based on the value of clock x.

Figure 1 presents an example of PTA, and Figure 2 presents its description in the PRISM language. As we can see, we have a set of states where some states must satisfy a set of clock invariants such as $s = 0$ and $s = 1$. When an action's guard is satisfied, it enables a transition that could include only a clock variable like action *retransmit*

($x \geq 3$), or it can also include other variables like action *send* with a guard on variable *tries*. At every state, we can have nondeterministic and probabilistic choices.

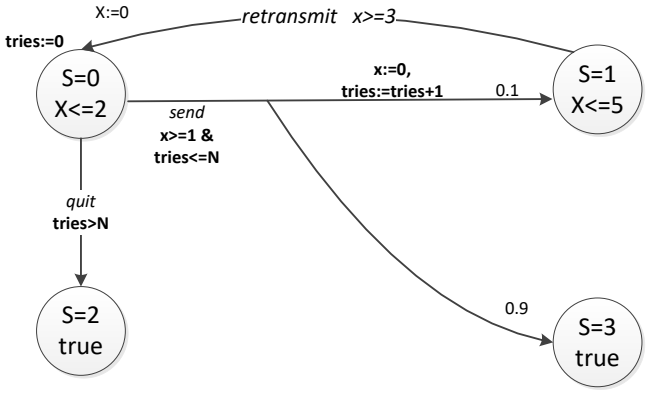

Figure 1. PTA

```
pta
const int N;

module transmitter
s : [0..3] init 0;
tries : [0..N+1] init 0;
x : clock;

invariant
(s = 0 ⇒ x ≤ 2)&(s = 1 ⇒ x ≤ 5)
&(s = 2 ⇒ true)&(s = 3 ⇒ true)
endinvariant

[send] s=0 & tries≤ N & x≥ 1 -->0.9: (s'=3) +
0.1:(s'=1)&(tries'=tries+1) & (x'=0);
[retransmmit] s=1 & x ≥ 3 --> (s'=0) & (x'=0);
[quit] s=0 & tries > N --> (s' =2);
endmodule
```

Figure 2. PTA in PRISM language

Its corresponding model in PRISM defines three variables: s, which denotes the possible values of the states starting in the initial state with $s = 0$, natural variable *tries*, which could have a maximum value of $N + 1$, and clock x. Then, we define the possible invariants on the states. Finally, we define the guarded commands that describe the behavior of the system.

3.6. Event-B

Event-B is a formal method that we use for system modeling and analysis [1]. This is based on the refinement principle, which allows systems to be modeled and analyzed through different levels of abstraction, and it uses mathematical proofs as a means of consistency verification between refinement levels.

An Event-B model is specified by the notion of an abstract state machine. An abstract state machine defines the states of the model based on a collection of variables and defines the operations on these states, which gives rise to the behavior of the system based on a set of events. In Event-B, the variables are strongly typed by introducing invariants. These invariants actually represent the important properties that should be preserved during system execution.

Now, concerning the behavior of the system, it is defined by a set of events of the following form:

evnt = **where** G **then** A end,

where G refers to the guard, and A refers to the set of actions that can be executed when G holds. The actions determine how the machine's variables evolve through symbol (:=). Each action of an event could introduce deterministic as well as non-deterministic assignments. Roughly speaking, a machine in Event-B can be seen as a transition system where the values of the variables encode the states and the events enable the transitions.

Formally, an Event-B model is a tuple $(S, V, C, I, \sigma, E, Init, A)$, where S defines the sets, V is a set of variables, C is a set of constants, I is a set of invariant properties over S, V, and C, σ is a state space that is defined by all of the possible values of V, E is a non-empty set of events that includes initialization event $Init$ (which denotes the initial state), and A defines the set of axioms that is used to express the typing invariants, for instance.

4. Mapping Event-B to PRISM

In this section, we will show how extended Event-B models with probability and time can be transformed into PTA, which is described in the PRISM language of the PRISM probabilistic model checker. Conventional model checking has been successfully applied to Event-B models thanks to the ProB model checker [27], which allows for the automatic animation of many B specifications and can be used to check a specification given in linear temporal logic (LTL); it can even generate counterexamples when the specification fails. However, probabilistic timed model checking for Event-B models has never been addressed before, since Event-B models lack the notions of probability and time. Due to the importance of these notions, however, many works have tried to incorporate probability and time in Event-B modeling (but in a separate way, unfortunately).

In this section, we will show how we can incorporate both probability and time into Event-B models by enabling a smooth mapping to PTA that is expressed in the

PRISM language. We should recall that PTA deals with nondeterminism, probability, and time clocks; therefore, we must deal with these main issues in addition to other existing elements such as variables, events, and invariants. The transformation from Event-B models to PRISM will be guided by two case studies: the IEEE 1394 root contention protocol (called FireWire [10]), and the CSMA/CD (carrier-sense multiple access with collision detection) protocol [12, 25].

FireWire root contention protocol is an election protocol that can be found in different networks. After a bus reset, two or more contending nodes try to elect a leader that will act as a manager. All nodes are considered equals; thus, all nodes communicate through the message, "be my parent." It might happen that two nodes contend the leadership; therefore, multiple rounds of elections are anticipated. So, a node is supposed to flip a coin; depending on the result, a node may either decide to wait for a short amount of time ("fast") or a long time ("slow"). Then, the node checks whether another node has deferred; if so, then it declares itself to be the leader. If it happens that both of the coin flips of the two nodes have the same results, then another round of the protocol is still needed. The PTA that corresponds to the abstract representation of the protocol is depicted in Figure 3.

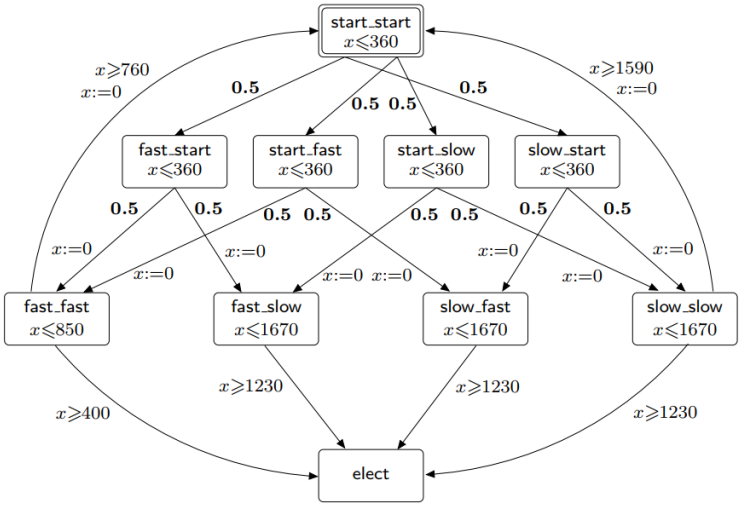

Figure 3. PTA of FireWire [25]

While the main building block in Event-B modeling is an abstract machine (i.e., a system is described by a set of abstract machines), the main building block in PRISM is a module (i.e., a system is described by a set of modules). Therefore, it is evident that we should map each abstract machine in Event-B into a PRISM module. In the following section, we will show the rest of the transformation conditions that should be met.

States, clocks, and invariants

We should recall that a state in PTA is determined by pair (l, v), where l is a location, and v is an invariant that must be satisfied in l.

For states in Event-B, we use *Sets* with invariants to encode the states. We will show that this representation in Event-B is very similar to the representation in PRISM. Before doing so, we also need to express clocks variables in Event-B, since they are needed in order to express the state invariants. In contrast to existing works that have tried to incorporate time into Event-B models through proposing a complex timing pattern [7], we can use a simple notation here by merely introducing clocks as real variables. To discriminate clock variables from other variables, we use a keyword *clock* that is added to the variable (i.e., *clock_x*), which refers to a clock x. For each clock, we define an invariant that states its type (which must be defined in $\mathbb{R}_+$).

By introducing clocks, we can now define invariants on those locations that form the possible states of the system. The mapping between the PRISM model and the Event-B of the FireWire protocol is introduced in Figures 4, 5, 6, and 7, respectively.

```
MACHINE
FireWire

SEES
global_Context_FireWire

VARIABLES
clock_x
s ∈ s_STATES

SETS
s_STATES = {start_start, fast_start, start_fast,
    start_slow, slow_start, fast_fast, fast_slow,
    slow_fast, slow_slow, done}
INVARIANTS

clock_x ∈ ℝ+

(s=start_start)⇒ (clock_x<=delay)

(s=fast_start)⇒(clock_x<=delay)
(s=start_fast)⇒ (clock_x<=delay)
(s=start_slow)⇒ (clock_x<=delay)
(s=slow_start)⇒ (clock_x<=delay)

(s=fast_fast)⇒ (clock_x<=rc_fast_max)
(s=fast_slow)⇒ (clock_x<=rc_slow_max)
(s=slow_slow)⇒(clock_x<=rc_slow_max)
(s=slow_fast)⇒ (clock_x<=rc_slow_max)
```

Figure 4. Event-B model of FireWire – variables and invariants

CONTEXT
global_Context_FireWire

CONSTANTS
delay
rc_fast_max
rc_slow_max
rc_fast_min
rc_slow_min
fast
slow
AXIOMS
axm1: $delay \in \mathbb{N}$
axm2: $rc\_fast\_max \in \mathbb{N}$
axm3: $rc\_slow\_max \in \mathbb{N}$
axm4: $rc\_fast\_min \in \mathbb{N}$
axm5: $rc\_slow\_min \in \mathbb{N}$
axm6: $fast \in \mathbb{R}_+$
axm7: $slow \in \mathbb{R}_+$
axm8: $slow = 1 - fast$
axm9: $fast = 0..1$

Figure 5. Event-B model of FireWire – context

EVENT
INITIALIZATION
s := start_start
NODE_A_SEND:
WHERE
isIN_start_start : s = start_start
THEN
Enter_Node_A_flips_coin: s ⊕ | fast_start @fast;
slow_start @slow
END
NODE_A_SEND2_FROM_FAST:
WHERE
isIN_fast_start : s = fast_start
THEN
Enter_Node_A_flips_coin: s ⊕| fast_fast @fast; fast_slow @slow
reset clock: clock_x:=0
END
NODE_A_SEND2_FROMSLOW:
WHERE
isIN_slow_start : s = slow_start
THEN
Enter_Node_A_flips_coin: s ⊕| slow_fast @fast; slow_slow @slow
reset clock: clock_x:=0
END
...

Figure 6. Event-B model of FireWire – actions

```
...
Node_B_Send_2_FromFast:
WHERE
isIN_fast_start : s = start_fast
THEN
Enter_Node_B_flips_coin: s ⊕| fast_fast @fast; slow_fast
    @slow
reset clock: clock_x:=0
END
Elect_A: // send after fast state
WHERE
isIn_slow_fast : s = slow_fast
grdActionClock: clock_x>= (rc_slow_min − delay)
THEN
elect_done : s := done
reset clock: clock_x:=0
END
Elect2_A:
WHERE
isInfast_fast : s = fast_fast
grdActionClock:clock_x>= (rc_fast_min − delay)
THEN
elect_done : s := done
reset clock: clock_x:=0
END
...
```

Figure 6. cont.

In the Event-B model (See Figure 4), *s_STATES* includes the possible states where we express them in PRISM by local state variable *s* (See Figure 7). Then, the invariants are introduced in PRISM in the same manner as they are in Event-B. *delay*, *rc_fast_max*, and *rc_slow_max*, etc. are all constants that are defined as constants in Event-B. In PRISM, they are defined globally in a similar way by using the **const** keyword (See Figure 7).

```
pta
const int rc_fast_max = 850;
const int rc_fast_min = 760;
const int rc_slow_max = 1670;
const int rc_slow_min = 1590;
// delay caused by the wire length
const int delay = 360;
// probability of fast and slow
const double fast = 0.5;
const double slow = 1−fast;
module abstract_firewire
x : clock;
s : [0..9];
```

Figure 7. PRISM model of FireWire [31]

```
// 0 - start_start
// 1 - fast_start
//...
// 9 - done
// clock invariant
invariant
(s=0 => x<=delay) &
(s=1 => x<=delay) &
(s=2 => x<=delay) &
(s=3 => x<=delay) &
(s=4 => x<=delay) &
(s=5 => x<=rc_fast_max) &
(s=6 => x<=rc_slow_max) &
(s=7 => x<=rc_slow_max) &
(s=8 => x<=rc_slow_max)
endinvariant
// start_start (initial state)
[] s=0 -> fast : (s'=1) + slow : (s'=4);
[] s=0 -> fast : (s'=2) + slow : (s'=3);
// fast_start
[] s=1 -> fast : (s'=5) & (x'=0) + slow : (s'=6) & (x
    '=0);
// start_fast
[] s=2 -> fast : (s'=5) & (x'=0) + slow : (s'=7) & (x
    '=0);
// start_slow
[] s=3 -> fast : (s'=6) & (x'=0) + slow : (s'=8) & (x
    '=0);
// slow_start
[] s=4 -> fast : (s'=7) & (x'=0) + slow : (s'=8) & (x
    '=0);
// fast_fast
[] s=5 & (x>=rc_fast_min) -> (s'=0) & (x'=0);
[] s=5 & x>= (rc_fast_min - delay) -> (s'=9) & (x'=0);
// fast_slow
[] s=6 & x>= (rc_slow_min - delay) -> (s'=9) & (x'=0);
// slow_fast
[] s=7 & x>= (rc_slow_min - delay) -> (s'=9) & (x'=0);
// slow_slow
[] s=8 & x>=rc_slow_min -> (s'=0) & (x'=0);
[] s=8 & x>= (rc_slow_min - delay) -> (s'=9) & (x'=0);
// done
[] s=9 -> true;
endmodule
```

Figure 7. cont.

Constants, local variables, and global variables

In addition to the main variables that denote the states and clocks, we may need additional variables (e.g., counters, probabilities values, and min/max values) to express the ranges, for instance.

In PRISM, we have two types of variables: global, and local. Local variables such as state and clock variables belong to only one module; thus, they can only be modified by this module. Global variables are declared to be outside all of the modules that use the keyword **global** at the beginning of the model; thus, they can be modified by all modules. In addition to global and local variables, we have constants; these are declared in PRISM by using the keyword **const**. In PRISM, constants are always declared globally.

From what has preceded, it is important to consider all of these issues for transforming Event-B models into PRISM. To do so, we propose the following:

- global variables and constants must be defined in global context that can be seen by all abstract machines;
- local variables such as states and clocks are declared locally at level of each abstract machine.

For the FireWire protocol, the global context that is seen by the *FireWire* abstract machine is introduced in Figure 5. It introduces the constants that the abstract machine needs to see and use.

Events

Events in PRISM also have a representation that is similar to Event-B. A set of updates or actions is executed once a guard is true. In addition, both PTA and Event-B models could introduce nondeterministic actions; however, Event-B models do not provide probabilistic actions. Therefore, we need to extend Event-B models to reason about the probabilities that are assigned to each action when a guard holds (just as we do in PRISM for describing PTA). To do so, we adopt a simple and yet efficient notation that was introduced in [33]. We use the $\oplus|$ operator to denote a probabilistic choice between two actions: one with probability $@p$, and the other with probability $@(1-p)$. For the FireWire protocol (See Figure 6), *slow* and $fast$ refer to probabilities p and $1-p$, respectively. Figure 6 shows the definition of the events of Nodes A and B that correspond to the guarded commands in PRISM (where we can observe the increased similarities between them). Due to the length of the entire model, only a subset of the events has been introduced in Figure 6.

Decomposition and parallel composition through synchronization

While we map most of the important elements of Event-B to PRISM, we still need to consider one more issue regarding PRISM, which is the parallel composition of different components through action synchronization.

To deal with the parallel composition of different modules through action synchronization in Event-B, we employ the decomposition technique from [6]. Butler proposed a decomposition technique that allows for the partitioning of a single machine into several sub-machines; this has its origins in the synchronous parallel composition of processes that are found in process algebra. Using process algebra operator $||$, the parallel composition of two machines A and B (denoted $A||B$) can be achieved.

The A and B machines can then interact via synchronizing shared events; i.e., those events that share the same names in both machines. Actually, this representation is the same that is adopted in PRISM to allow for the parallel composition of the modules. Therefore, we adopt this representation in Event-B to allow for a smooth transformation from Event-B to PRISM.

While the previous case study of FireWire consisted of one component, we will now address a case study that consists of different machines or modules. This case study concerns the CSMA/CD (carrier-sense multiple access with collision detection) protocol [12, 25]. Through this case study, we will show how we can deal with the issue of synchronization.

CSMA/CD is a protocol for carrier transmission access in Ethernet networks that avoids collisions (minimizing the simultaneous use of a channel) when a network interface card (NIC) tries to send its packet. The protocol is modeled as a PTA in PRISM and consists of the following main components or modules: at least two senders (namely, Station 1 and Station 2), and the bus (or the medium). The protocol functionality is as follows: if a station has data to send, it first listens to the medium: if the medium is free, the station sends the data; otherwise (the bus is busy), it repeats the process after a random amount of time. If there is a collision, the station attempts to re-transmit the packet where the scheduling of the re-transmission is determined by a truncated binary exponential backoff process. The description of the PTA that is related to CSMA/CD can be found in [25], and the complete model in the PRISM language is available in the PRISM benchmark suite [31].

a)

CONTEXT
global_Context_CSMA

CONSTANTS
k
slot
sigma
lambda
AXIOMS
axm1: $k \in \mathbb{N}$
axm2: $slot \in \mathbb{N}$
axm3: $sigma \in \mathbb{N}$
axm4: $lambda \in \mathbb{N}$
axm5: $slot = 2 * sigma$
axm6: $p \in \mathbb{R}_+$
axm7: $p = 0..1$

b)

MACHINE
Station1
SEES
global_Context_CSMA
VARIABLES
clock_x1
cd
s1 ∈ stat_STATES
SETS
stat_STATES = {initial, transmit, collide, wait, done}
INVARIANTS
clock_x1 ∈ $\mathbb{R}_+$
cd ∈ 1..k
(s1=initial) ⟶ (x=0)
(s1=transmit) ⟶ (x ≤ *lambda*)
(s1=collide) ⟶ (x=0)
(s1=wait) ⟶ (clock_x1 ≤ pow(2, cd1)∗slot)

Figure 8. Context, variables, and invariants of CSMA: a) station1 machine of CSMA – variables and invariants; b) event-B model of CSMA – context

As shown before, each abstract machine will refer to a module in PRISM in our mapping; therefore, we will define different sets of states in Event-B at the level of

each abstract machine in addition to the clock variables. For CSMA/CD, we have two main components: the station, and the bus. As we see in Figures 8b and 9, we declare two sets (named *stat_states* and *bus_states*, respectively) at each abstract machine. In addition, we define the clock variable for each machine locally as well as their corresponding invariants. Similar to the FireWire protocol, the global constants that must be seen by both machines are defined in one shared context (See Figure 8a).

```
MACHINE
Bus
SEES
global_Context_CSMA
VARIABLES
clock_y
b ∈ bus_STATES
SETS
bus_STATES = {initial, transmit, collide}
INVARIANTS
clock_y ∈ ℝ+
(b=collide) ⟶ (y ≤ sigma)
```

Figure 9. Bus machine of CSMA – variables and invariants

For the events, we can see in Figures 10 and 11 that synchronizing shared events have the same names (*Bus_S1_Wait*, *Bus_S1_cd*, etc.). However, a machine could still define non-shared events such as *Station_Retransmit* in *Station 1*.

```
EVENT
INITIALIZATION:
s1 := initial  cd:=0
Bus_Send_S1:
WHERE
isIn_Initial : s1=initial
THEN
Enter_Transmit: s1:=transmit
END
Send_S1_AfterTransmit:
WHERE
isIn_Wait_S: s1=wait
guard_clock_S: clock_x1=pow(2, cd1)*slot))
THEN
Enter_Transmit_S: s1:=transmit
Reset_clock_S:cx1:=0
END
Bus_S1_Initial:
WHERE
isInInitial_S    : s1=initial
THEN
Enter_Collide_S: s1:=collide
Reset_clock_S: clock_x1:=0
count_cd: cd1:=min(K,cd1+1)
END
```

Figure 10. Events of Station 1 – CSMA

```
Bus_S1_Wait:
WHERE
isIn_Wait_S: s1=wait
guard_clock_S: clock_x1=pow(2, cd1)*slot)
THEN
Enter_Collide_S: s1:=collide
Reset_clock_S: clock_x1:=0
count_cd: cd1:=min(K,cd1+1)
END
Bus_S1_cd:
WHERE
isIn_Transmit_S: s1=transmit
THEN
Enter_Collide_S: s1:=collide
Reset_clock_S: clock_x1:=0
count_cd: cd1:=min(K,cd1+1)
END
Bus_S1_end1:
WHERE
isIn_Transmit_S: s1=transmit
guard_clock_S: clock_x1=lambda
THEN
success_sent: s1:=done
Reset_clock_S: clock_x1:=0
END
Station_Retransmit:
WHERE
isIn_Collide_S : s1=transmit
coll_detected: cd1=1
THEN
station1 flips coin: s ⊕| s1=3 ∧ (clock_x1=0*slot) @p; s1=3 ∧ (
    clock_x1=1*slot) @1−p
END
```

Figure 10. cont.

```
EVENT
INITIALIZATION
b:= initial
Bus_Send_S1:
WHERE
isIn_Initial : b=initial
THEN
Enter_Transmit: b1:= transmit
END
Send_S1_AfterTransmit:
WHERE
isIn_Transmit_M: b=transmit
guard_clock_M: clock_y<sigma
THEN
Enter_Collide_M: b:=collide
Reset_clock_M: clock_y:=0
END
Bus_S1_Initial:
WHERE
isIn_Transmit_M : b=transmit
guard_clock_M : clock_y>=sigma
THEN
skip_M: b:=transmit
END
```

Figure 11. Events of bus – CSMA

```
Bus_S1_Wait:
WHERE
isIn_Transmit_M : b=transmit
THEN
skip_M:b=1
END
Bus_S1_cd:
WHERE
isIn_Collide_M : b=2
guard_clock_M: clock_y<=sigma
THEN
Enter_Transmit_M: b:=transmit
END
Bus_S1_end1:
WHERE
isIn_Transmit_M: b=transmit
THEN
isIn_Initial_M: b:=initial
Reset_clock_M: clock_y:=0
END
```

Figure 11. cont.

Since all stations share the same behavior, the description of one station is sufficient. It is worth noting that, when we have multiple stations (which is the real case), we can deal with this issue in PRISM through renaming; this allows us to duplicate the modules.

As we see from the both case studies, our mapping approach of Event-B models into PTA in PRISM can work for both cases regardless of whether the system consists of a single component or different components. Through these two case studies, we can also see that Event-B and PRISM already share various semantics (especially with the expression of events) for both non-synchronizing and synchronizing events. Such a similarity would allow for a smooth automatic translation from Event-B to PRISM.

Property analysis

After transforming the Event-B models into PRISM models, we can use the PRISM model checker to verify and analyze different probabilistic reachability and time-bounded probabilistic reachability properties. These properties can be found in the PRISM benchmark suite [31]. Since PTA is a non-deterministic model, we use Pmin and Pmax in PRISM to indicate the minimum and maximum probabilities, respectively.

For the FireWire model, we choose to compute the results of the following property:

$Pmin =?[F(\text{“}done\text{”})]$.

This probabilistic property tries to compute the minimum probability that, from the initial state, a leader is elected before the time bound *deadline* is reached, where “done” is a label that denotes $s = 9$, and *deadline* is a variable that is added to express this property. The results for different deadline values are depicted in Table 1.

We can also reason about simple liveness properties. For instance, property $P >= 1[F(\text{“}done\text{”})]$ tries to check whether a node will eventually be made a leader (which is satisfied).

Table 1
Probabilities of leader election – FireWire

Deadline	Probability
2000	0.500
3000	0.625
4000	0.781
5000	0.851
6000	0.931
7000	0.962

For the second model of CSMA, we can state the following property:
$Pmin =?[F <= T(\text{“}all\_delivered\text{”})]$.

Given a bound T, this bounded property tries to compute the minimum probability that both stations will eventually send their packets correctly. PRISM has a simulation framework that enables us to visualize the results of any property. The simulation results of this property given different values of T as generated by PRISM are depicted in Figure 12a.

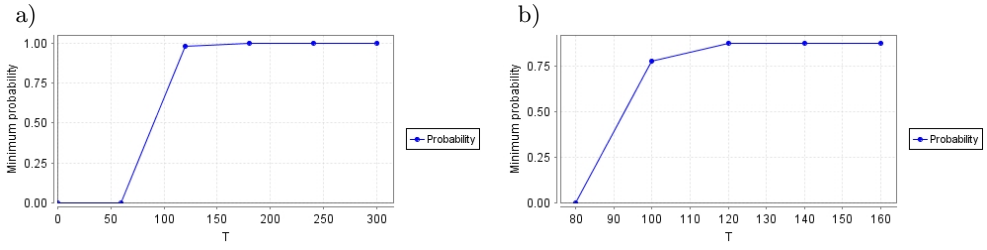

Figure 12. Analysis results: a) CSMA – minimum probability of reaching done state given T;
b) CSMA – minimum probability of reaching done state given T before collision with max backoff

We can also introduce another property:
$Pmin =?[!\text{“}collision\_max\_backoff\text{”}U <= T\text{“}all\_delivered\text{”}]$.

Given a bound T, this property aims to compute the minimum probability that both stations successfully send their packets correctly before a collision with max backoff is detected; in other words, not reaching a collision with max backoff until the “done” state is reached, where collision is identified by the event *cd*, and the maximum number of collisions can be introduced. The simulation results for this property for a maximum backoff limit of 2 and the given different values of T are depicted in Figure 12b.

5. Conclusion

In this paper, we introduced an approach for representing both probability and time in Event-B models. Then, we presented an approach for mapping Event-B models into PTA, which can be expressed in the PRISM language for verifying and analyzing probabilistic timed properties. The transformation considers basic elements of Event-B models such as variables and events (in addition to probability and time). While time is expressed simply through a clock variable, probability is expressed through an operator that introduces the probabilistic choice between different actions. We benefit from the similarities between Event-B and the PRISM language in order to deliver a smooth transformation; this would be very helpful for combining Event-B and model checking, thus facilitating the development of probabilistic real-time systems. The transformation technique presented here is designed in such a way that it can facilitate the automatic translation of Event-B models that are augmented with probability and time into PRISM.

References

[1] Abrial J.R.: *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, 2010.

[2] Abrial J.R., Cansell D., Méry D.: A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol, *Formal Aspects of Computing*, vol. 14, pp. 215–227, 2003.

[3] Alur R.: Timed Automata. In: N. Halbwachs, D. Peled (eds.), *Computer Aided Verification*, pp. 8–22, 1999.

[4] Aouadhi M., Delahaye B., Lanoix A.: Introducing probabilistic reasoning within Event-B, *Software & Systems Modeling*, vol. 18, pp. 1953–1984, 2019.

[5] Bengtsson J., Larsen K., Larsson F., Pettersson P., Yi W.: UPPAAL – a tool suite for automatic verification of real-time systems. In: R. Alur, T.A. Henzinger, E.D. Sontag (eds.), *Hybrid Systems III. Verification and Control*, Lecture Notes in Computer Science, vol. 1066, pp. 232–243, Springer, Berlin–Heidelberg, 1996. doi: 10.1007/BFb0020949.

[6] Butler M.: Decomposition Structures for Event-B. In: *Integrated Formal Methods*, pp. 20–38, Springer, Berlin–Heidelberg, 2009.

[7] Cansell D., Méry D., Rehm J.: Time Constraint Patterns for Event B Development. In: *B 2007: Formal Specification and Development in B*, pp. 140–154, Springer, Berlin–Heidelberg, 2006. doi: 10.1007/11955757_13.

[8] Cimatti A., Clarke E., Giunchiglia F., Roveri M.: NUSMV: a new symbolic model checker, *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 410–425, 2000.

[9] Dalvandi M., Butler M., Rezazadeh A.: From Event-B Models to Dafny Code Contracts. In: *Fundamentals of Software Engineering. FSEN 2015*, p. 308–315, Springer-Verlag.

[10] Daws C., Kwiatkowska M., Norman G.: Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM, *International Journal on Software Tools for Technology Transfer volume*, vol. 5, pp. 221–236, 2004. doi: 10.1007/s10009-003-0118-5.

[11] Dijkstra E.W.: *A Discipline of Programming*, Prentice Hall International, Englewood Cliffs, N.J., 1976.

[12] Duflot M., Fribourg L., Herault T., Lassaigne R., Magniette F., Messika S., Peyronnet S., Picaronny C.: Probabilistic Model Checking of the CSMA/CD protocol using PRISM and APMC, *Electronic Notes in Theoretical Computer Science*, vol. 128(6), pp. 195–214, 2004. doi: 10.1016/j.entcs.2005.04.012.

[13] Hadad A.S.A., Ma C., Ahmed A.A.O.: Formal Verification of AADL Models by Event-B, *IEEE Access*, vol. 8, pp. 72814–72834, 2020. doi: 10.1109/ACCESS.2020.2987972.

[14] Hallerstede S., Hoang T.S.: Qualitative Probabilistic Modelling in Event-B. In: J. Davies, J. Gibbons (eds.), *Integrated Formal Methods. IFM 2007*, Lecture Notes in Computer Science, vol. 4591, pp. 293–312, Springer, Berlin–Heidelberg, 2007. doi: 10.1007/978-3-540-73210-5_16.

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

[16] Henzinger T.A., Nicollin X., Sifakis J., Yovine S.: Symbolic Model Checking for Real-Time Systems, *Information and Computation*, vol. 111(2), pp. 193–244, 1994. doi: 10.1006/inco.1994.1045.

[17] Hinton A., Kwiatkowska M., Norman G., Parker D.: PRISM: A tool for automatic verification of probabilistic systems. In: H. Hermanns, J. Palsberg (eds.), *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2006*, Lecture Notes in Computer Science, vol. 3920, pp. 441–444, Springer, Berlin–Heidelberg, 2006. doi: 10.1007/11691372_29.

[18] Hoang T.S.: *The Development of a Probabilistic B-Method and a Supporting Toolkit*, Ph.D. thesis, School of Computer Science and Engineering, UNSW, 2005.

[19] Holzmann G.: *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley, 2004.

[20] Iliasov A., Romanovsky A., Laibinis L., Troubitsyna E., Latvala T.: Augmenting Event-B modelling with real-time verification. In: *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, p. 51–57, 2012. doi: 10.1109/FormSERA.2012.6229789.

[21] Katoen J.P., Khattri M., Zapreev I.S.: A Markov reward model checker. In: *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pp. 243–244, 2005. doi: 10.1109/QEST.2005.2.

[22] Kozen D.: A probabilistic PDL, *Journal of Computer and System Sciences*, vol. 30(2), pp. 162–178, 1985.

[23] Kwiatkowska M., Norman G., Segala R., Sproston J.: Automatic verification of real-time systems with discrete probability distributions, *Theoretical Computer Science*, vol. 286(1), pp. 101–150, 2002.

[24] Kwiatkowska M., Norman G., Sproston J.: Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol, *Formal Aspects of Computing*, vol. 14(3), p. 295–318, 2003.

[25] Kwiatkowska M., Norman G., Sproston J., Wang F.: Symbolic model checking for probabilistic timed automata, *Information and Computation*, vol. 205(7), pp. 1027–1077, 2007.

[26] Méry D., Rosemary M.: Transforming EVENT B Models into Verified C# Implementations. In: A. Lisitsa, A. Nemytykh (eds.), *VPT 2013 – First International Workshop on Verification and Program Transformation*, EPIC, vol. 16, pp. 57–73, 2013. https://hal.inria.fr/hal-00862050.

[27] Michael L., Michael B.: ProB: A Model Checker for B. In: *FME 2003: Formal Methods*, pp. 855–874, Springer, Berlin–Heidelberg, 2003.

[28] Morgan C., Hoang T.S., Abrial J.R.: The Challenge of Probabilistic Event B. In: *ZB 2005: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science, vol. 3455, pp. 162–171, Springer, Berlin–Heidelberg, 2005. doi: 10.1007/11415787_10.

[29] Ndukwu U., McIver A.K.: YAGA: Automated Analysis of Quantitative Safety Specifications in Probabilistic B. In: A. Bouajjani, W.N. Chin (eds.), *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, vol. 6252, pp. 378–386, Springer, Berlin–Heidelberg, 2010. doi: 10.1007/978-3-642-15643-4_31.

[30] nuXmv: https://nuxmv.fbk.eu/.

[31] PRISM benchmark suite – Models. http://www.prismmodelchecker.org/benchmarks/models.php.

[32] Sena L., Xiangyu L., Zuxi C.: Combining Theorem Proving and Model Checking in the Safety-Critical Software Development through Translating Event-B to SMV. In: *MATEC Web Conf*, vol. 128, 2017. doi: 10.1051/matecconf/201712804004.

[33] Tarasyuk A., Troubitsyna E., Laibinis L.: Towards Probabilistic Modelling in Event-B. In: *Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 6396, pp. 275–289, Springer, Berlin–Heidelberg, 2010. doi: 10.1007/978-3-642-16265-7_20.

[34] Tarasyuk A., Troubitsyna E., Laibinis L.: Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In: *Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 7321, pp. 237–252, Springer, Berlin–Heidelberg, 2012. doi: 10.1007/978-3-642-30729-4_17.

[35] Tarasyuk A., Pereverzeva I., Troubitsyna E., Laibinis L.: Formal Development and Quantitative Assessment of a Resilient Multi-robotic System. In: A. Gorbenko, A. Romanovsky, V. Kharchenko (eds.), *Software Engineering for Resilient Systems*, Lecture Notes in Computer Science, vol. 8166, pp. 109–124, Springer, Berlin–Heidelberg, 2013. doi: 10.1007/978-3-642-40894-6_9.

[36] Thomas M., Shin N.: Rodin Plugin to Link Event-B and SPIN. Technical Report, IEICE Digital Library, 2009.

Affiliations

Hichem Debbi
University of M'sila, Department of Computer Science, M'sila, Algeria,
hichem.debbi@univ-msila.dz

Received: 07.12.2021
Revised: 10.04.2022
Accepted: 15.06.2022