Computing and Informatics, Vol. 38, 2019, 85–114, doi: 10.31577/cai_2019_1_85

A PROBABILISTIC EXTENSION OF UML-B

Mohammad Nosrati, Hassan Haghighi

Shahid Beheshti University
Evin, Tehran,Iran
e-mail: mohammad.nosrati@gmail.com, h_haghighi@sbu.ac.ir

Abstract. This paper extends the graphical and formal language of UML-B to provide the ability to model probabilities. Discrete probabilities, interval probabilities, and stochastic delays are added to the UML-B's state-machine syntax, and their corresponding semantics are defined in Event-B. In addition, as a secondary contribution, UML-B (probabilistic) state-machine models are defined as MDP (Markov Decision Process) models in order to provide a means of quantitative verification in PRISM (Probabilistic Symbolic Model Checker). As an important feature of the proposed method, it does not change the Event-B syntax or semantics. To evaluate this work, as a case study, the Zeroconf protocol will be modeled in the extended UML-B using the Rodin tool, and its Event-B counterpart is converted to a PRISM model. The results of evaluations indicate that this study's additions provide the capability of modeling and verification of probabilistic and stochastic systems.

Keywords: UML-B, Event-B, probabilistic systems, interval probabilities, stochastic delay, probabilistic model verification, MDP, PRISM

1 INTRODUCTION

To facilitate software specification and design, and to simplify the communication between software stakeholders (especially, software engineers), graphical, semi-formal languages, such as UML, have been developed to model software artifacts.

In spite of the benefits that semi-formal languages provide, the lack of a precise formal semantics can lead to ambiguity and inconsistency. For example, Reggio et al. [27] have identified 31 problems concerning ambiguity, incompleteness, and inconsistency in UML 1.3. But then, the difficulty of writing formal specifications and understanding these specifications by software practitioners are serious problems. If

the priority of a modeler is to use a means for abstract communication, a less formal language is preferred. But in case the modeler is seeking semantic correctness and rigor, then formal method is favored.

One approach to increase formalism is to transform specifications notated in languages like UML to an intermediate formal language, such as Fiacre (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) [11] to perform formal analysis and verification.

As another approach to avoid the problems with semi-formal methods such as UML and to overcome the difficulties in applying formal methods like Event-B while retaining their benefits, a language called UML-B has been proposed that attempts to combine the simplicity and intuitiveness of UML and preciseness and unambiguity of Event-B.

Probability is one of the main concepts required for expressing aspects common in real-time, fault-tolerant, and distributed systems because it allows the designer to specify system's random behavior quantitatively. Due to the importance of modeling and verification of probabilistic systems and behaviors, various programming and modeling languages have covered this notion. Especially, there are a number of major approaches for formally modeling and analyzing probabilistic systems in the literature.

Probabilistic model checking is one of the commonly used techniques. In this approach, a state-based mathematical model of the probabilistic system is constructed from its description in a high level language, and then, it is examined whether this model satisfies a given probabilistic property. A number of tools have been developed based on this method, with PRISM being among the most notable ones. Güdemann et al. [14] introduced SAML (Safety Analysis Modeling Language) to unify probabilistic and logical analysis of models to decouple the model from the actual verification tool, by converting model.

Another formal approach to model probabilities is using proof-based methods and languages, such as Hoare Logic [8], B [18] and Event-B [15], which have been extended with probabilistic choice. In [18], Hoang et al. have extended Abrial's Generalized Substitution Language (GSL) [1] to get pGSL that includes random algorithms within its scope to initiate the development of probabilistic B (pB). Hallerstede et al. [15] have extended the Event-B formalism with a qualitative probabilistic choice operator. When extending proof-based methods, refinements or probabilistic invariants are used to reason about the probabilistic system. One major benefit of this approach is its automatic nature.

A third alternative approach is to use the Higher Order Logic (HOL) to formalize random systems. In this approach, random variables are expressed formally in the higher order logic, and probabilistic properties are verified in a theorem prover. Hurd's Ph.D. thesis [19] and [17] are among works in this area.

Since, despite its advantages and applications, there has been no research on the subject of modeling probabilities in UML-B, the main objective of this research is to provide the ability to model probabilistic requirements in UML-B. By this contribution, it will be possible to specify discrete and interval probabilities, as well

as stochastic delays in UML-B and transform the resulting models to the Event-B formal specification language. To achieve this goal, the syntax and semantics of the mentioned notions are defined. As a secondary contribution, this paper defines UML-B state-machines as MDP models to make it possible to convert them to PRISM models for quantitative and probabilistic verification purposes.

Despite similarities between UML and UML-B from a syntactic viewpoint, there are considerable differences between the semantics of these two languages. UML-B, concerned in our work, is based on the Event-B formal methods. Therefore, our main contribution in comparison to works like [20, 21] is our formal approach for defining the semantics of probabilistic constructs. Relying on UML-B and Event-B, the proposed method not only benefits from simplicity and intuitiveness of graphical modeling languages, but also offers the advantages of formal methods, such as lack of ambiguity in specifications, increasing accuracy and consistency, and providing means to formal verification and reasoning. In addition, in contrast to similar works, like [32], one of the main features of this paper is that it does not change the Event-B syntax and semantics to achieve its goals. Therefore, the proposed method can be directly used in the current Event-B tools.

The paper is organized as follows. In Section 2, the research background and an overview on the most related work are briefly presented. Section 4 introduces our extensions to UML-B. In Section 5, we show the applicability of the proposed method by applying it to a case study. Section 6 is dedicated to the conclusions and some directions for future work.

2 BACKGROUND

2.1 Event-B

As an evolution of B-Method developed by Jean-Raymond Abrial, Event-B [1] is a formal method for modeling and analyzing systems. In Event-B, machines model reactive (event-based) systems that continually execute enabled events. Event-based systems have no interfaces or parameters. Instead, inputs are modeled as nondeterministic changes. The state of the model is represented as a collection of variables. The dynamic behavior of the system is defined by a number of events. Events modify the system state, by executing an action [6]. An event has the following form:

$$e: \ \textbf{ANY}\ lv\ \textbf{WHERE}\ G_e\ \textbf{THEN}\ r\ \textbf{END}$$

where lv is a list of local variables, guard G_e is a predicate over the system state and local variables, and action r is a multiple assignment over the system variables. An event becomes enabled only when its corresponding guard becomes true. As a result of executing an action, one or multiple parallel assignments will be performed. Variable assignments can be deterministic or nondeterministic. The deterministic assignment is denoted by $x := E(v)$ where x is a variable and $E(v)$ is an expression over system variables. Nondeterministic assignments are denoted by $x :\in S$ or

$x :| BA_e(x, v, x')$. S is a set, and $BA_e(x, v, x')$ is a predicate over system variables. As a result of these assignments, x is assigned any value from the set S, or it gets a value x' such that $BA_e(x, v, x')$ is true [26].

An Event-B model is a tuple $(C, \Sigma, A, v, I, S, E, Init)$ where C is a set of model constants; Σ is a set of model sets; A is a set of axioms over C and Σ; v is a set of system variables; I is a set of invariant properties; S is a set of model states, defined by all possible values of v; E is a set of system events; and $Init$ is a predicate defining the set of initial states.

The semantics of events is defined using before-after (BA) predicates [2]. A before-after predicate describes a relationship between the variable values before and after the execution of an event. An event $e \in E$ is a tuple $e = (G_e, BA_e)$ where $G_e \in S \rightarrow BOOL$ is the guard and $BA_e \in S \times S \rightarrow BOOL$ is the before-after predicate [10].

Model correctness is demonstrated by generating and discharging a collection of Proof Obligations (POs). Every Event-B model should satisfy the event feasibility $((G_e(\sigma) \wedge I(\sigma')) \Rightarrow \exists \sigma'.BA_e(\sigma, \sigma'))$ and invariant properties $((G_e(\sigma) \wedge I(\sigma) \wedge BA_e(\sigma, \sigma')) \Rightarrow I(\sigma'))$, where σ and σ' are states and I is the invariant. The feasibility of an event means that whenever an event is enabled, there is some reachable after-state. Each event should also preserve the model invariant.

The behavior of an Event-B model is given by a transition system for which transition relations are given by this rule:

$$\frac{\exists (G_e, BA_e) \in E. \exists \sigma, \sigma' \in S. I(\sigma) \wedge G_e(\sigma) \wedge BA_e(\sigma, \sigma') \wedge I(\sigma')}{\sigma \rightarrow \sigma'}.$$

This rule states if there is a (G_e, BA_e) pair in the set of system events, and there are states σ and σ' such that the invariant and guard G_e are true in state σ, before-after predicate BA_e holds for states σ and σ', and finally, the invariant is true for state σ', then a transition exists from σ to σ'.

Event-B is supported by the Rodin tool [3]. The Rodin platform is an open source Eclipse-based IDE that effectively supports refinement and mathematical proof of models.

2.2 UML-B

UML-B [31] is a graphical formal notation based on the graphical notation of UML [29]. It relies on Event-B semantically, although, its initial version was translated to classical B [1].

UML-B provides four kinds of diagrams. They are package, context, class and state-machine diagrams. Package diagrams are used to describe the relationships between top level components (machines and contexts). The context diagram defines the static (constant) part of a model. Transitions of a state-machine represent events. Class diagrams are used to describe the behavioral part of a model. For further information see [30].

2.3 PRISM

Analysis and verification of a system involve establishing qualitative and quantitative properties of the system. For example, the property that states "the system eventually terminates" is a qualitative one. On the other hand, the property that states "the system terminates within a given time limit with a given probability" is a quantitative one. To perform probabilistic and quantitative verification on the resulting models of this work, we need to use a probabilistic model checker.

PRISM [24] is a model checking tool which supports verification of probabilistic models. This tool takes as input a description of a probabilistic system written in the PRISM language. It constructs a model, such as Deterministic-Time Markov Chain (DTMC), Continuous-Time Markov Chain (CTMC), or MDP from this description. It also accepts the properties specification in languages such as PCTL (Probabilistic Computation Tree Logic) [16] and performs model checking to determine which states of the model satisfy the specified property and with what probabilities. Model checking is reduced to a combination of reachability-based computation and the solution of linear equation systems. The PRISM kernel handles these computations using different engines [22].

The basic syntax of PCTL [23] is given by this grammar:

$$\Phi ::= \mathit{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid P_{\sim p}[\Box\Phi] \mid P_{\sim p}[\Phi \mathbf{U} \Phi]$$

where a is an atomic proposition, $\sim \in \{<, \leq, \geq, >\}$, and $p \in [0, 1]$. The operator $P_{\sim p}[\Phi]$ means that the probability of Φ being true in a state s in a model (such as an MDP), is $\sim p$. $\Box\Phi$ expresses that Φ is satisfied in the next step and $\Phi_1 \mathbf{U} \Phi_2$ means that Φ_2 is eventually satisfied and Φ_1 is true until then. Other useful operators can be derived from the basic PCTL syntax, such as $\Diamond\Phi \equiv \mathit{true}\ \mathbf{U}\Phi$, meaning Φ will be eventually true.

2.4 Markov Decision Process (MDP)

In this paper, we use MDPs as models to which probabilistic state-machines are converted during the translation of probabilistic UML-B to the PRISM language. So, we give a brief overview on MDPs in this subsection. An MDP [4] is a tuple $M = (S, Act, \mathbf{P}, \iota_{init}, AP, L)$, where S is a set of states, Act is a set of actions, $\mathbf{P} : S \times Act \times S \to [0, 1]$ is the transition probability function, $\iota_{init} : S \to [0, 1]$ is the initial distribution, AP is a set of atomic propositions (atomic propositions represent the basic properties that hold at some point of execution), $L : S \to 2^{AP}$ is a labeling function, and $L(s)$ are atomic propositions in AP satisfied in state s.

A Markov decision process is an extension of Markov chains that allows both probabilistic and nondeterministic choices. In any state, there is a nondeterministic choice between several discrete probability distributions over successor states.

3 RELATED WORK

In this subsection, a brief overview on the most related work is given.

3.1 Probabilities and Stochastic Delays in UML

Jansen et al. [20] have introduced randomness to UML statechart diagrams via enhanced statecharts, called probabilistic statecharts or P-statecharts. They have based the semantics of P-statecharts on Markov decision process models. Figure 1 shows an example of P-statechart models.

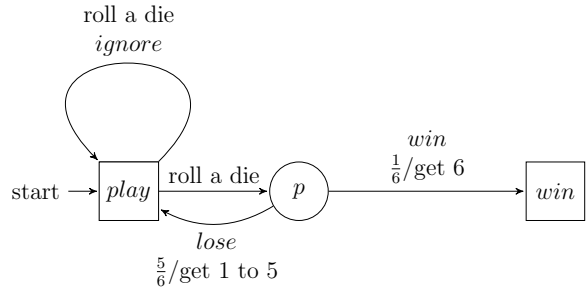

Figure 1. An example of P-statecharts

Furthermore, Jansen et al. [21] have presented StoCharts, which model random delays in statecharts. In these models, on entering a node with an outgoing edge labeled **after**(F), a sample is taken from distribution F and a timer is set accordingly. The corresponding edge becomes enabled once the timer expires. The semantics of StoCharts are defined using Stochastic I/O Automata (IOSA, for short).

3.2 Probabilities in Event-B

Hallerstede et al. [15] have extended the Event-B formalism with a new operator, qualitative probabilistic choice, denoted as $\oplus|$. The assignment $x \oplus| \ BA(v, x')$ assigns a value to x with a positive, but unknown probability. Similarly, Tarasyuk et al. [32] augmented Event-B models with the quantitative probabilistic assignment

$$x \oplus| \ x_1@p_1; \ldots; x_n@p_n$$

where $\sum_{i=1}^{n} p_i = 1$. This assignment allows for specifying exact numerical probabilities for each value x_i; variable x will have value x_i with probability p_i.

As stated before, the advantage of our approach compared to these works is that our work does not change the syntax and semantics of Event-B.

3.3 Time in Event-B

In order to model probabilistic time delays, we first need a method to model time in Event-B. In [5, 7, 28] a method for adding time to Event-B has been presented.

In this model, *time* is the current time, and *at* is the set of active times. Active times are times in future where an event might be activated. For a simple clock, *at* will be $\{time + 1, time + 2, \ldots\}$. An event called *tick_tock*, non-deterministically chooses a value for the current time, and so the progress of time is achieved. Constraint $at \neq \varnothing \Rightarrow time \leq \min(at)$ in the INVARIANT part enforces that active times are in the future, and time cannot be moved beyond the first active moment.

3.4 Probabilistic Model Checking of Event-B Models Using PRISM

Rodin [3] supports development and qualitative verification of Event-B and UML-B models, but it lacks the tools required for quantitative reasoning and verification of probabilistic systems. To enable quantitative analysis of Event-B models, one can convert Event-B models to the PRISM language. Tarasyuk et al. [34] have described the required mappings from Event-B models to the PRISM language. For example, the assignment using Tarasyuk's probabilistic choice operator $\oplus|$, (i.e. $x \oplus| \; x1@p1; \ldots xn@pn)$) [26] is expressed as the following command in PRISM:

$$[]\ true -> p1 : (x' = x1) + \ldots + pn : (x' = xn).$$

4 PROBABILISTIC UML-B

4.1 Overall Structure

In this section, in order to add abilities for modeling probabilistic and stochastic systems through the notion of state-machines in UML-B, a number of new structures are added to its graphical syntax. The corresponding semantics are also defined in Event-B.

The most basic probabilistic structure that needs to be addressed is the ability to specify discrete probabilities. This applies to scenarios where the set of possible outcomes is discrete, such as a coin toss. We take one step further and also take into account scenarios where the exact values of probabilities are unknown, but their intervals are known. We introduce a solution to model interval probabilities in this condition. Another important feature that can benefit modelers is the capability to model discrete stochastic delays, where one can specify a random amount of time before moving to the next state. Stochastic delays are present in many distributed and networking systems, and even in biological processes. We cover three types of stochastic delays: fixed time delay, uniform distribution delay and geometric distribution delay.

For every structure that we define, its semantics is also defined in Event-B. For discrete probabilities and interval probabilities, the overall approach is to update the

current state of the UML-B machine based on probabilities specified by the modeler. A random number will be generated, and the generated number will determine the target state. For stochastic delays, a timeout value is calculated upon entering a state which has an outgoing transition with a delay. The outgoing transition will have a guard to ensure a delay. The guard protects from entering the next state without first waiting in the current state for a duration of at least the calculated timeout.

The ability to generate random sequences of numbers which are uniformly distributed is essential in any work related to probabilities. Event-B does not have built-in support for generating random numbers. Therefore, we need to use an algorithm that generates sequences of numbers that are close enough to a true sequence of random numbers. Section 4.2 discusses our approach to define a random generator in Event-B. This random generator is one of the core elements of the semantics defined for most structures in this work.

Section 4.3 discusses adding discrete probabilities to the state-machine diagrams. Section 4.4 presents how to add interval probabilities to state-machines. Section 4.5 outlines stochastic delays and discusses Fixed-time, Uniform Distribution, and Geometric Distribution delays. Section 4.6 defines UML-B state-machines as MDPs to provide a theoretical basis for translating UML-B state-machines to PRISM models. Using this basis, we present the translation method in Section 4.7. The resulting PRISM models make it possible to verify probabilistic properties in the initial UML-B models, quantitatively.

4.2 The *random* Function

Throughout this section, we use the function *random* to generate uniform random numbers in the range $[l, u]$. This function can be defined using any common pseudo-random generators such as a linear congruential generator, which uses the recurrence

$$X_n = l + (aX_{n-1} + c) \mod (u - l + 1) \tag{1}$$

where X_n and X_{n-1} are respectively the next and current pseudo-random numbers, and a and c are large integer numbers. X_0 is called the seed or start value. Axioms in Figure 2 define a basic pseudo-random generator. *seed* can be a machine variable to store the last generated random value as a seed to the next iteration of the function.

4.3 Adding Discrete Probabilities to the State-Machine Diagram

We first introduce a structure for specifying discrete probabilities in UML-B. It is worth noting that only probabilities that have rational values are supported.

4.3.1 Discrete Probabilities Syntax

We propose to specify discrete probabilities using a new notion which we call pseudo-states. The difference between a pseudo-state and a normal state is that state-

AXIOMS
$random \in \mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$
$\forall l, u, seed, r.$
$l \in \mathbf{N} \wedge u \in \mathbf{N} \wedge seed \in \mathbf{N} \wedge r \in \mathbf{N} \Rightarrow$
$(l \mapsto u \mapsto seed \mapsto r \in random \Leftrightarrow$
$r = l + (a \times seed + c) \mod (u - l + 1))$
END

Figure 2. The *random* function

machines will not stay in pseudo-states; the role of pseudo-state is only to determine how the transitions from previous states to next states are done. Figure 3 shows the new structure to specify discrete probabilities using a pseudo-state p.

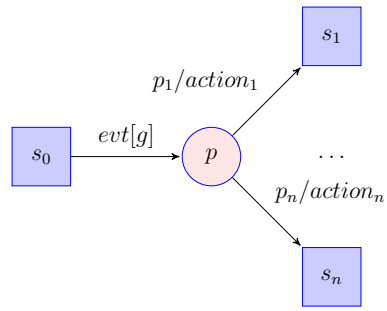

Figure 3. Discrete probabilities in UML-B

In Figure 3, if the edge with probability p_i is selected, $action_i$ will be performed. For each $i \in 1 \ldots n$, there exist numbers m_i and d, where $p_i = \frac{m_i}{d}$, and

$$d \neq 0 \wedge (m_i \in \mathbf{N}_1) \wedge \sum_{i=1}^{n} m_i = d. \tag{2}$$

Probabilities are defined by rational fractions, in form of natural numerators and denominators. If the model's probabilities are in form of $p_1 = q_1/d_1, \ldots, p_n = q_n/d_n$, the positive number d is the least common multiple of these denominators ($d = \mathbf{lcm}(d_1, \ldots, d_n)$) and $m_i = q_i \frac{d}{d_i}$.

4.3.2 Discrete Probabilities Semantics

Intuitively, semantics of the given structure in Figure 3 can be described as follows. When the state-machine is in state s_0, if the event evt is selected, and its guard g holds, the machine will enter into state s_i and will perform the correspondent $action_i$ with the probability p_i ($i \in 1 \ldots n$).

Formally, the semantics of the given structure in Event-B is defined as in Figure 4.

$$
\begin{array}{l}
evt: \\
\quad \textbf{WHERE} \\
\quad\ \mathrm{STATE} = s_0 \wedge g \\
\quad \textbf{THEN} \\
\quad\ \mathrm{STATE}, seed :| \exists r.r = random(1 \mapsto d \mapsto seed) \ \wedge \\
\quad\quad seed' = r \wedge \\
\quad\quad ((r \leq m_1 \Rightarrow \mathrm{STATE}' = s_1 \wedge action_1) \wedge \\
\quad\quad (r > m_1 \wedge r \leq m_1 + m_2 \Rightarrow \mathrm{STATE}' = s_2 \wedge action_2) \wedge \\
\quad\quad \ldots \\
\quad\quad (r > \sum_{j=1}^{i-1} m_j \wedge r \leq \sum_{j=1}^{i} m_j \Rightarrow \mathrm{STATE}' = s_i \wedge action_i) \wedge \\
\quad\quad \ldots \\
\quad\quad (r > \sum_{j=1}^{n-1} m_j \wedge r \leq \sum_{j=1}^{n} m_j \Rightarrow \mathrm{STATE}' = s_n \wedge action_n)) \\
\quad \textbf{END}
\end{array}
$$

Figure 4. Discrete probabilities in Event-B

Number r is randomly selected from numbers 1 to d. The next state will be selected as follows: If the random number r is less than or equal to m_1, the next state is s_1; if r is greater than m_1 and is less than or equal to $m_1 + m_2$, the next state is s_2 and so on.

Theorem 1. The semantics for discrete probabilities, given in Figure 4, provides the expected probabilities for the corresponding actions.

Proof. Variable r is randomly chosen from one of d numbers $(1 \ldots d)$ with equal probabilities. Now, whenever the condition:

$$r > \sum_{j=1}^{i-1} m_j \ \wedge r \leq \sum_{j=1}^{i} m_j \tag{3}$$

holds, the assignment $\mathrm{STATE} := s_i$ will be made. So, the probability of transition to s_i is:

$$\Pr\{\mathrm{STATE} := s_i\} = \frac{\sum_{j=1}^{i} m_j - \sum_{j=1}^{i-1} m_j}{d} = \frac{m_i}{d} \tag{4}$$

which is equal to the expected probability p_i. □

4.4 Interval Probabilities

Interval probabilities are used when the probabilistic design is abstract and underspecified [9, 12]. It is assumed that in the specification stage, the exact values of

probabilities are unknown, but their intervals are known, and will probably become exact in the next stages of the refinement.

4.4.1 Interval Probabilities Syntax

Similar to the discrete case, interval probabilities are specified here via a pseudo-state p. Figure 5 shows the structure we propose to model interval probabilities.

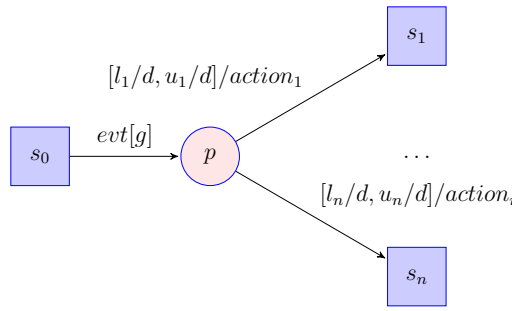

Figure 5. Interval probabilities in UML-B

In Figure 5, u_i and l_i $(i \in 1 \ldots n)$ are positive integers less than or equal to d. Intervals that the modeler chooses must allow for selecting a number from each interval such that the sum of the selected numbers is equal to 1. For example, suppose there are two branches in Figure 5. If the modeler chooses interval $[0, 0.3]$ for the first branch and interval $[0.2, 0.8]$ for the other branch, since the number 0.2 can be chosen from the first interval and 0.8 from the second one, and $0.2 + 0.8 = 1$, the selected intervals are allowed. But for $I_1 = [0, 0.3]$ and $I_2 = [0.2, 0.6]$, there are no two numbers $p_1 \in I_1$, $p_2 \in I_2$ such that $p_1 + p_2 = 1$ and therefore, these intervals are not allowed.

4.4.2 Interval Probabilities Semantics

The semantics of the structure given in Figure 5 is defined as in Figure 6. Number r is chosen from 1 to d, and each m_i $(i \in 1 \ldots n)$ is chosen from its respective interval. r will fall into one of the intervals formed by m_is and its value determines the next value for STATE.

Theorem 2. The semantics for interval probabilities, given in Figure 6, provides the expected probabilities for the corresponding actions.

Proof. We prove that the probability of moving to the state s_i will be within the specified interval.

Since there exists the condition $m_i \in l_i \ldots u_i$, the inequality

$$\frac{l_i}{d} \leq \frac{m_i}{d} \leq \frac{u_i}{d} \tag{5}$$

evt :
WHERE
STATE $= s_0 \wedge g$
THEN
STATE :| $\exists r, m_1, \ldots, m_n . r = random(1 \mapsto d \mapsto seed) \ \wedge$
$seed' = r \ \wedge$
$m_1 \in l_1 \ldots u_1 \ \wedge$
$\ldots$
$m_n \in l_n \ldots u_n \ \wedge$
$\sum_{j=1}^{n} m_j = d \ \wedge$

$((r \leq m_1 \Rightarrow \text{STATE}' = s_1 \wedge \ action_1) \ \wedge$
$(r > m_1 \wedge r \leq m_1 + m_2 \Rightarrow \text{STATE}' = s_2 \wedge action_2) \ \wedge$
$\ldots$
$(r > \sum_{j=1}^{i-1} m_j \ \wedge r \leq \sum_{j=1}^{i} m_j \Rightarrow \text{STATE}' = s_i \wedge action_i) \ \wedge$
$\ldots$
$(r > \sum_{j=1}^{n-1} m_j \ \wedge r \leq \sum_{j=1}^{n} m_j \Rightarrow \text{STATE}' = s_n \wedge action_n))$
END

Figure 6. Interval probabilities in Event-B

holds. Variable r is randomly chosen from one of d numbers $(1 \ldots d)$ with equal probabilities. Now, whenever the condition:

$$r > \sum_{j=1}^{i-1} m_j \ \wedge r \leq \sum_{j=1}^{i} m_j \tag{6}$$

holds, the assignment $\text{STATE}' = s_i$ will be made. Therefore, for the probability of the transition to s_i, the following holds:

$$\frac{l_i}{d} \leq \Pr\{\text{STATE}' = s_i\} = \frac{\sum_{j=1}^{i} m_j - \sum_{j=1}^{i-1} m_j}{d} = \frac{m_i}{d} \leq \frac{u_i}{d}. \tag{7}$$

Therefore, the probability of moving to the destination state is within the desired interval. □

4.4.3 An Alternative Method to Define the Semantics

In this subsection, one alternative semantics that can be used instead of the semantics presented in Section 4.4.2 is introduced. In Section 4.7.5, we will need this semantics to translate the probabilistic UML-B constructs to PRISM, because in PRISM it is not possible to resolve both the non-determinism and probabilities in one transition. This new semantics is defined using an additional state. We consider

an additional real state P_INTERVAL for resolving the non-determinism in probabilities, and computing discrete probabilities based on interval probabilities, and then, determining the next state by using these computed probabilities. In this way, the state-machine in Figure 5 is defined as the structure in Figure 7. The proof that the semantics given in Figure 8 provides the expected probabilities for the corresponding actions, is similar to the proof given in Theorem 2 with minor differences; so, we do not present this proof anymore. It should be noted that because of the introduction of a new real state, and the possibility of a delay, the definition given in Figure 8 is not exactly equivalent to the definition provided in Figure 6.

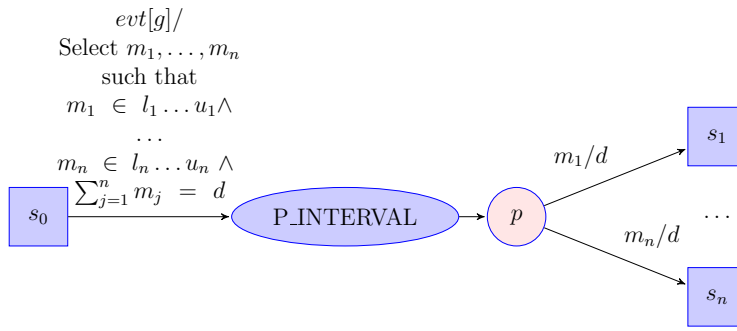

Figure 7. The semantics of interval probabilities by using an additional state

The pseudo-code for event evt is composed of evt_1 and evt_2 as shown in Figure 8; for each outgoing edge with label m_i/d, a machine variable m_i is defined.

$evt \mathrel{\hat{=}} evt_1 \ ; \ evt_2$

evt_1 :
WHERE
STATE $= s_0 \wedge g$
THEN
$m_1, \ldots, m_n :| \ m_1' \in l_1 \ldots u_1 \wedge \ldots$
$\wedge m_n' \in l_n \ldots u_n \wedge \sum_{j=1}^{n} m_j = d$
STATE := P_INTERVAL
END

evt_2 :
WHERE
STATE = P_INTERVAL
THEN
STATE $:| \ \exists r.r = random(1 \mapsto d \mapsto seed) \ \wedge$
$seed' = r \wedge$
$((r > 0 \wedge r \leq m_1 \Rightarrow \text{STATE}' = s_1 \wedge action_1) \wedge$
$(r > m_1 \wedge r \leq m_1 + m_2 \Rightarrow \text{STATE}' = s_2 \wedge action_2) \wedge$
$\ldots$
$(r > \sum_{j=1}^{i-1} m_j \wedge r \leq \sum_{j=1}^{i} m_j \Rightarrow \text{STATE}' = s_i \wedge action_i) \wedge$
$\ldots$
$(r > \sum_{j=1}^{n-1} m_j \wedge r \leq \sum_{j=1}^{n} m_j \Rightarrow \text{STATE}' = s_n \wedge action_n))$
END

Figure 8. The pseudo-code for evt as a composition of evt_1 and evt_2

4.5 Discrete Stochastic Delay

In some systems, an action may be performed when a specific amount of time is passed after reaching to a state. The duration of this delay can be fixed or can

be probabilistically selected based on a distribution function. In this subsection, the delay structure is added to UML-B. We restrict our work to discrete times and delays.

To indicate the activation of a transition after time t, guard **after**(t) is added to the UML-B syntax. In addition, for specifying the notion of probabilistic time, time t can be specified randomly. In other words, instead of **after**(t), **after**(F) is used where $F : \mathbf{N} \rightarrow [0, 1]$ is the distribution function of timeout. For delays corresponding to the geometric distribution with parameter $p = \frac{m}{d}$, and for delays with the uniform distribution, **after**$(\mathrm{G}(m/d))$ and **after**$(\mathrm{UNIF}(t_{min}, t_{max}))$ are respectively used, where, t_{min} and t_{max} are minimum and maximum values of delay. We restrict our work to the uniform and geometric distributions. We also consider distributions that have a random generator function. After passing the delay time, one of the output edges will be activated and the machine will transit to one of the target nodes.

Since Event-B does not have the notion of time, the method presented in [28] and reviewed in Section 3.3, is used to express the concept of time. The auxiliary variables and events in Figure 9 are added to the Event-B model to handle the notion of time:

VARIABLES
$time, at$
INVARIANT
$time \in 0 \ldots MAX\_TIME \ \wedge$
$at \subseteq MAX\_TIME \ \wedge$
$(at \neq \varnothing \Rightarrow time \leq \min(at))$
INITIALISATION
$time := 0$
$at := 0 \ldots MAX\_TIME$
EVENTS
$tick\_tock$:
ANY
tm
WHERE
$tm \in MAX\_TIME \wedge$
$tm > time \ \wedge$
$(at \neq \varnothing \Rightarrow tm \leq \min(at))$
THEN
$time := tm$
END

$process\_time$:
WHERE
$time \in at$
THEN
$at := at - time$
END
END

Figure 9. Auxiliary variables and events for handling time in Event-B

In Figure 9, variable *time* is the current value of time, and *at* is the remaining active times at which the *process_time* event will be triggered. The *tick_tock* event increases the value of *time*, and the *process_time* event removes the current value of *time* from active times. To avoid state explosion and keeping the model finite, times are restricted by constant *MAX_TIME*.

In the next subsections, we will use variable *time* to compute the timeout moment and also to obtain the time at which an event with a guard containing a delay can be executed.

4.5.1 Fixed Time Delay

In this subsection, at first the syntax of the fixed time delay is introduced in UML-B, and then, its semantics is presented.

a. Fixed Time Delay Syntax. The structure specified in Figure 10 is added to UML-B. The parameter $t \in \mathbf{N}$ is the amount of delay before actions of *evt* can be executed.

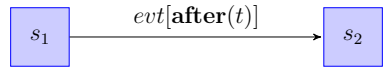

Figure 10. The fixed time delay in UML-B

b. Fixed Time Delay Semantics. The structure specified in Figure 10 is translated to the structure in Figure 11. We define T as the set of all transitions that their destinations are the starting state s_1 (e.g. STATE := s_1) in the machine. Every such statement must be accompanied by a parallel assignment $timeout := time + t$ to specify the amount of timeout.

For every transition, starting from the state s_1 that has an $after$ condition, the guard $time \geq timeout$ must be added. If there are more than one $after$ condition, different *timeout* variables must be defined (i.e. $timeout_0, timeout_1, \ldots$), and for each $after$ condition, the corresponding variable must be used. All these *timeout* variables should be initialized at the moment that STATE := s_0 is being done.

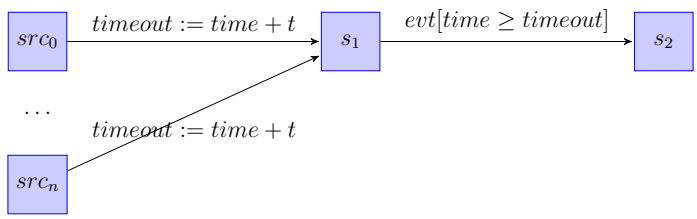

Figure 11. The structure equivalent to the syntax in Figure 10

The events in Figure 11 are defined as in Figure 12.

Theorem 3. The semantics for fixed time delay, given in Figure 12, provides the expected delays for the corresponding actions.

Proof. We prove that the lower bound of the delay for the transition STATE := s_2 is the constant t. If at any moment in the interval $[t_0, t_0 + t)$ (t_0 is the moment that the timeout is set), the event *evt* is chosen for execution, the guard $time \geq timeout = t_0 + t$, which is necessary for moving to the state s_2, will not hold and the transition will not be done. At the time $t_0 + t$, the evaluation of this guard will change to *true*. Therefore, the lower bound of the transition to the destination state is $t_0 + t - t_0 = t$. □

All transitions STATE := s_1 in T will be accompanied by an assignment setting the timeout:

STATE := s_1
$timeout := time + t$

evt :
WHERE
STATE = $s_1 \wedge time \geq timeout$
THEN
STATE := s_2
END

Figure 12. The fixed time delay in Event-B

4.5.2 Uniform Distribution Delay

In this subsection, at first the syntax of the uniform distribution time delay is introduced in UML-B, and then, its semantics is presented.

a. Uniform Distribution Delay Syntax. The structure in Figure 13 is added to UML-B. The parameters $U, L \in \mathbf{N}$ are the lower and upper bounds of delay before actions of *evt* can be executed.

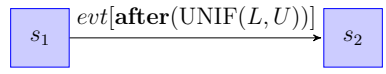

Figure 13. The uniform distribution delay in UML-B

b. Uniform Distribution Delay Semantics. Intuitively, the diagram in Figure 13 expresses that when the state-machine is in s_1, it will move to s_2 with a delay which equals to a random time selected uniformly in the interval L to U.

For defining the semantics formally, the action $timeout := time + t$ in Figure 12 is changed to the following action:

$$timeout, seed :| \exists r.r = random(L \mapsto U \mapsto seed) \wedge \\ timeout' = time + r \ \wedge \ seed' = r. \tag{8}$$

Theorem 4. The semantics for uniform distribution delay, given in Figure 11 with the change mentioned in Equation (8), provides the expected delays for the corresponding actions.

Proof. We prove that the lower bound of the delay for the transition STATE $:= s_2$ is a random number from the interval $[L, U]$. We designate t_0 to the moment the variable *timeout* is evaluated. At t_0, *timeout* will be assigned a random value in the interval $[t_0 + L, t_0 + U]$. If at any moment in the interval $[t_0, timeout)$, the event *evt* is chosen for execution, the guard $time \geq timeout$, which is necessary for moving to the state s_2, will not hold and the transition will not be done. At the time $t_0 + timeout$, the evaluation of this guard will change to *true*. Therefore, the lower bound of the delay is an integer number in $[L, U]$. □

4.5.3 Geometric Distribution Delay

Many continuous-time systems exhibit delays with exponential distribution. The exponential distribution describes the time for a continuous process to change state or an event to occur. The discrete analog for exponential distribution is the geometric distribution. This distribution describes the number of Bernoulli trials needed for the first success. Therefore, the geometric distribution can be seen as describing the number of steps a discrete process needs to change state. If X is a geometrically distributed random variable, and the probability of success on each trial is p, the probability of $X = k, k \in \mathbf{N}_1$ is $(1-p)^{k-1}p$. In this subsection we propose a syntax and semantics for the geometric delay.

a. Geometric Distribution Delay Syntax. Figure 14 shows the geometric distribution delay structure in UML-B. The fraction m/d is the parameter of the geometric distribution.

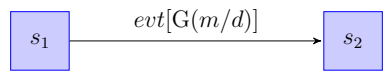

Figure 14. Geometric distribution delay in UML-B

b. Geometric Distribution Delay Semantics. The semantics for the geometric distribution is similar to the one for the uniform distribution except that it uses the function *geometric* instead of *random*. The main difficulty lies in the problem of generating numbers from the geometric distribution. We define the function

geometric as an axiom to generate numbers from the geometric distribution. Since the seed for the *random* function needs to be different for each iteration of *random*, we use recursion and ideas from dynamic programming for storing the previous values of the *random* function in the function *rand*. Figure 15 shows the definition of the function *geometric*.

The semantics is defined by changing the action $timeout := time + t$ in Figure 12 to the following action:

$$\begin{aligned} timeout, seed :| \ & \exists r, g.g \mapsto r = geometric(m \mapsto d \mapsto seed) \wedge \\ & timeout' = time + g \ \wedge \ seed' = r. \end{aligned} \tag{9}$$

AXIOMS

$\forall n, X, rand, m, d, seed, g, newSeed.$
$n \in 0 \ldots MAX\_TIME \wedge X \in \mathbf{N} \to 0 \ldots 1 \wedge m \in \mathbf{N} \wedge d \in \mathbf{N} \wedge$
$rand \in \mathbf{N} \to \mathbf{N} \wedge seed \in \mathbf{N} \wedge newSeed \in \mathbf{N} \wedge g \in 0 \ldots MAX\_TIME \Rightarrow$
$rand(0) = random(1 \mapsto d \mapsto seed) \wedge$
$rand(n) = random(1 \mapsto d \mapsto rand(n-1)) \wedge$
$((rand(n) \leq m \Leftrightarrow X(n) = 0) \wedge$
$(rand(n) > m \Leftrightarrow X(n) = 1)) \wedge$
$g = \min(\{j | j \in 0 \ldots MAX\_TIME \wedge j \mapsto 1 \in X\}) \wedge$
$newSeed = rand(g) \wedge$
$g \mapsto newSeed = geometric(m \mapsto d \mapsto seed)$

END

Figure 15. The definition of the *geometric* function

Theorem 5. The function *geometric* generates numbers from the geometric distribution.

Proof. The function *rand* is a sequence of randomly generated numbers. The function X is a sequence of 0 or 1s, with success probability $p = \frac{m}{d}$. X can be considered as a sequence of results of independent Bernoulli trials. The set $S = \{j | j \in 0 \ldots MAX\_TIME \wedge j \mapsto 1 \in X\}$ is the set of indices of all successful Bernoulli trials, and $g = \min(S)$ is the index of the first successful Bernoulli trial. For any j ($j \in 0 \ldots MAX\_TIME$), the probability of $g = j$ equals to $(1-p)^j p$. Therefore, *geometric* generates numbers with geometric distribution with parameter $p = \frac{m}{d}$. □

Theorem 6. The semantics for geometric distribution delay, given in Figure 11 with the change mentioned in Equation (9), provides the expected delays for the corresponding actions.

Proof. We prove that the lower bound of the delay for the transition STATE := s_2 is a random number with the geometric distribution. We designate t_0 to the moment the variable *timeout* is evaluated. At t_0, *timeout* will be assigned the value $t_0 + geometric(\frac{m}{d})$. As mentioned above, values of $geometric(\frac{m}{d})$ have the geometric distribution with parameter $p = \frac{m}{d}$. If at any moment in the interval $[t_0, timeout)$, the event *evt* is chosen for execution, the guard $time \geq timeout$, which is necessary for moving to the state s_2, will not hold and the transition will not be done. At the time $t_0 + timeout$, the evaluation of this guard will change to *true*. Therefore, the lower bound of the delay for the transition to the destination state is a number with the geometric distribution. □

4.5.4 Generalization – Arbitrary Discrete Distribution

For random delays with arbitrary distributions, if the generator function F is available, one can model the delay through a method similar to that of the previous subsection. It is sufficient to replace the timeout assignment in Figure 12 with the appropriate assignment.

4.6 UML-B State-Machine as a Probabilistic Transition System

In this subsection, the state-machine models are defined as MDPs. The objective is to have a theoretical ground for translating UML-B state-machines to PRISM models for quantitative and probabilistic model checking. The provided definition is similar to the definitions in [10] and [33] in which Event-B models are defined as Transition Systems. In addition to standard Event-B structures and assignments, defined in [10] and [33], our definition takes into account probabilities and the current state in UML-B state-machines.

In order to define UML-B state-machines as MDPs, we need to slightly change the definition of the projection operator π [13] to extract a component of a tuple by its name, not its index.

Definition 1. Let $A_1, \ldots, A_n$ be sets, and $i \in 1 \ldots n$. If $T \subseteq A_1 \times \ldots \times A_n$, then function $\pi_i : T \to A_i$ is defined by $\pi_i(a_1, \ldots, a_n) = a_i$. If the i^{th} component of tuple $(a_1, \ldots, a_n)$ is denoted by variable v, we define $\pi_v(a_1, \ldots, a_n) = a_i$.

Definition 2. Every UML-B state-machine model is defined as tuple $M = (S, E, \mathbf{P}, \iota_{init}, AP, L)$ where:

- S is the set of states of the state-machine. Let $v_1, \ldots, v_n$ be the variables of the machine. We define V_i to be the type of variable v_i $(i \in 1 \ldots n)$. Therefore, $S = V_1 \times \ldots \times V_n$. The state-machine has at least one variable STATE which specifies the current state of the state-machine.
- $E \subseteq S \times S$ is the set of all events of the state-machine. Any event $evt(s)$ is defined as:

$$evt(s) : \textbf{WHERE } G_{evt}(s) \textbf{ THEN } R_{evt}(s) \textbf{ END}$$

where s is the current state of the state-machine, $G_{evt} : S \to BOOL$ is the event guard, and $R_{evt} \subseteq S \times S$ determines the relationship between the current state and the next state (by one or multiple assignments). For example, for the following event:

$$evt : \textbf{WHERE } x = 1 \textbf{ THEN } x := 0 \textbf{ END}.$$

We have:

$$\forall s.s \in S \Rightarrow (\pi_x(s) = 1 \Leftrightarrow G_{evt}(s) = true),$$
$$\forall s, s'.s, s' \in S \Rightarrow (\pi_x(s') = 0 \Leftrightarrow (s, s') \in R_{evt}).$$

In a state-machine, there is an edge from SM_STATE to SM_STATE' (which are two states in the state-machine) if and only if:

$$\begin{aligned} \exists s, s'.G_{evt}(s) \wedge I(s) \wedge SM\_STATE = \pi_{\text{STATE}}(s) \wedge \\ (s, s') \in R_{evt} \wedge SM\_STATE' = \pi_{\text{STATE}}(s') \wedge I(s') \end{aligned} \tag{10}$$

where $I : S \to BOOL$ is the machine invariant. This predicate means there exists a transition from SM_STATE to SM_STATE', if and only if the invariant holds in states s and s', STATE is equal to SM_STATE, and s is R_{evt}-related to s'.

- $\mathbf{P} : S \times E \times S \to [0, 1]$ is the transition probability function:

$$\mathbf{P}(s, evt, s') = \begin{cases} 0, & \neg(I(s) \wedge G_{evt}(s) \wedge I(s') \wedge (s, s') \in R_{evt}), \\ p_{evt}(s, s'), & I(s) \wedge G_{evt}(s) \wedge I(s') \wedge (s, s') \in R_{evt} \end{cases} \tag{11}$$

  where $p_{evt}(s, s')$ is the probability of going from state s to s'.
- $\iota_{init} : S \to [0, 1]$ is the initial distribution. This is determined using the INITIALIZATION statements of the machine and determines the probability of being at various states of the machine at time 0.
- $AP = \varnothing$ is the set of atomic propositions. Since we are not labeling our MDP states, the set of atomic propositions is empty.
- $\forall s \in S. L(s) = \varnothing$. We simply choose not to label any MDP state.

In state s, event evt is selected according to the transition probability function $\mathbf{P}$, and the machine is transited to state s' according to $p_{evt}(s, s')$.

By this definition, every UML-B state-machine model can be described as a Markov Decision Process.

4.7 Translating Probabilistic UML-B State-Machines to PRISM

The PRISM tool can receive its inputs as MDP models. As indicated in Section 4.6, a state-machine in UML-B can be interpreted as an MDP. In this subsection, we

use this correspondence to present a method for translating a UML-B state-machine model to a PRISM model for the purpose of quantitative model checking. To automate the process of translating from UML-B state-machines to PRISM models, a number of straightforward conversions are presented. The proposed method is similar to the method presented in [34], in which a number of conversions from Event-B to PRISM are shown, including translating actions containing the $\oplus|$ operator. The conversions proposed in this section can be performed indirectly through the methods of the aforementioned work to translate models to PRISM models. But, we adapt the conversions to the specific structures present in UML-B and our probabilistic extension of it, to achieve a more readable and clear final PRISM model.

In what follows, we present the needed translation for each UML-B construct in order to translate a UML-B model to a PRISM model.

4.7.1 State-Machine and Its States

Let STATEMACHINE be the name of the state-machine, and its states be S1 to Sn; now, variable SM_STATE of type $[0 \dots n-1]$ is defined in PRISM. Constants $s1$ to sn with values 0 to $n-1$ are also defined to specify the different states of the machine. The initial state of the state-machine is defined in the **init** section (Figure 16).

const int $s1 = 0$;
$\dots$
const int $sn \ = n-1$;
global SM_STATE $[s1 \dots sn]$ **init** $s1$;

Figure 16. State-machine states in PRISM

For each state-machine, a module with the name of that state-machine is defined to model its transitions (i.e. **module** statemachine $\dots$ **endmodule**). The body of a module will be the transitions taking place in the module.

4.7.2 State Transition

Figure 17 indicates a state transition in UML-B and its translation in PRISM. Guard SM_STATE $= s1$ determines if the model is in $s1$. Transition SM_STATE$' = s2$ changes the current state.

When a transition also has guards and actions, they will be included, too (Figure 18).

Figure 17. Left: A state transition in UML-B. Right: Its translation in PRISM.

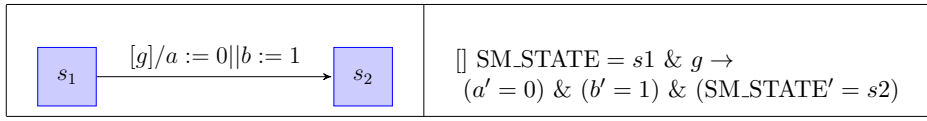

Figure 18. Left: A state transition with guards and actions. Right: Its translation in PRISM.

4.7.3 Nondeterministic Transition

Nondeterministic transitions which have the same guards are translated as shown in Figure 19.

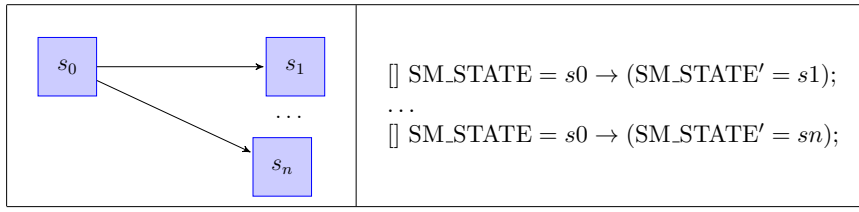

Figure 19. Left: Nondeterministic transitions. Right: Their translation in PRISM.

4.7.4 Discrete Probabilities

In order to translate the structure shown in Figure 20, the probabilistic selection in PRISM is used.

4.7.5 Interval Probabilities

We take the approach discussed in Section 4.4.3 and Figure 7 to perform the translation of interval probabilities to PRISM. Variable INTERVALP is used for maintaining the state to which the state-machine will move after the probabilistic selection. Variables m_1 to m_n are defined to keep each transition probability's fraction's numerator. Every combination of $m_1 + \ldots + m_n$, where $L_1 \leq m_1 \leq U_1, \ldots, L_n \leq m_n \leq U_n$, is considered; and $m_1, \ldots, m_n$ are assigned non-deterministically. d is the denominator of probabilities fractions.

In order to generate all valid transitions, for each numerator variable m_i ($i \in 1 \ldots n$), a value is taken from its corresponding interval $L_i \ldots U_i$. For example, the first transition shown in Figure 21 takes the lower bound of each numerator variable

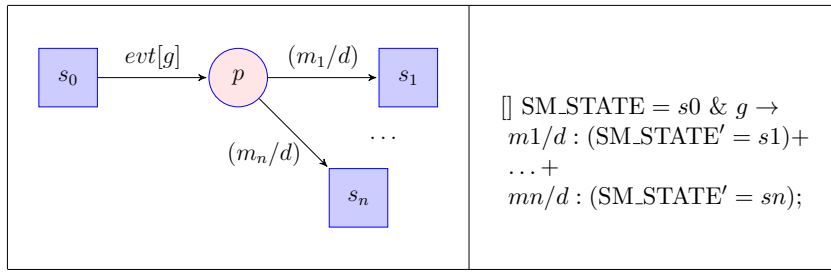

Figure 20. Left: A probabilistic transition in UML-B. Right: Its PRISM counterpart.

[] SM_STATE = $s0$ & $(L1 + \ldots + Ln = d) \rightarrow$
$(m1' = L1) \& \ldots \& (mn' = Ln) \&$
(SM_STATE′ = INTERVALP);
[] SM_STATE = $s0$ & $((L1 + 1) + \ldots + Ln = d) \rightarrow$
$(m1' = L1 + 1) \& \ldots \& (mn' = Ln) \&$
(SM_STATE′ = INTERVALP);
. . .
[] SM_STATE = $s0$ & $(U1 + \ldots + Un = d) \rightarrow$
$(m1' = U1) \& \ldots \& (mn' = Un) \&$
(SM_STATE′ = INTERVALP);

[] SM_STATE = INTERVALP & $m1 + \ldots + mn = d \rightarrow$
$m1/d$: (SM_STATE′ = $s0$)+
. . .
mn/d : (SM_STATE′ = sn);

Figure 21. Interval probabilities in PRISM

as its value, and adds a guard to check if $\sum_i m_i$ equals to d. From those transitions for which this guard is true, one is chosen non-deterministically and SM_STATE will move to INTERVALP. The last transition in Figure 21 computes the probability of moving to every state and performs the transition of SM_STATE.

4.7.6 Time

A module named *tick_tock* is defined for advancing the integer variable time. The transition [] $time < MAX\_TIME- > (time' = time + 1)$; is used for advancing time. Constant *MAX_TIME* is used to limit the possible values of variable time to avoid state explosion.

4.7.7 Fixed Time Delay

In every ingoing transition to a state in which a fixed time delay exists, the integer variable $after\_time$ is set to $time + t$, where t is the fixed delay. In the ongoing transitions that include guard **after**(t), condition $time >= after\_time$ is added to make sure those transitions are activated only when the specified time has passed.

4.7.8 Uniform Distribution Delay

This case is similar to the fixed time delay, but the *after_time* variable is set randomly. In the ingoing transitions to the state in which condition **after**(UNIF(L, U)) exists, *after_time* is set to one of the values $time + L$, $time + (L + 1)$, ..., $time + U$, with equal probability.

4.7.9 Geometric Distribution Delay

Because of the lack of recursive function support in the current version of PRISM, the translation is not yet possible.

5 CASE STUDY

In this section the applicability of the probabilistic extension of UML-B is illustrated through a case study.

5.1 Zeroconf Configuration Protocol

In this case study, we consider the Zeroconf configuration protocol for local addresses [25]. This protocol configures an IP address for a newly joined device to the local network. When a host connects to the network, it first randomly selects an IP address from a pool of 65 024 available addresses in the range of 169.254.1.0 to 169.254.254.255. The host waits for a random time between 0 and 2 seconds before starting to send four Address Resolution Protocol (ARP) packets, called probes, to all other hosts. These probes contain the IP address selected by the host, and are sent at 2 seconds intervals. A host which is already using this address will respond with an ARP reply packet, and the original host will restart reconfiguration. If the host encounters 10 IP conflicts, it remains idle for 1 minute. If the host sends four probes without receiving any ARP reply packet, then it starts to use the chosen IP address. This host sends two further messages, called gratuitous ARPs, at 2 seconds intervals. A host that has started using an IP address must reply to ARP packets containing the same IP address. It continues to use the address unless it receives a gratuitous ARP containing the same IP address. In this case, the host can either defend its IP address, or defer to the conflicting host. The host may only defend its address if it has not received a previous conflicting packet within the previous ten seconds; otherwise, it must defer. A defending host sends an ARP packet containing the IP address. A deferring host restarts the protocol and reconfigures.

5.2 Modeling the Zeroconf Protocol in UML-B

We consider one host, which is trying to configure its IP address in a network of N other hosts. If the number of all available IP addresses is IP, then the probability of the host choosing a fresh IP address is $(IP - N)/IP$. Possible values for IP addresses are abstracted to values 1 and 2. Value 1 represents an IP address already assigned to a host in the network. Value 2 represents a fresh IP address. Also, the delay over the interval $[0, 2]$ is abstracted to a choice over $\{0, 1, 2\}$.

Figure 22 shows the UML-B state-machine model for the host. In the RECONF state, the host chooses a new IP address denoted by *iph*, by moving to the CHOOSE state. If it has encountered 10 address conflicts, it moves to the CHOOSEWAIT state and chooses a new address after waiting for one minute. In states CHOOSE and CHOOSEWAIT, the host probabilistically selects an address. The random delay before sending probes is modeled using the after structure. In order to model probabilities and random delays, the methods discussed in Section 4 are used. In state WAITSP, the host sends K probes before moving to the WAITSG state. In this state, the host sends two more ARPs before moving to the USE state and using the selected IP address. If, while in WAITSG, the host receives a packet with the same IP address, the host moves to RESPOND. For further details refer to [25].

The model for the network is shown in Figure 23. If the network is in the IDLE state, it moves to the NET_SEND state after a delay of 0 or 1 second and probabilistically selects the value 0 or 1 for the IP address sent by one of the hosts of the network, denoted by *ip*.

Figure 23 shows the model for time. This is a simplified model that only advances *time* by one. At any time, the action $time := time+1$ is chosen nondeterministically among other active UML-B events.

5.3 Translating the Model to PRISM

Using the method presented in Section 4.7, the created UML-B model was translated to a PRISM model. The resulting model is an MDP with three modules, namely host, network and time. A number of constants have been defined to represent different states. In each module, a variable holds the current state. Different transitions start with a guard checking the current state.

The resulting PRISM model can be used to verify a number of probabilistic properties about our model. For instance, the property saying "the maximum probability that the host finally chooses value 2 for *iph* and moves to state USE" is expressed in PCTL as $P_{max}(\Diamond((host\_state = USE) \wedge iph = 2)))$.

6 CONCLUSION AND FUTURE WORK

We have added abilities for modeling probabilistic and random systems in UML-B. A number of new structures have been added to the graphical syntax of UML-B, and

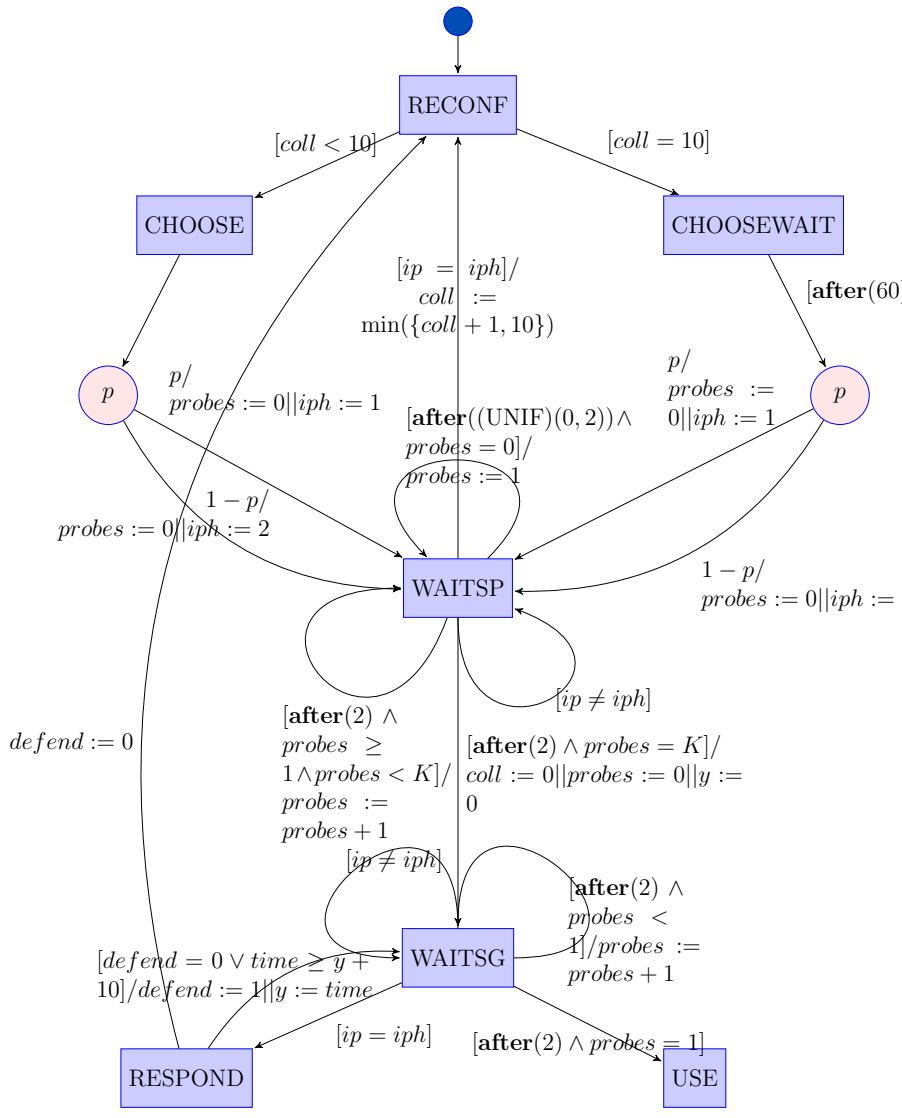

Figure 22. The UML-B state-machine model for the host component of the Zeroconf protocol

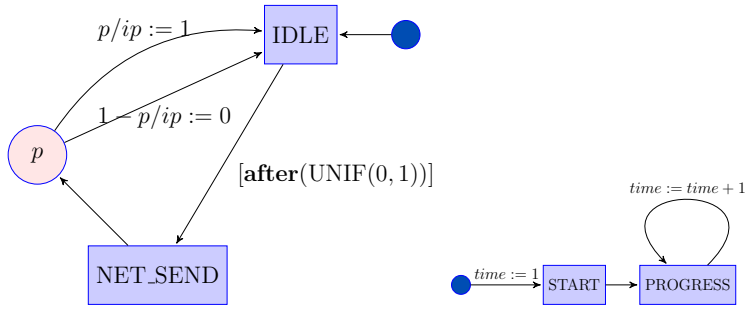

Figure 23. The UML-B state-machine model for the network and time components of the Zeroconf protocol

their semantics has been defined in Event-B. In addition, a method for translating UML-B models into PRISM language has been presented in order to perform quantitative and probabilistic model checking. To show the applicability of the proposed method, a case study on Zeroconf protocol was presented.

In future work, to increase the mathematical rigor of the proposed extensions, we would like to introduce rules and proof obligations for refinement of the proposed probabilistic structures. Furthermore, the proposed methods for translations and conversions need an automatic tool.

REFERENCES

[1] Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, 2005.

[2] Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010, doi: 10.1017/cbo9781139195881.

[3] Abrial, J.-R.—Butler, M.—Hallerstede, S.—Hoang, T. S.—Mehta, F.—Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer (STTT), Vol. 12, 2010, No. 6, pp. 447–466, doi: 10.1007/s10009-010-0145-y.

[4] Baier, C.—Katoen, J.-P.—Larsen, K. G.: Principles of Model Checking. MIT Press, 2008.

[5] Butler, M.—Falampin, J.: An Approach to Modelling and Refining Timing Properties in B. Refinement of Critical Systems (RCS), 2002.

[6] Cansell, D.—Méry, D.: The Event-B Modeling Method: Concepts and Case Studies. In: Bjørner, D., Henson, M. C. (Eds.): Logics of Specification Languages. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008, pp. 47–152.

[7] Cansell, D.—Méry, D.—Rehm, J.: Time Constraint Patterns for Event B Development. In: Julliand, J., Kouchnarenko, O. (Eds.): B 2007: Formal Specification and

Development in B (B 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4355, 2006, pp. 140–154, doi: 10.1007/11955757_13.

[8] Corin, R.—Den Hartog, J.: A Probabilistic Hoare-Style Logic for Game-Based Cryptographic Proofs. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (Eds.): Automata, Languages and Programming (ICALP 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4052, 2006, pp. 252–263, doi: 10.1007/11787006_22.

[9] Delahaye, B.—Larsen, K. G.—Legay, A.—Pedersen, M. L.—Wąsowski, A.: Decision Problems for Interval Markov Chains. In: Dediu, A. H., Inenaga, S., Martín-Vide, C. (Eds.): Language and Automata Theory and Applications (LATA 2011). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6638, 2011, pp. 274–285, doi: 10.1007/978-3-642-21254-3_21.

[10] Dotti, F. L.—Iliasov, A.—Ribeiro, L.—Romanovsky, A.: Modal Systems: Specification, Refinement and Realisation. In: Breitman, K., Cavalcanti, A. (Eds.): Formal Methods and Software Engineering (ICFEM 2009). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5885, 2009, pp. 601–619, doi: 10.1007/978-3-642-10373-5_31.

[11] Berthomieu, B.—Bodeveix, J.-P.—Farail, P.—Filali, M.—Garavel, H.—Gaufillet, P.—Lang, F.—Vernadat, F.: Fiacre: An Intermediate Language for Model Verification in the TOPCASED Environment. European Congress on Embedded Real-Time Software (ERTS), 2008.

[12] Fecher, H.—Leucker, M.—Wolf, V.: Don't Know in Probabilistic Systems. In: Valmari, A. (Ed.): Model Checking of Software (SPIN 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3925, 2006, pp. 71–88, doi: 10.1007/11691617_5.

[13] Grimaldi, R. P.: Discrete and Combinatorial Mathematics: An Applied Introduction. 5th ed. Pearson, 2003.

[14] Güdemann, M.—Lipaczewski, M.—Struck, S.—Ortmeier, F.: Unifying Probabilistic and Traditional Formal Model Based Analysis. 8th Dagstuhl-Workshop MBEES 2012 – Model-Based Development of Embedded Systems, 2012.

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

[16] Hansson, H.—Jonsson, B.: A Logic for Reasoning about Time and Reliability. Formal Aspects of Computing, Vol. 6, 1994, No. 5, pp. 512–535, doi: 10.1007/bf01211866.

[17] Hasan, O.—Tahar, S.: Formal Probabilistic Analysis: A Higher-Order Logic Based Approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (Eds.): Abstract State Machines, Alloy, B and Z (ABZ 2010). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5977, 2010, pp. 2–19, doi: 10.1007/978-3-642-11811-1_2.

[18] Hoang, T. S.: The Development of a Probabilistic B-Method and a Supporting Toolkit. Ph.D. thesis, The University of New South Wales, 2005.

[19] Hurd, J.: Formal Verification of Probabilistic Algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, 2003.

[20] Jansen, D. N.—Hermanns, H.—Katoen, J.-P.: A Probabilistic Extension of UML Statecharts. In: Damm, W., Olderog, E. R. (Eds.): Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2469, 2002, pp. 355–374, doi: 10.1007/3-540-45739-9_21.

[21] Jansen, D. N.—Hermanns, H.—Katoen, J.-P.: A QoS-Oriented Extension of UML Statecharts. In: Stevens, P., Whittle, J., Booch, G. (Eds.): "UML" 2003 – The Unified Modeling Language. Modeling Languages and Applications (UML 2003). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2863, 2003, pp. 76–91, doi: 10.1007/978-3-540-45221-8_7.

[22] Kwiatkowska, M.—Norman, G.—Parker, D.: Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In: Katoen, J. P., Stevens, P. (Eds.): Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2280, 2002, pp. 52–66, doi: 10.1007/3-540-46002-0_5.

[23] Kwiatkowska, M.—Norman, G.—Parker, D.: Advances and Challenges of Probabilistic Model Checking. 2010 48$^{\text{th}}$ Annual Allerton Conference on Communication, Control, and Computing (Allerton), IEEE, 2010, pp. 1691–1698, doi: 10.1109/allerton.2010.5707120.

[24] Kwiatkowska, M.—Norman, G.—Parker, D.: Prism 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (Eds.): Computer Aided Verification (CAV 2011). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6806, 2011, pp. 585–591, doi: 10.1007/978-3-642-22110-1_47.

[25] Kwiatkowska, M.—Norman, G.—Parker, D.—Sproston, J.: Performance Analysis of Probabilistic Timed Automata Using Digital Clocks. Formal Methods in System Design, Vol. 29, 2006, No. 1, pp. 33–78, doi: 10.1007/978-3-540-40903-8_9.

[26] Lopatkin, I.—Iliasov, A.—Romanovsky, A.—Prokhorova, Y.—Troubitsyna, E.: Patterns for Representing FMEA in Formal Specification of Control Systems. 2011 IEEE 13$^{\text{th}}$ International Symposium on High-Assurance Systems Engineering (HASE), IEEE, 2011, pp. 146–151, doi: 10.1109/hase.2011.10.

[27] Reggio, G.—Wieringa, R. J.: Thirty One Problems in the Semantics of UML 1.3 Dynamics. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99), 1999.

[28] Rehm, J.: A Method to Refine Time Constraints in Event B Framework. Automatic Verification of Critical Systems (AVoCS 2006), 2006, pp. 173–177.

[29] Rumbaugh, J.—Jacobson, I.—Booch, G.: Unified Modeling Language Reference Manual. Pearson Higher Education, 2004.

[30] Said, M. Y.—Butler, M.—Snook, C.: Class and State Machine Refinement in UML-B. Proceedings of Workshop on Integration of Model-Based Formal Methods and Tools (associated with IFM 2009), 2009.

[31] Snook, C.—Butler, M.: UML-B and Event-B: An Integration of Languages and Tools. Proceedings of the IASTED International Conference on Software Engineering, 2008, pp. 336–341, doi: 10.1145/1125808.1125811.

[32] Tarasyuk, A.—Troubitsyna, E.—Laibinis, L.: Towards Probabilistic Modelling in Event-B. In: Méry, D., Merz, S. (Eds.): Integrated Formal Methods (IFM 2010). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6396, 2010, pp. 275–289, doi: 10.1007/978-3-642-16265-7_20.

[33] Tarasyuk, A.—Troubitsyna, E.—Laibinis, L.: Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (Eds.): Integrated Formal Methods, 2012 (IFM 2012). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7321, pp. 237–252, doi: 10.1007/978-3-642-30729-4_17.

[34] Tarasyuk, A.—Troubitsyna, E.—Laibinis, L.: Quantitative Reasoning about Dependability in Event-B: Probabilistic Model Checking Approach. In: Petre, L., Sere, K., Troubitsyna, E. (Eds.): Dependability and Computer Engineering: Concepts for Software-Intensive Systems. IGI Global, 2012, pp. 459–472, doi: 10.4018/978-1-60960-747-0.ch019.

Mohammad Nosrati is a Ph.D. student in software engineering at Shahid Beheshti University, Tehran, Iran. He holds a master's degree in software engineering from Shahid Beheshti University. His research interests are software testing, formal methods, machine learning and image processing.

Hassan Haghighi received his Ph.D. in computer engineering from Sharif University of Technology. He is Associate Professor at the Faculty of Computer Science and Engineering in Shahid Beheshti University, Tehran, Iran. His research focus is on software testing, formal methods, and software architecture.