Science of Computer Programming 128 (2016) 22–50

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

PTRebeca: Modeling and analysis of distributed and asynchronous systems

Ali Jafari [a,*], Ehsan Khamespanah [a,b], Marjan Sirjani [a,c], Holger Hermanns [d], Matteo Cimini [e]

[a] Reykjavik University, School of Computer Science and CRESS, Iceland
[b] University of Tehran, School of ECE, Islamic Republic of Iran
[c] Mälardalen University, Embedded Systems, Sweden
[d] University of Saarland, School of Computer Science, Germany
[e] Indiana University, Bloomington, Center for Research in Extreme Scale Technologies, United States

ARTICLE INFO

Article history:
Received 16 May 2015
Received in revised form 12 March 2016
Accepted 14 March 2016
Available online 31 March 2016

Keywords:
Probabilistic Timed Automata
Timed Markov Decision Process
IMCA model checker
Probabilistic Timed Rebeca
Model checking
Performance analysis

ABSTRACT

Distributed systems exhibit probabilistic and non-deterministic behaviors and may have time constraints. Probabilistic Timed Rebeca (PTRebeca) is introduced as a timed and probabilistic actor-based language for modeling distributed real-time systems with asynchronous message passing. The semantics of PTRebeca is a Timed Markov Decision Process. In this paper, we provide SOS rules for PTRebeca, introduce a new tool-set and describe the corresponding mappings. The tool-set automatically generates a Markov Automaton from a PTRebeca model in the form of the input language of the Interactive Markov Chain Analyzer (IMCA). The IMCA can be used as a back-end model checker for performance analysis of PTRebeca models against expected reachability and probabilistic reachability properties. Comparing to the existing tool-set, proposed in the conference paper, we now have the ability of analyzing significantly larger models, and we also can add different rewards to the model. We show the applicability of our approach and efficiency of our tool by analyzing a Network on Chip architecture as a real-world case study.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Our modern society more and more relies on software systems that are distributed and consist of concurrently executing components which communicate asynchronously over networks. Modeling and analyzing these complex systems is a non-trivial and intricate task. There is thus a need for modeling languages that match well with computational models of such systems, and are supported by tools for analyzing performance and dependability aspects of these systems.

A well-established paradigm for modeling the functional behavior of distributed and asynchronous systems is the actor model. Actor model is introduced by Hewitt as an agent-based language for programming distributed systems [1], and is later developed by Agha [2–4] into a concurrent object-based model. Actors are distributed, autonomous objects that interact via asynchronous message passing. Building on an event-driven and message-based foundation, actors provide scalability and are easy-to-grasp concurrency models. With the growth of cloud computing, web services, networks of embedded computers, and multicore architectures, programming using the actor model has become increasingly relevant.

* Corresponding author. Tel.: +354 776 6603, +98 936 723 6840.
E-mail address: ali11@ru.is (A. Jafari).

http://dx.doi.org/10.1016/j.scico.2016.03.004
0167-6423/© 2016 Elsevier B.V. All rights reserved.

Popular actor programming languages and frameworks include Erlang [5] and the Scala/Akka family [6]. Many projects in industry, e.g. at Google (like DART) and Microsoft (like Asynchronous Agents Library), have explored the actor model. Large applications such as Twitter's message queuing, image processing in MS Visual Studio 2010, as well as the Vendetta game engine [7] have been designed on the basis of this model.

Rebeca [8,9] is an actor-based modeling language designed to enable formal verification of actor models. It hence bridges the gap between formal methods and software engineering. Using Rebeca we can deploy a model-driven development approach with a formal basis. Rebeca is supported by formal verification tools and techniques which are based on the formal semantics of the language [10]. An extension of Rebeca [11] has been proposed to provide the ability of modeling and verification of distributed systems with real-time constraints. In this context, Floating Time Transition System (FTTS) is introduced to significantly reduce the state space generated when model checking Timed Rebeca (TRebeca) models [12]. Checks for absence of deadlock freedom and schedulability analysis of TRebeca models can be performed using FTTS.

Since its introduction, TRebeca has been used in different areas. Examples include the analysis of different routing algorithms and scheduling policies in NoC (Network on Chip) designs [13,14], as well as schedulability analysis of distributed real-time sensor network applications [15], more specifically a real-time continuous sensing application for structural health monitoring in [16]. In analyzing the above mentioned applications, we observed the need for modeling probabilistic behavior. In an earlier work, pRebeca has beem proposed as an extension of Rebeca to model probabilistic systems [17]. However, pRebeca does not support the timing features.

In [18], we proposed Probabilistic Timed Rebeca (PTRebeca) which benefits from and integrates modeling features of TRebeca and pRebeca, combining their respective syntax. This aims at enhancing our modeling ability in order to cover more properties, so as to support performance evaluation of probabilistic real-time actors. To keep the consistency, we designed the syntax of PTRebeca as a combination of TRebeca and pRebeca. Still, as to be expected, existing formal semantics and supporting tools are not directly applicable to PTRebeca. Consequently, Timed Markov Decision Processes (TMDP) are used as the semantics of PTRebeca, to support timing, probabilistic, and non-deterministic features. TMDP can be regarded as the discrete-time semantics of probabilistic timed automata (PTA) [19], or as variation of interactive probabilistic chains [20]. For performance evaluation of PTRebeca models we employ probabilistic model checking, for both functional verification and performance evaluation. The benefits of combining performance evaluation with functional verification is elaborated upon in [21].

This paper is an extended version of the paper presented at AVoCS conference [18]. In this paper, we provide Structural Operational Semantics (SOS rules) for the PTRebeca language in the style of Plotkin [22]. The tool developed in [18] uses PRISM [23] as a back-end model checker while in this paper we instead use the IMCA (Interactive Markov Chain Analyzer) tool [24]. In our conference paper [18], we mapped a PTRebeca model to a single, flat Markov Decision Process (MDP) module. This approach is consistent with the semantics of PTRebeca. But as the entire PTRebeca model is mapped into a single MDP module, the module becomes prohibitively large. As a consequence, the analysis time is very high. To overcome this problem, we used the explicit engine of PRISM which works with an intermediate transition matrix representation. This allows us to analyze larger models, but PRISM does not provide full support for this format. Therefore, we were only able to use it for the analysis of probabilistic reachability properties, but not for the expected reachability ones.

An alternative way, detailed in the present paper, maps each component (reactive object) in a PTRebeca model to a Probabilistic Timed Automaton (PTA). Then the parallel composition of PTA (of all components) represents the behavior of the PTRebeca model. We call this approach the parallel composition approach and we will show this approach in Section 4.3. In Section 4.3, we will also compare the parallel composition approach with TMDP semantics. We will demonstrate that the state space generated via the TMDP semantics (proposed in the conference paper) is much smaller than the state space generated from the parallel composition approach. So, although this way we can use the full power of PRISM the state space explosion problem occurs very quickly.

To deal with the restriction of the previous approaches, as explained above, we turned to the IMCA (Interactive Markov Chain Analyzer) model checker [24], and we used it as the back-end model checker for the analysis of PTRebeca models. IMCA accepts Markov Automata (MA) [25] and Interactive Markov Chain (IMC) [26] models. An MA-transition is either labelled with an action (probabilistic transition), or with a positive real number representing the rate of a negative exponential distribution (Markovian transition). An action (probabilistic) transition leads to a discrete probability distribution over states. MA can thus model action transitions as in labelled transition systems, probabilistic branching, as well as delays that are governed by exponential distributions [27].

In order to use IMCA as a back-end model checker, we need to convert the TMDP of an underlying PTRebeca model to its corresponding MA. There are two types of transitions in a TMDP: action (probabilistic) transition, leading to a discrete probability distribution over states, and delay transition, carrying a positive integer value. Probabilistic transitions in TMDP are mapped directly to probabilistic transitions in MA. For the transition rate in the MA, the inverse of the integer value of a delay transition in TMDP is considered as the rate of the corresponding transition in the MA. This conversion is proved to be correct for checking expectation properties, and not for time-bounded reachability property. Therefore, we can use our previously developed tools to generate the TMDP of our models automatically. The obtained TMDP is converted to its Markov Automaton which is then the input to IMCA. Using this approach, we are able to evaluate the performance of our models against probabilistic reachability, expected reward reachability, and expected time reachability properties. In Section 5, we mathematically prove that the values of expected time reachability in TMDP and its corresponding MA are identical.

$$Model ::= Class^{*}\ Main$$
$$Main ::= \textbf{main}\ \{\ InstanceDcl^{*}\ \}$$
$$InstanceDcl ::= className\ rebecName(\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*});$$
$$Class ::= \textbf{reactiveclass}\ className\ \{\ KnownRebecs\ Vars\ MsgSrv^{*}\ \}$$
$$KnownRebecs ::= \textbf{knownrebecs}\ \{\ VarDcl^{*}\ \}$$
$$Vars ::= \textbf{statevars}\ \{\ VarDcl^{*}\ \}$$
$$VarDcl ::= type\ \langle v\rangle^{+};$$
$$MsgSrv ::= \textbf{msgsrv}\ methodName(\langle type\ v\rangle^{*})\ \{\ Stmt^{*}\ \}$$
$$Stmt ::= v = e;\ |\ v = ?(e\langle, e\rangle^{+});\ |\ Call;\ |\ if\ (e)\ \{\ Stmt^{*}\ \}\ [else\ \{\ Stmt^{*}\ \}]$$
$$Call ::= rebecName.methodName(\langle e\rangle^{*})$$

(a) Abstract Syntax of Rebeca

$$Stmt ::= v = e;\ |\ v = ?(e\langle, e\rangle^{+});\ |\ Call;\ |\ if\ (e)\ \{\ Stmt^{*}\ \}\ [else\ \{\ Stmt^{*}\ \}]\ |\ \textbf{delay}(v);$$
$$Call ::= rebecName.MethodName(\langle e\rangle^{*})\ [\textbf{after}(v)]\ [\textbf{deadline}(v)]$$

(b) Changes in the syntax of Rebeca to build TRebeca

$$Stmt ::= v = e;\ |\ v = ?(e\langle, e\rangle^{+});\ |\ Call;\ |\ if\ (e)\ \{\ Stmt^{*}\ \}\ [else\ \{\ Stmt^{*}\ \}]\ |$$
$$delay(v);\ |\ \mathbf{v} = ?(\mathbf{e_p} : \mathbf{e}\langle, \mathbf{e_p} : \mathbf{e}\rangle^{+});$$

(c) Changes in the syntax of TRebeca to build PTRebeca

Fig. 1. (a) Abstract syntax of Rebeca. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition at least once, superscript ∗ for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. The symbol ? shows non-deterministic choice. Identifiers *className*, *rebecName*, *methodName*, *v*, *literal*, and *type* denote class name, rebec name, method name, variable, literal, and type, respectively; and *e* denotes an (arithmetic, boolean or nondetermistic choice) expression. (b) Changes for Timed Rebeca. The timing primitives are added to *Stmt* and *Call* statements. The value of variable *v* in timing primitives is a natural number. (c) Changes for Probabilistic Timed Rebeca. The probabilistic assignment is added to *Stmt*. The expression e_{p_i} denotes an expression which returns probability. The symbol ? shows either non-deterministic assignment or probabilistic assignment.

The main contributions of this paper in comparison with the conference paper are as follows:

- Semantics: We present the SOS semantics of PTRebeca language. This provides a formal presentation that is approachable for more researchers, and is the reference for any implementation effort.
- Analysis: We use Interactive Markov Chain Analyzer (IMCA) as a back-end model checker for PTRebeca, and we use probabilistic model checking algorithms developed for Markov Automata for the analysis of probabilistic timed properties. We are able to check probabilistic reachability, long-run average, expected reward reachability, and expected time reachability properties for PTRebeca models.
- Implementation: We use our tool developed in [18] to generate the TMDP of a PTRebeca models automatically. The generated TMDP is in the form of an XML file. The XML file is converted to the input language of IMCA.
- Case Study: We present an analysis of a Network on Chip (NoC) architecture to demonstrate the feasibility of our approach for a real-world case study.

2. Probabilistic Timed Rebeca

In this section, we introduce Probabilistic Timed Rebeca (PTRebeca). We first present Rebeca [8,9], and then we show its extension with timing features to build TRebeca [11]. Finally we discuss how probability and time are added to Rebeca to build PTRebeca, enabling the modeling of probabilistic timed behaviors. The syntax of PTRebeca is presented in Fig. 1. We model a simple ticket service example to explain the modeling features of PTRebeca.

Rebeca Rebeca is an actor-based modelling language with formal semantics that is supported by model checking tools. A Rebeca model consists of the definition of reactive classes and the instantiation part which is called main. The main part

```
reactiveclass TicketService {
  knownrebecs {
    Agent a;
  }
  statevars {
    int issueDelay;
  }
  msgsrv initial(int myDelay) {
    issueDelay = myDelay;
  }
  msgsrv requestTicket() {
    delay(issueDelay);
    a.ticketIssued(1);
  }
}

reactiveclass Agent {
  knownrebecs {
    TicketService ts;
    Customer c;
  }
  msgsrv requestTicket() {
    a = ?(4,5);
    ts.requestTicket() deadline(a);
  }
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}

reactiveclass Customer {
  knownrebecs {
    Agent a;
  }
  msgsrv initial() {
    self.try();
  }
  msgsrv try() {
    a.requestTicket();
  }
  msgsrv ticketIssued(byte id) {
    b = ?(0.75:30,0.25:10);
    self.try() after(b);
  }
}

main {
  Agent a(ts, c):();
  TicketService ts(a):(3);
  Customer c(a):();
}
```

Fig. 2. The PTRebeca model of the ticket service system.

defines instances of reactive classes, called *rebecs*. The behavior of the instances of a reactive class is determined by its message servers. The internal state of a reactive class is represented by the valuation of its state variables.

In Rebeca, computation is event-driven, where messages can be seen as events. Each rebec takes a message from its message queue and executes the corresponding message server. Execution of a message server body takes place atomically (non-preemptively). Communication takes place by asynchronous message passing, which is non-blocking for both sender and receiver. The sender rebec sends a message to the receiver rebec and continues its work. The message is put in the message queue of the receiver. The message stays in the queue until the receiver takes and serves it. Although in theory we define no boundary for the queue length, in the supporting tools we always have a queue length that is defined by the user. The operational semantics of Rebeca is introduced in [9], to which we refer for more details. The syntax of Rebeca is represented in Fig. 1.

Timed Rebeca TRebeca was introduced as an extension of the Rebeca language to model real-time reactive systems. Just as with Rebeca, the formal semantics of TRebeca is defined using Structural Operational Semantics (SOS) [11]. In a TRebeca model, each rebec has its own local time, which can be considered as synchronized distributed clocks. Methods are executed atomically, but passing of time can be modeled while executing a method. Instead of a message queue for each rebec, there exists a bag containing sent messages together with the timing information, which are used to process the message in the intended order in time. Different timing primitives are added to Rebeca syntax to cover a variety of timing features that a modeler might need to address in a message-based, asynchronous and distributed setting. These timing primitives are *delay*, *deadline* and *after*, and are detailed below. The syntax of timing primitives is shown in Fig. 1.

Delay: *delay(t)* increases the value of the local time of the respective rebec by the amount of t.

Deadline: *r.m() deadline(t)*, after t units of time the message m of rebec r is not valid any more and is to be purged from the bag.

After: *r.m() after(t)*, the message cannot be taken from the bag before t time units have passed.

Upon sending a message, it is put in the message bag of the receiver, together with its associated time tag and *deadline* tag. The time tag of a message is the value of the local time of the sender when the message was sent, unless the message is augmented with an *after* primitive. In this case the value of the argument of *after* is added to the value of local time of the sender to build the time tag.

Probabilistic Timed Rebeca PTRebeca language supports modeling and verification of real-time systems with probabilistic behaviors. Syntax of PTRebeca is a combination of pRebeca and TRebeca. In Fig. 1, we show the extension made to the syntax of TRebeca to build PTRebeca [18]. In a probabilistic assignment, a value is assigned to the variable with the specified probability. In the probabilistic assignment, $e_{p_1} \ldots e_{p_n}$ are real values between 0 and 1, and sum up to 1. Notably, by using probabilistic assignments, the value of the timing constructs (delay, after, and deadline) can also become probabilistic.

Different probabilistic behaviors can be modeled using PTRebeca language, depending on the system under study. We present a simple ticket service system in Fig. 2 to illustrate how PTRebeca can be applied. Each entity in the system is mapped to an actor in the PTRebeca model. The ticket service model includes a customer, a ticket service, and an agent. The customer `c` sends a ticket request by sending the message `requestTicket()` to the agent `a` (line 39). The agent forwards the request to the ticket service `ts` by sending the message `requestTicket()` (line 24). The message `requestTicket()` has a deadline which is set non-deterministically (line 23). The ticket service issues a ticket and replies to the agent request by sending the message `ticketIssued()` (line 13). The agent sends the message `ticketIssued` to the customer to complete the issuing process (line 27). The customer sends a new request after 10 or 30 units of time with probabilities of 0.25 or 0.75, respectively (lines 42 and 43).

3. Structural operational semantics of PTRebeca

We present the TMDP of a PTRebeca model as a tuple $(S, s_0, Act, \rightarrow, \Rightarrow)$ where S is a set of states, s_0 is the initial state, Act is a set of actions which consists of τ, signatures of all the messages, and $\mathbb{N}$. The union of scheduler and msg-fetcher transitions is $\rightarrow$ (probabilistic transitions) and the set of time-progress transitions (delay transitions) is $\Rightarrow$. Scheduler transitions, msg-fetcher transitions, and time-progress transitions are defined in the following paragraphs.

In this section we provide an SOS semantics for PTRebeca in the style of Plotkin [22]. The behavior of PTRebeca programs is described by means of transition relations that govern the step-by-step evolution of the system.

The states of the system are tuples (Env, B, T), where Env is a finite set of environments, B is a bag of messages and T is a natural number that represents the current time of the system. For each rebec A of the system, Env contains an environment σ_A that is a function that maps variables to their values. Basically, σ_A is the private store of the rebec A. Environments contain four special-purpose variables: *self*, which contains the name of the rebec, *pc*, which stands for *program counter* and contains the code that is currently being executed, *rt*, which stores the resume time of the rebec, and *sender*, which stores the name of the rebec that invoked the method that is currently being executed. Whenever a rebec A of a reactive class O is created, an environment σ_A is assumed to be initialized. In particular, the code of each message server m of O is loaded in $\sigma_A(m)$ as a *null*-terminated list of statements.

The bag contains an unordered collection of messages of the form

$$(A_i, m(\overline{v}), A_j, TT, DL).$$

Intuitively, such a tuple says that at time TT the sender A_j sent the message to the rebec A_i asking it to execute its method m with actual parameters $\overline{v}$. Moreover this message expires at time DL.

We denote by $Tmsg$ the set of all the possible messages. Given a message $msg \in Tmsg$, $ar(msg)$ denotes the arrival time of the message msg, that is, TT in the tuple above. At each step, the system progresses thanks to one of three transition relations: $\xrightarrow{\tau}$, $\xrightarrow{msg}$ with $msg \in Tmsg$, and $\xrightarrow{n}$ with $n \in \mathbb{N}$. Any of these transitions evolves a state (Env, B, T) into a probability distribution pv that assigns probability values to states. For readability, we represent pv as a set of mappings, for instance the probability distribution $\{(Env, B, T) \mapsto 1\}$ maps the state (Env, B, T) to probability 1. Whenever more cases need to be specified for pv, they will be embraced in a large bracket and the mappings involved in the distribution will be graphically clear. States that are not mentioned in pv are assumed to be mapped to probability 0.

As a convention, whenever we single out an element from a set, as in the sets $\sigma_A \cup Env$ and $msg \cup B$, we will assume that $\sigma_A \notin Env$ and $msg \notin B$. Moreover, we will use the notation $\sigma[x = e]$ to denote the mapping σ where x is redefined in order to map x to e.

The transitions $\xrightarrow{\tau}$, $\xrightarrow{msg}$, $\xrightarrow{n}$, are formally defined by the following rules.

$$(scheduler)\ \frac{\sigma_A(pc) = s \quad s \neq null \quad \sigma_A(rt) = T \quad (s, \sigma_A[pc = null], Env, B, T) \xrightarrow{s} pv}{(\{\sigma_A\} \cup Env, B, T) \xrightarrow{\tau} pv}$$

$$(msg\text{-}fetcher)\ \frac{\sigma_{A_i}(pc) = null \quad TT \leq T \leq DL \quad \sigma'_{A_i} = \sigma_{A_i}[pc = \sigma_{A_i}(m), \sigma_{A_i}(rt) = T, \overline{arg} = \overline{v}, sender = A_j] \quad msg = \{(A_i, m(\overline{v}), A_j, TT, DL)\}}{(\{\sigma_{A_i}\} \cup Env, msg \cup B, T) \xrightarrow{msg} \{(\{\sigma'_{A_i}\} \cup Env, B, T) \mapsto 1\}}$$

$$(time\text{-}progress)\ \frac{(Env, B, T) \not\xrightarrow{\tau} \quad (Env, B, T) \not\xrightarrow{msg} \quad n_1 = \min_{\sigma \in Env}\{\sigma(rt)\} \quad n_2 = \min_{msg \in B}\{ar(msg)\} \quad T' = \min\{n_1, n_2\} \quad n = T' - T}{(Env, B, T) \xrightarrow{n} \{(Env, B, T') \mapsto 1\}}$$

The *(scheduler)* rule is responsible for picking a rebec and executing its pending statements. This rule choses a rebec non-deterministically among those for which the program counter still contains statements to execute (conditions $\sigma_A(pc) = s$ and $s \neq null$). Moreover, a rebec is eligible for being chosen only as long as its resume time coincides with the current time (condition $\sigma_A(rt) = T$). Rebecs that have previously executed a delay statement might have a resume time ahead of the current time and in that case they would not be chosen. The execution of the statement is performed with the auxiliary transition relation $\xrightarrow{s}$, described later in detail. Such a transition is responsible for the execution of one statements from the list of statements s, the first one. It is to notice that the program counter is consumed immediately before the call to statement execution (indeed, the environment $\sigma_A[pc = null]$ is passed). However, s might contain more than one statement and, moreover, statements such as if-then-else might imply the execution of further statements (one of the branches). As we will see later, these scenarios are taken care by the transition $\xrightarrow{s}$. This transition will be responsible to feed the program counter back with the possible leftover statements to be executed.

In our semantics, the transition $\xrightarrow{s}$ returns the probability distribution pv for the next state of the system. The *(scheduler)* rule simply uses pv for the transition. In order to let the system progress after a step, we implicitly assume picking a state of pv according to its probability.

The (*msg-fetcher*) rule allows the system to progress by picking up a message from the bag and initialize the rebec receiver of the message for the execution of such message. This rule is applicable for a rebec only as long as this latter is not in the phase of executing any other message (condition $\sigma_{A_i}(pc) = null$). Moreover, the message can be picked only so long it is not too soon for fetching it nor too late (condition $TT \leq T \leq DL$). The rule prepare the rebec A_i for the execution of the message m in the following way.

- The method body of m is looked up from the environment of A_i and loaded in the program counter.
- The resume time for A_i is set to the current time of the system, stating that is to be executed immediately.
- The variable *sender* is set to the sender of the message.
- In executing the method m, the formal parameters $\overline{arg}$ are set to the values of the actual parameters $\overline{v}$. Methods of arity k are indeed supposed to have $arg_1, arg_2, \ldots, arg_k$ as formal parameters. This is without loss of generality since such a change of variable names can be performed in a pre-processing step for any program.

The (*time-progress*) rule is responsible for letting time pass for some units of time. This happens when the system has no eligible statements of rebecs to execute and no eligible messages that can be picked from the bag (eligible w.r.t. the conditions of rules *(scheduler)* and (*msg-fetcher*), respectively). In such a scenario, the system lets the time pass for the minumum amount of time necessary to enable the rebec whose resume time is the closest to the current time ($\min_{\sigma \in Env}\{\sigma(rt)\}$) or to enable the fetch of a message whose picking time is the closest to the current time ($\min_{msg \in B}\{ar(msg)\}$).

Fig. 3 shows the SOS rules for the execution of statements in PTRebeca. The transition relation $\xrightarrow{s}$ defines the execution of statements. The general form of this type of transition is $(s, \sigma, Env, B, T) \xrightarrow{s} pv$, where s is a list of statements or a single statement[1], σ is the local environment where to evaluate statements, and Env, B, and T are the components of the system state. The step evolves into a probability distribution pv. Carrying the global bag B is important because new messages may be added to it with the execution of a statement. The global set of environments Env is also required because *new* statements create new rebecs and may therefore add new environments to it. In the semantics, σ is separated from Env and passed as a parameter for the sake of clarity and also because nearly every rule needs to readily affect it. A few statements make use of the current time T which is therefore promoted as parameter as well.

For all rules with the exception of *(prob)*, the result of the step $\xrightarrow{s}$ first creates a new state that has a new environment, a new bag and the current time, then this state is injected into a probability distribution function where it has probability 1. The simple and non-deterministic assignment statements are handled by rules (*assign*) and (*non-det*), respectively, and it is easy to see that they follow the schema just depicted: their semantics coincides indeed with the standard one, modulo our injection to a probability distribution.

Rule *(prob)* handles the probabilistic assignment $x =? \ p_1 : e_1 \oplus p_2 : e_2 \ldots \oplus p_n : e_n$. In such a case, n states are created that differ only in the assignment to the variable x for the local environment being used. These states are injected into a probability distribution pv that maps them to the probabilities p_1, p_2, ..., p_n. The rules for the timing primitives deserve some explanation.

- Rule *msg* describes the effect of method invocation statements. For the sake of brevity, we limit ourselves to presenting the rule for method invocation statements that involve both the *after* and *deadline* keywords. The semantics of instances of that statement without those keywords can be handled as special cases of that rule by setting the argument of *after* to zero and that of *deadline* to $+\infty$, meaning that the message never expires. Method invocation statements put a new message in the bag, taking care of properly setting its fields. In particular the arrival time for the message is the current time T plus the number d that is the parameter of the *after* keyword.
- Delay statements change the resume time of the rebec to $T + d$, where d is the parameter of the *delay* keyword.

[1] We overload $\xrightarrow{s}$ for lists of statements in rule ($stmts^*$). We prefer this presentation rather than splitting $\xrightarrow{s}$ into two relations or splitting the scheduler into two parts.

$$(msg)\ \frac{pv = (\sigma \cup Env, \{(\sigma(varname), m(eval(\overline{v}, \sigma)), \sigma(self), T + d, T + DL)\} \cup B, T) \mapsto 1}{(varname.m(\overline{v})\ after(d)\ deadline(DL), \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(delay)\ \frac{pv = (\sigma[rt = T + d] \cup Env, B, T) \mapsto 1}{(delay(d), \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(assign)\ \frac{pv = (\sigma[x = eval(e, \sigma)] \cup Env, B, T) \mapsto 1}{(x = e, \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(non\text{-}det)\ \frac{pv = (\sigma[x = eval(e_i, \sigma)] \cup Env, B, T) \mapsto 1 \qquad (\text{with } 1 \le i \le n)}{(x =?\ e_1 \oplus e_2 \ldots \oplus e_n, \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(prob)\ \frac{pv = \begin{cases} (\sigma[x = eval(e_1, \sigma)] \cup Env, B, T) \mapsto p_1 \\ (\sigma[x = eval(e_2, \sigma)] \cup Env, B, T) \mapsto p_2 \\ \qquad \ldots \\ (\sigma[x = eval(e_n, \sigma)] \cup Env, B, T) \mapsto p_n \end{cases}}{(x =?\ p_1 : e_1 \oplus p_2 : e_2 \ldots \oplus p_n : e_n, \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(create)\ \frac{\begin{array}{l} \sigma_A = initialEnviroment(O) \qquad \text{with } A \text{ fresh in } \sigma \cup Env \\ pv = \Big(\sigma[varname = A] \cup \{\sigma_A[self = A, pc = null]\} \cup Env, \\ \{(A, initial(eval(\overline{v}, \sigma)), \sigma(self)), T, +\infty)\} \cup B, \quad T\Big) \mapsto 1 \end{array}}{(varname = new\ O(\overline{v}), \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(cond_1)\ \frac{eval(e, \sigma) = true \quad pv = (\sigma[pc = s_1] \bigcup Env, B, T) \mapsto 1}{(\text{if } (e) \text{ then } s_1 \text{ else } s_2, \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(cond_2)\ \frac{eval(e, \sigma) = false \quad pv = (\sigma[pc = s_2] \bigcup Env, B, T) \mapsto 1}{(\text{if } (e) \text{ then } s_1 \text{ else } s_2, \sigma, Env, B, T) \xrightarrow{s} pv}$$

$$(stmts^*)\ \frac{(s_1, \sigma, Env, B, T) \xrightarrow{\tau,n} pv \quad inject(rest, \sigma(self), pv) = pv'}{(s_1 :: rest, \sigma, Env, B, T) \xrightarrow{s} pv'}$$

where the function *inject(rest,ref,pv)* is defined below:

$$\text{if } pv = \begin{cases} (\sigma \cup Env_1, B_1, T) \mapsto p_1 \\ (\sigma \cup Env_2, B_2, T) \mapsto p_2 \\ \qquad \ldots \\ (\sigma \cup Env_n, B_n, T) \mapsto p_n \end{cases} \quad \text{where } \sigma(self) = ref$$

$$\text{then } inject(rest,ref,pv) = \begin{cases} (\sigma[pc = \sigma(pc) :: rest] \cup Env_1, B_1, T) \mapsto p_1 \\ (\sigma[pc = \sigma(pc) :: rest] \cup Env_2, B_2, T) \mapsto p_2 \\ \qquad \ldots \\ (\sigma[pc = \sigma(pc) :: rest] \cup Env_n, B_n, T) \mapsto p_n \end{cases}$$

we assume that the append operation :: is such that $null :: rest = rest$.

Fig. 3. SOS rules for the execution of statements of PTRebeca.

The creation of new rebecs is handled by the rule *create*. Whenever a rebec must be created out of the reactive class O, we first pick a fresh name A that it is used to identify the newly created rebec. The name A is assigned to the variable *varname* of the sender. We assume a function *initialEnvironment*(O) that returns a new environment σ_A that is initialized depending on the specification of the rebec O, i.e. inspecting the body of the specification **reactiveclass** O .... In particular, the code of each message server m of O is loaded in $\sigma_A(m)$ as a *null*-terminated list of statements. Ultimately, a message is put in the bag in order to execute the *initial* method of the newly created rebec.

The reader should recall that the scheduler sets the program counter to *null* before executing a statement (passing $\sigma_A[pc = null]$ in rule *(scheduler)*). The statements that we have described so far have no continuation and simply leave the *pc* variable set to *null*.

A conditional statement *if* (e) *then* s_1 *else* s_2 is different in this respect because after evaluating the guard e the continuation is either s_1 or s_2. Rules $(cond_1)$ and $(cond_2)$ handle the execution of conditional statements and they take care of setting pc to s_1 or s_2 according to the evaluation of e.

Another rule that affects the pc variable is $(stmts^*)$. This rule handles the execution of the first statement of a list. After the first statement has been executed, it might return some continuation statements. We therefore need to put these latter statements in front, before evaluating the rest of the original list of statements (*rest*). However, the execution step $\xrightarrow{s}$ returns a probability distribution. We therefore use an auxiliary function in order to inject these statements in all of the possible states of such distribution. Precisely, the function *inject(rest,ref,pv)* seeks for the private store of the rebec *ref* in each of the states of the distribution pv and queues the statements *rest* in the program counter of those private stores.

4. Performance analysis of PTRebeca models using PRISM

In [18], we presented the Afra tool-set in order to generate the TMDP corresponding to a PTRebeca model. The resulting TMDP can be considered as the integral semantics of a probabilistic time automaton with one digital clock. As shown in [19], probabilistic reachability and expected reachability properties can be analyzed for PTA with digital clocks. Therefore, we are able to analyze PTRebeca models against these two important performance measures.

To be able to implement PTA with digital clocks in PRISM, there are some restrictions in [19]. It does not allow atomic constraints of the form $x > c$ or $x < c$ (closed) or $x - y \sim c$ (diagonal free), where $c \in \mathbb{N}$, $\sim \in \{\leq, =, \geq\}$, x and y are different clocks. In this way, "digital clocks" engine of PRISM is used, and PTA modules are defined. In another way, we can consider MDP modules with integer-valued variables representing clocks in order to define PTA with digital clocks in PRISM. Using this approach, there is no need to satisfy the above restrictions. In an MDP module, variables can be compared together without any limitations.

In [19], the semantics of probabilistic timed automata is defined in terms of *timed probabilistic systems*, which show timed, non-deterministic, and probabilistic behaviors. They are a variant of Markov decision processes [28] and Segala's probabilistic timed automata [29]. The syntax of probabilistic timed automata is defined as follows.

Definition 1 *(Syntax of PTA).* A probabilistic timed automaton is a $tuple(L, \bar{l}, X, \sum, I, prob)$ where: L is a finite set of locations including the initial location $\bar{l}$; X is a set of clocks; $\sum$ is a finite set of events; the function $I : L \to Zones(X)$ is the invariant condition; and the finite set $prob \subseteq L \times Zones(X) \times \sum \times Dist(2^X \times L)$ is the probabilistic edge relation. □

Let $\mathbb{T} \in \{\mathbb{R}, \mathbb{N}\}$ be the time domain of either the non-negative reals or naturals. A point $v \in \mathbb{T}^{|X|}$ is referred to as a *clock valuation*. Let $\mathbf{0} \in \mathbb{T}^{|X|}$ be the clock valuation which assigns 0 to all clocks in X. For any $v \in \mathbb{T}^{|X|}$ and $t \in \mathbb{T}$, the clock valuation $v \oplus t$ denotes the *time increment* of values in v by t. We use $v[X := 0]$ to denote the clock valuation obtained from v by resetting all of the clocks in $X \in X$ to 0. Let $Zones(X)$ be the set of zones over X, which are conjunctions of atomic constraints of the form $x \sim c$ for $x \in X$, $\sim \in \{\leq, =, \geq\}$, and $c \in \mathbb{N}$. The clock valuation v satisfies the zone ζ, written $v \models \zeta$, if and only if ζ resolves to true after substituting each clock $x \in X$ with the corresponding clock value from v. A state of a probabilistic timed automaton is a pair (l, v) where $l \in L$ and $v \in \mathbb{T}^{|X|}$ are such that $v \models I(l)$.

4.1. Problem statement

In this section we discuss two previously investigated approaches for modeling and verification of PTRebeca models in PRISM. We need an approach in which two features are preserved: First, the definition of rewards is possible to be able to verify expected reachability properties. Second, model checking of a large PTRebeca model should take a reasonable amount of time.

Standard PRISM input language In [19], the integral semantics of a probabilistic timed automaton is implemented in PRISM by using an MDP module and integer-valued variables representing clocks. Similarly, in [18] the TMDP semantics of a PTRebeca model is implemented in PRISM using one MDP module and an integer-valued variable for modeling passage of time. The MDP module is defined using the standard input language of PRISM. We define a specific action called *time*, and each transition of module corresponding to passage of time is labeled with this action. Using time action and the ability of assigning rewards to transitions in PRISM, we can analyze expected-time reachability and time-bounded probabilistic reachability properties. More generally, expected reachability and probabilistic reachability properties can be verified for PTRebeca models.

In [18], we examine different case studies with different sizes. When the PTRebeca model is small, like the ticket service example, the model checking is fast and takes a few seconds. When the model has a medium size, like the sensor network example, its corresponding MDP module includes many states and transitions and its analysis takes a few minutes. Obviously, model checking a large case study will take more time. To support this claim, in Section 6.3 we report the time and memory needed to evaluate different PTRebeca models with PRISM as the backend model checker. Although this approach supports the definition of rewards, model checking of large PTRebeca models takes a significant amount of time.

Explicit engine of PRISM To tackle the problem mentioned above, instead of using the standard PRISM input language, we decided to use the possibility of constructing models in PRISM through direct specification of transition and state matrices. In [18], we used this method for the sensor network case study. We provided the corresponding MDP in the form of its transition and state matrices and input the matrices into PRISM for model checking. This method is faster than the previous one, but rewards are not supported and we can only evaluate probabilistic reachability properties.

To provide faster model checking and support rewards for PTRebeca models, we introduce the parallel composition approach for PTRebeca models which can be verified using PRISM. In this approach, rewards can be defined and so the evaluation of both expected reachability and probabilistic reachability properties is possible. In the following section, we first introduce the approach and then we have to investigate the efficiency of our approach for medium-size and large PTRebeca models in terms of the state space size.

4.2. Parallel composition approach for Probabilistic Timed Rebeca

Probabilistic timed automata (PTA) is one of the most widely used modeling languages for modeling of real-time probabilistic systems. It is supported by the Modest toolset [30] and by PRISM. An alternative approach for performance analysis of a PTRebeca model is a component-wise mapping of the PTRebeca model to a number of PTA. The parallel composition of these modules (PTAs) represents the PTRebeca model. We optimized the mapping to achieve the smallest possible state space, similar to what we did for mapping from Timed Rebeca to timed automata in [12]. In the proposed mapping, each rebec is mapped into two timed automata, called *rebec-behavior* automaton and *rebec-bag* automaton. Additionally, one time automaton is defined to handle the behavior of *after* primitive for all rebecs, called *after-handler* automaton.

The *rebec-behavior* automaton models the behavior of a rebec according to the statements of its message servers and valuations of state variables. The state variables of each rebec are mapped into variables of its corresponding *rebec-behavior* automaton and its statements are mapped to transitions of the automaton. The *rebec-bag* automaton handles the behavior of the message bag of each rebec using an internal buffer. The *rebec-bag* accepts messages which are sent to its corresponding rebec asynchronously, regardless of the state of the corresponding *rebec-behavior* automaton. The *after-handler* automaton handles the messages which should be delivered to *rebec-bag* automata in the future (messages which are sent by *after* primitive). The *after-handler* automaton accepts messages and put them into its buffer until the release time of the messages arrives. When a message in buffer of *after-handler* is released, it is sent to its corresponding *rebec-bag* automaton. Each probabilistic timed automaton can be implemented in PRISM in the form of an MDP module with integer-valued variables representing digital clocks.

In Rebeca language and its extensions, the execution of message servers are atomic, making the coarse-grain execution of a Rebeca model possible [9]. The coarse-grain execution of message servers reduces the state space size significantly. We use the same approach in the TMDP semantics of a PTRebeca model to reach a smaller state space. In the parallel composition approach, we implemented coarse-grain execution by combining statements of different transitions; however, because of synchronization points among automata, there is a poor chance for combining statements. It is the main obstacle against using network of PTA as an ideal approach for PTRebeca models. Different automata need to be synchronized on different points: when a message is sent; when a message is taken from the message bag to start its execution; when a transition modeling a *delay* statement is reached; when it is the time for a sent message to be delivered to its receiver. The mapping from a PTRebeca model to PTA is discussed in more details in the following subsection.

4.2.1. Mapping from a PTRebeca model to PTA.

To show the mapping procedure, the PTRebeca model of Fig. 2 in Section 2 is considered and the resulting PTAs are explained. In these PTAs, the condition on a state should be satisfied on its outgoing transitions. It means that, the condition on a state can be considered as a guard for all the outgoing transitions. The transition guard is the same as the one in PTA, meaning that a transition can be enabled if its guard is satisfied. The mapping is not straightforward because in PTRebeca message passing is asynchronous, while in PTA message passing occurs synchronously.

Rebec-behavior The rebec-behavior automaton models the behavior of a rebec according to the statements of its message servers and valuations of variables. To construct the rebec-behavior automaton of a rebec, a corresponding PTA is generated for each message server, and then PTAs (of message servers) are connected together in a way to describe the overall behavior of the rebec. Figs. 4, 5, and 6 show the rebec-behavior PTA for reactive classes of Customer, TicketService, and Agent, respectively.

Here, we explain the mapping of different statements and valuation of state variables.

- State variables: State variables are mapped to variables of the PTA.
- Ordinary Statements: The mapping for statements like conditionals, loops, assignments, etc., is straightforward.
- Non-deterministic Assignment: A non-deterministic statement is mapped to a number of states and transitions. The number of states depends on the number of different possible values for the variable. Line 23 of Fig. 2 is mapped to transitions from “S2” to “S5” and to “S6” in Fig. 6. The transitions are chosen non-deterministically.

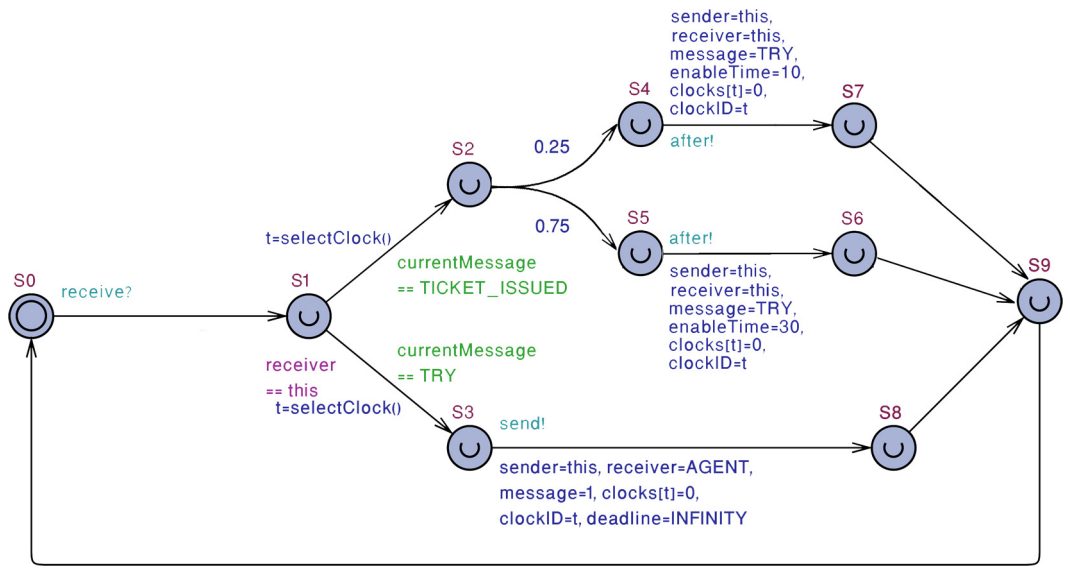


Fig. 4. The rebec-behavior PTA of Customer reactive class.

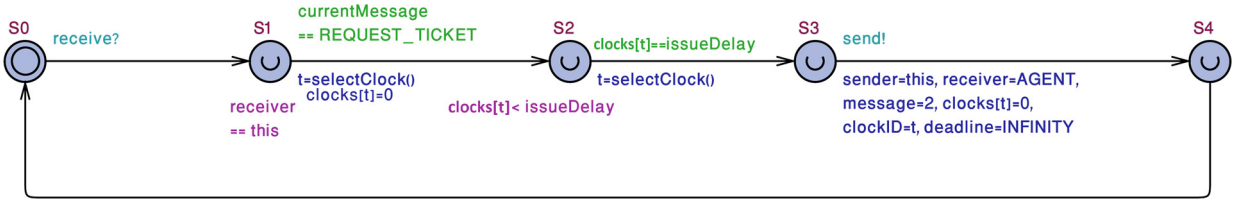


Fig. 5. The rebec-behavior PTA of TicketService reactive class.

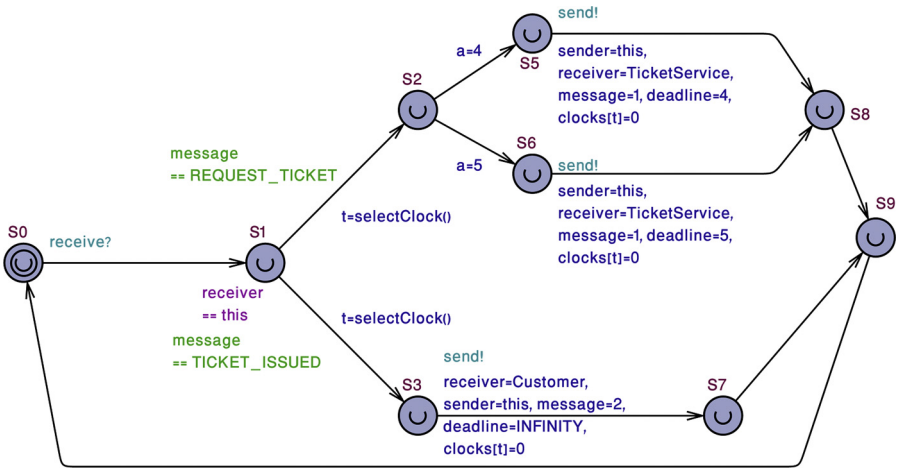


Fig. 6. The rebec-behavior PTA of Agent reactive class.

- Probabilistic Assignment: In PTA, a probabilistic assignment statement is mapped to a number of states, each of them assigning a different value to the variable, and a probabilistic transition into the states. The mapping of line 42 of Fig. 2 is shown in Fig. 4 as the transitions from “S2” to “S4” and “S5”.
- Delay Statement: Delays are mapped by the use of one clock, a location and transition guards to PTA. Mapping for delay statement of line 12 of Fig. 2 is depicted in Fig. 5, specified as transitions from “S1” to “S3”. The required clock is extracted from pool of clocks using function *selectClock*.
- Sending Message Statement: In PTRebeca, each rebec has an internal clock, which shows the time elapsed since the creation of the rebec. This specifies an absolute model of time, which cannot be implemented in PTA, because it makes clock values to grow unboundedly. To solve the problem, for message sending, a clock is dedicated to the message. The

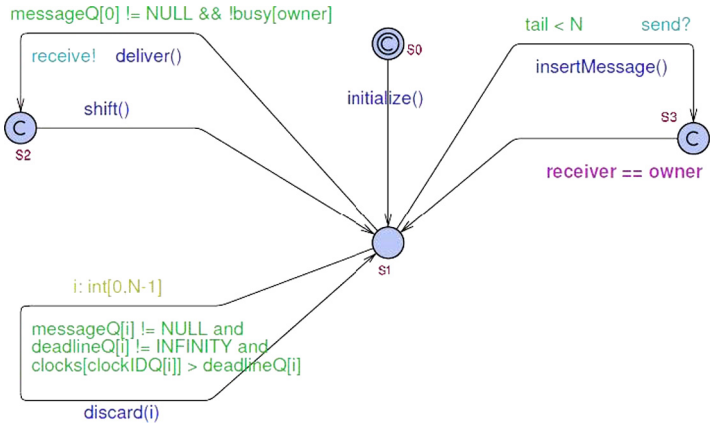


Fig. 7. PTA of rebec-bag for a rebec.

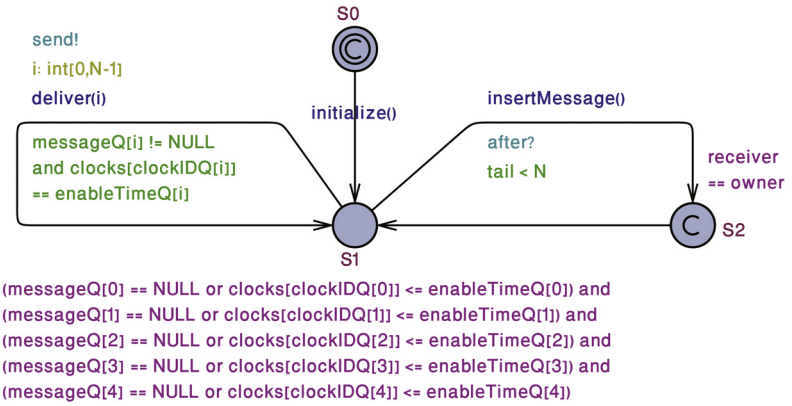


Fig. 8. PTA of After-handler.

clock of a message is used for checking its deadline and enabling time. The clock is returned to the pool, when the message is delivered to the rebec-behavior automaton for execution. For example, message sending of Line 24 of Fig. 2 is mapped to the transitions from “S1” to “S3”, and from “S3” to “S7” in Fig. 6.

Message sending is synchronized with either rebec-bag automaton or after-handler automaton. We use channel “send” if the message is sent immediately and channel “after” if the sent message has “after” value. Messages which are sent via send channel are directly put in the rebec-bag of their receivers. Messages which are sent via after channel are put in a buffer in after-handler automaton. A message will be delivered to the rebec-bag of the receiver when the value of the clock which is dedicated to the message reaches the value of “after”.

Rebec-bag The rebec-bag PTA always accept messages asynchronously, regardless of the state of the corresponding rebec-behavior, and then delivers them, upon the rebec-behavior automaton’s request. The rebec-bag is responsible to handle activation time and deadlines of messages. As depicted in Fig. 7, rebec-bag PTA inserts the incoming messages of the owner rebec (transition from “S1” to “S3”), discards the messages with passed deadlines (self loop transition in “S1”), and extracts the messages from its buffer and delivers them (transition from “S1” to “S2”). Extracting the message from the buffer is done by *shift* function which is used as the update function of transition from “S2” to “S1”.

After-handler The after-handler probabilistic timed automaton always accepts message asynchronously and puts them in a buffer until its enabling time. Fig. 8 shows the PTA of after-handler. As depicted in Fig. 8, it inserts the incoming messages (transition from “S1” to “S2”) and extracts the messages from its buffer and delivers them if the clock of any of them reaches the value of its corresponding enable time, i.e. the value of its after (self loop transition of “S1”).

4.3. Comparison of parallel composition approach with TMDP semantics

In this section, we discuss which of the proposed analysis techniques is appropriate for performance analysis of PTRebeca models. In [12] we reported the size of the state space for timed semantics of Timed Rebeca and parallel composition

approach of Timed Rebeca. In the parallel composition approach, each rebec is converted to a timed automaton and parallel composition of timed automata represents the behavior of Timed Rebeca model. There, UPPAAL was used, a well-known model checker for timed systems, for the parallel composition of timed automata. We developed a tool to generate the state space based on timed semantics of Timed Rebeca.

Experimental results show that the parallel composition of timed automata generates too many states in comparison to timed semantics of Timed Rebeca. The main reason of this difference lies in the modeling of asynchronous message passing between actors using synchronous communication between timed automata. This increases the number of states. This problem is also mentioned in [31] on modeling distributed systems using timed automata. Additionally, the number of clocks grows linearly by the number of rebecs. When the number of clock increases, the state space grows exponentially. We have the same results for comparison of the parallel composition of PTAs and the TMDP semantics of a PTRebeca model. The parallel composition of PTAs generates too many states.

We explained the parallel composition approach in Section 4.2 without considering the details of implementation. We chose PRISM, a well-established model checker, for modeling PTAs and verifying probabilistic properties. Since the input language of PRISM is a state-based language and lacks array, conditional and loops statements, implementing these statements in PTAs increases the number of generated states significantly. For example, implementation of rebec-queue PTA and related functions (like insert, shift, and discard shown in Fig. 7) adds many states to the corresponding PTA. So, the proposed approach becomes more complicated and creates a large state space.

Although the parallel composition approach supports the definition of rewards, it generates more states comparing to TMDP semantics of a PTRebeca model, and it cannot be the suitable approach for large PTRebeca models. In Section 5 and Section 6, we provide a new approach in which the TMDP model of a large case study is converted into one Markov automata. Then, the IMCA is used as the back-end model checker to analyze the PTRebeca model against expected reachability and probabilistic reachability properties.

5. Performance analysis of PTRebeca models using IMCA

As we concluded in Section 4.3, the parallel composition approach is not efficient for performance analysis of large PTRebeca models. To provide a practical approach, we convert the TMDP underlying a PTRebeca model to a Markov automaton (MA) [32] to be able to use the IMCA model checker for performance analysis of PTRebeca models. In this section, we mathematically prove that expectation properties are preserved by this conversion. The proofs are presented for minimum expected time reachability and minimum expected reward reachability properties. Maximum values of expected time reachability and expected reward reachability can be proved similarly.

5.1. Preliminaries

Prior to our proof, we have to prepare the following definitions and notations for TMDP. We also define how a TMDP is converted to MA.

Definition 2 *(Timed Markov decision process).* A timed Markov decision process $\mathcal{T} = (S, s_0, Act, \hookrightarrow, L)$ consists of a set S of states, an initial state $s_0 \in S$, a set Act of actions, a timed probabilistic, non-deterministic transition relation $\hookrightarrow \subseteq S \times Act \times \mathbb{N} \times Dist(S)$ such that, for each state $s \in S$, there exists at least one tuple $(s, a, d, \mu) \in \hookrightarrow$. □

The transitions in a TMDP are performed in two steps: given that the current state is s, the first step is a non-deterministic selection of $(s, act, d, \mu) \in \hookrightarrow$, where act denotes a possible action and d specifies the duration of the transition; in the second step, a probabilistic transition to state s' is made with probability $\mu(s')$. Function $\mu \in Dist(S)$ denotes a discrete probability distribution.

We present the TMDP $\mathcal{T}_{\mathcal{M}}$ of a given PTRebeca model $\mathcal{M}$ as a tuple $(S, s_0, Act, \rightarrow, \Rightarrow)$ where S is a set of states, s_0 is the initial state, Act is a set of actions which consists of τ, signatures of all the messages, and $\mathbb{N}$. Considering Section 3, the union of scheduler and msg-fetcher transitions is $\rightarrow$ (probabilistic transitions) and the set of time-progress transitions (delay transitions) is $\Rightarrow$.

In the TMDP of a PTRebeca model, because of the maximal progress assumption, probabilistic transitions have a higher priority than delay transitions in the execution as their execution time is zero. According to the maximal progress assumption, transitions with execution time of zero, i.e. probabilistic transitions, must be executed before any time progress which is caused by the execution of delay transitions. Therefore, in states with enabled probabilistic transitions, delay transitions are disabled. Here, states with some enabled probabilistic transitions are called probabilistic states (PS) and states with delay transitions are called delay states (DS). For a given delay state s the value of its unique outgoing delay transition is shown by d_s.

Definition 3 *(Paths).* A path in a TMDP is an infinite sequence $\pi = s_0 \xrightarrow{\sigma_0, \mu_0, t_0} s_1 \xrightarrow{\sigma_1, \mu_1, t_1} \cdots$ where $s_i \in S$, $\sigma_i \in Act \cup \{\perp\}$, and $t_i \in \mathbb{N}$. In case of $\sigma_i \in Act$ the value of t_i is zero and it means that TMDP moves from s_i to s_{i+1} using a probabilistic transition with the probability of $\mu_i = \mu_{\sigma_i}^{s_i}(s_{i+1})$. In case of $\sigma_i = \perp$ the value of t_i is larger than zero and TMDP moves from

s_i to s_{i+1} after residing t_i units of time with the probability of $\mu_i = 1$. For any given $t \in \mathbb{N}_{>0}$, $\pi@t$ denotes the sequence of all states that π occupies at time t, i.e. $\pi@t$ specifies a path at time t. □

Due to the instantaneous probabilistic transitions, a TMDP may occupy various states at the same time instance. The time elapsed along the path π is computed by $\sum_{i=0}^{\infty} t_i$. Path π is Zeno whenever this summation converges to a number and its corresponding TMDP has Zeno behavior. A TMDP has Zeno behavior if and only if it has a strongly connected component with only probabilistic transitions. In the rest of this paper we assume that TMDPs do not have Zeno behavior.

Definition 4 *(Policies).* Policies are used to resolve non-deterministic choices in states. To define a probability space, non-determinism should be resolved. A policy is a measurable function (ranged over D) which provides for each finite path ending in state s, a probability distribution over the set of enabled transitions in s. A stationary deterministic policy is a special type of policy which always takes the same decision in a state s. □

Definition 5 *(Stochastic Shortest Path (SSP) problem).* A tuple $(S, s_0, G, Act, \mathbf{P}, c, g)$ is a SSP problem (non-negative) such that $(S, s_0, Act, \mathbf{P})$ is a MDP, $G \subseteq S$ is a set of goal states, $c : S \setminus G \times Act \to \mathbb{R}_{\geq 0}$ is a cost function for non-goal states, $g : G \times Act \to \mathbb{R}_{\geq 0}$ is a cost function for goal states. □

As described in [33], the minimum expected cost reachability of one of the goals states in G from state s, shown by $eR^{min}(s, \Diamond G)$, can be obtained by solving a linear programming (LP) problem. To compute the minimum expected cost reachability, we reduce the analysis of a TMDP to the analysis of a non-negative SSP problem to be able to use an LP problem.

In addition to the above definitions on TMDPs, we have to formally define MAs.

Definition 6 *(A Markov Automaton).* An MA is a transition system with two types of transitions, called probabilistic and Markovian transitions, shown by tuple $(S', s'_0, Act', \to', \Rightarrow')$. Here, S' is a set of states, $s'_0 \in S'$ is an initial state, Act' is a set of actions, $\to'$ is set of probabilistic transitions, and $\Rightarrow'$ is a set of Markovian transitions. Probabilistic transitions are instantaneous transitions which are defined as $\to' \subseteq S' \times Act' \times Distr(S')$ (where $Distr(S')$ denote the set of discrete probability distribution functions over the countable set S') and Markovian transitions are defined as $\Rightarrow' \subseteq S' \times \mathbb{R}_{\geq 0} \times S'$ [32]. □

Here, transition $(s', \alpha, \mu) \in \to'$ is abbreviated to $s' \overset{\alpha}{\to}' \mu$ and $(s', \lambda, t') \in \Rightarrow'$ by $s' \overset{\lambda}{\Rightarrow}' t'$. An MA can evolve via its probabilistic and Markovian transitions. In case of $s' \overset{\alpha}{\to}' \mu$, it leaves state s' by executing action α and state t' is its destination with the probability of $\mu(t')$. Here, s' is called a probabilistic state (PS). In case of $s' \overset{\lambda}{\Rightarrow}' t'$, state s' is left after waiting for exponentially distributed units of time with rate λ and the target state is t'. It means that the expected delay from s' to t' is $1/\lambda$. Here, state s' is called Markovian state (MS).

In the rest of this paper we used the primed version of alphabet and arrows to address MAs and the normal ones to address TMDPs.

Definition 7 *(Conversion of the TMDP of PTRebeca model $\mathcal{M}$).* A given TMDP $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \to, \Rightarrow)$ is converted to MA $\mathcal{A}_{\mathcal{M}} = (S', s'_0, Act', \to', \Rightarrow')$ such that $S = S'$, $s_0 = s'_0$, $Act = Act'$, and $\to = \to'$. In addition, $(s, d, t) \in \Rightarrow$ implies that $(s', 1/d, t') \in \Rightarrow'$. In other words, $\mathcal{T}_{\mathcal{M}}$ and $\mathcal{A}_{\mathcal{M}}$ are the same except that the delay transitions in $\mathcal{T}_{\mathcal{M}}$ are converted to Markovian transitions in $\mathcal{A}_{\mathcal{M}}$. In this conversion, if a given state $s \in S$ is a delay state in $\mathcal{T}_{\mathcal{M}}$, its corresponding state $s' \in S'$ is a Markovian state in $\mathcal{A}_{\mathcal{M}}$, and if s is a probabilistic state in $\mathcal{T}_{\mathcal{M}}$ its corresponding state s' is probabilistic state in $\mathcal{A}_{\mathcal{M}}$. □

5.2. Expected time reachability in TMDP

Assume that for a given PTRebeca model $\mathcal{M}$, its TMDP $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \to, \Rightarrow)$ is given and the set of goal states $G \subseteq S$ is defined. Here, we want to find the minimum expected time for reaching one of the states in G. So, we need to define a random variable on the time which is spent in paths from the start state to one of the goal states of G. Assume that random variable $V_G : Paths \to \mathbb{N}$ is this random variable. So, the minimum expected time reachability for a given state $s \in S$ to one of the goal states is defined by

$$eT^{min}(s, \Diamond G) = \inf_D \mathbb{E}_{s,D}(V_G) = \inf_D \sum_{\pi \in Paths} V_G(\pi) \cdot Pr_{s,D}(\pi)$$

where D is a generic policy on $\mathcal{M}$. To compute the value of $eT^{min}(s, \Diamond G)$ we have to reformulate the above equation into a linear equation system, as shown in the following theorem. Note that the proofs of theorems are depicted in the Appendix.

Theorem 1. *The function eT^{min} is a fix point of the Bellman operator*

$$[L(v)](s) = \begin{cases} d_s + v(t) & s \in DS \setminus G \\ \min\limits_{a \in Act(s)} \left\{ \sum\limits_{t \in S} \mu_a(t) \cdot v(t) \right\} & s \in PS \setminus G \\ 0 & s \in G \end{cases} \quad \square \tag{1}$$

The characterization of $eT^{min}(s, \Diamond G)$ in Theorem 1 allows us to reduce computing the minimum expected time reachability problem in a TMDP to the minimum expected time reachability in a non-negative SSP problem, denoted by ssp_{et}.

Definition 8 *(SSP for minimum expected time reachability).* The SSP of a given TMDP $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$ for the expected time reachability to a set of goal states $G \subseteq S$ is a tuple $ssp_{et}(\mathcal{M}) = (S, s_0, Act \cup \{\perp\}, G, c, g)$ where:

- $S, s_0,$ and Act in TMDP and ssp_{et} are the same,
- $\mathbf{P}(s, \alpha, t) = \begin{cases} 1 & s \in DS \setminus G \\ \mu^s_\alpha(t) & s \in PS \setminus G \\ 0 & s \in G \end{cases}$,
- $c(s, \alpha) = \begin{cases} d_s & s \in DS \setminus G \wedge \alpha = \perp \\ 0 & \text{otherwise} \end{cases}$,
- $g(s) = 0$. $\square$

As shown in [34], the minimum expected cost problem of a SSP has a unique fixed point; which enables us using standard solution techniques like value iteration and linear programming to compute the minimum expected cost of SSP. Using the reduction from TMDPs to SSP problems, we can use the same techniques as there is only one fixed point in TMDPs.

Theorem 2. *For a TMDP $\mathcal{T}_{\mathcal{M}}$ the value of $eT^{min}(s, \Diamond G)$ equals to $cR^{min}(s, \Diamond G)$ in $ssp_{et}(\mathcal{M})$.* $\square$

This way, we showed how the minimum expected time reachability for a TMDP is computed. As we want to use the IMCA for computing the expected time reachability of the TMDP, we present the conversion of a TMDP to its corresponding MA (which can be analyzed by the IMCA). Then, we prove that expected time reachability in the TMDP and its conversion in the form of MA are equal.

As shown in [32] for a given Markov Automaton $\mathcal{A}_{\mathcal{M}} = (S', s'_0, Act', \rightarrow', \Rightarrow')$ the following Bellman operator is used for finding the expected time reachability.

$$[L(v)](s') = \begin{cases} \frac{1}{E(s')} + \sum\limits_{t' \in S'} P(s', t') \cdot v(t') & s' \in MS \setminus G' \\ \min\limits_{a \in Act(s')} \left\{ \sum\limits_{t' \in S'} \mu_a(t') \cdot v(t') \right\} & s' \in PS \setminus G' \\ 0 & s' \in G' \end{cases} \tag{2}$$

The TMDP conversion to its corresponding MA preserves the expected time reachability properties. As depicted in Equations (1) and (2), for a given state $s \in S$ in $\mathcal{T}_{\mathcal{M}}$ where s is a probabilistic (or goal) state, its corresponding state $s' \in S'$ in $\mathcal{A}_{\mathcal{M}}$ is probabilistic (or goal) state, and the equations of finding $[L(v)](s)$ is the same as $[L(v)](s')$. In the case of s is a delay state, based on the semantics of PTRebeca, delay states have only one outgoing delay transition. So, in its corresponding state s' in $\mathcal{A}_{\mathcal{M}}$ there is only one outgoing transition with the probability of one, results in changing the formula of computing the expected time reachability from $\frac{1}{E(s')} + \sum_{t' \in S'} P(s', t') \cdot v(t')$ to $\frac{1}{E(s')} + v(t')$. As during conversion from $\mathcal{T}_{\mathcal{M}}$ to $\mathcal{A}_{\mathcal{M}}$ a delay value d_s is changed to $1/d_s$, there is $\frac{1}{E(s')} + v(t') = d_s + v(t)$. Here, we assumed that there are states $t \in S$ and $t' \in S'$ such that $s \overset{d_s}{\Rightarrow} t$ and $s' \overset{1/d_s}{\Rightarrow'} t'$. In a nutshell, the minimum expected time reachability in all three cases of Equation (1) for state s is the same as the minimum expected time reachability in all three cases of Equation (2) for state s'.

5.3. Expected reward reachability in TMDP

We want to compute expected reward reachability in TMPDs where the rewards are associated to delay states and probabilistic transitions. This is similar to what we did for computing expected time reachability in TMDPs. Assume that there are two functions ρ and r for accessing to the associated rewards to states and transitions respectively. For a given

state s function $\rho(s)$ returns the reward which is associate to s. For a given transition from s to t with action α, function $r(s,\alpha)$ returns the reward which is associated to the transition.

Now, assume that TMDP $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$ is given and the set of goal states is defined as $G \subseteq S$. Here, we want to find the minimum expected reward which is gained from each state $s \in S$ to one of the states in G. So, we need to define a random variable on the total reward which is gained in paths from s to one of the goal states of G. Assume that random variable $R_G : Paths \rightarrow \mathbb{N}$ is this random variable. So, the minimum expected reward reachability from s to one of the goal states is defined by

$$eR^{min}(s, \Diamond G) = \inf_D \mathbb{E}_{s,D}(R_G) = \inf_D \sum_{\pi \in Paths} R_G(\pi) \cdot Pr_{s,D}(\pi)$$

where D is a generic policy on $\mathcal{M}$. To compute the value of $eR^{min}(s, \Diamond G)$ we have to reformulate the above equation into a linear equation system, as shown in Theorem 3.

Theorem 3. *The function eR^{min} is a fix point of the Bellman operator*

$$[L(v)](s) = \begin{cases} d_s \times \rho(s) + v(t) & s \in DS \setminus G \\ \min\limits_{a \in Act(s)} \left\{ r(s,a) + \sum\limits_t \mu_a(t) \cdot v(t) \right\} & s \in PS \setminus G \\ 0 & s \in G \end{cases} \tag{3}$$

The characterization of $eR^{min}(s, \Diamond G)$ in Theorem 3 allows us to reduce computing the minimum expected reward reachability problem in TMDPs to the minimum expected cost in non-negative SSP problems, shown by ssp_{er}.

Definition 9 *(SSP for minimum expected reward reachability).* The SSP of a given TMDP $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$ for the expected reward reachability to a set of goal states $G \subseteq S$ is a tuple $ssp_{er}(\mathcal{M}) = (S, s_0, Act \cup \{\perp\}, G, c, g)$ where:

- S, s_0, and Act in TMDP and ssp_{er} are the same,
- $\mathbf{P}(s,\alpha,t) = \begin{cases} 1 & s \in DS \setminus G \\ \mu^s_\alpha(t) & s \in PS \setminus G \\ 0 & s \in G \end{cases}$,
- $c(s,\alpha) = \begin{cases} r(s,\alpha) & s \in MS \setminus G \\ 0 & \text{otherwise} \end{cases}$,
- $g(s) = \rho(s)$. □

As the problem is reduced to the minimum expected cost problem of a SSP, we conclude that there is only one fixed point in TMDPs as discussed before.

Theorem 4. *For a TMDP $\mathcal{T}_{\mathcal{M}}$ the value of $eR^{min}(s, \Diamond G)$ equals to $cR^{min}(s, \Diamond G)$ in $ssp_{er}(\mathcal{M})$.* □

This way, we showed how the minimum expected reward reachability for a TMDP is computed. As we want to use the IMCA for computing the expected reward reachability of TMDPs, we present the conversion of a TMDP to its corresponding MA. Then, we prove that expected reward reachability in a TMDP and its conversion in the form of MA are equal.

A given TMDP $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$ with reward functions ρ and r is converted to MA $\mathcal{A}_{\mathcal{M}} = (S', s'_0, Act', \rightarrow', \Rightarrow')$ with reward functions ρ' and r' such that $S = S'$, $s_0 = s'_0$, $Act = Act'$, $\rightarrow = \rightarrow'$, $r(s,a) = r'(s',a)$, and $\rho(s) = \rho'(s')$. The properties of this conversion is the same as the properties of conversion which is described in Section 5.2.

As shown in [35] for a given Markov Automaton $\mathcal{A}_{\mathcal{M}} = (S', s'_0, Act', \rightarrow', \Rightarrow')$ with reward functions ρ' and r' the following Bellman operator is used for finding the expected time reachability.

$$[L(v)](s') = \begin{cases} \dfrac{\rho'(s')}{E(s')} + \sum\limits_{t' \in S'} P(s',t') \cdot v(t') & s' \in MS \setminus G' \\ \min\limits_{a \in Act(s')} \left\{ r'(s',a) + \sum\limits_{t' \in S'} \mu_a(t') \cdot v(t') \right\} & s' \in PS \setminus G' \\ 0 & s' \in G' \end{cases} \tag{4}$$

This conversion preserves the expected time reachability of states. As depicted in Equations (3) and (4), for a given state $s \in S$ in $\mathcal{T}_{\mathcal{M}}$ where s is a probabilistic (or goal) state, its corresponding state $s' \in S'$ in $\mathcal{A}_{\mathcal{M}}$ is probabilistic (or goal) state,

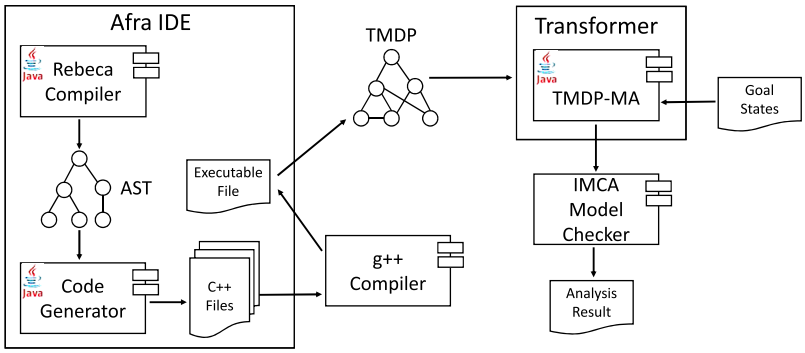


Fig. 9. The architectural overview of the analyzer of PTRebeca models.

and the equations of finding $[L(v)](s)$ is the same as $[L(v)](s')$. In the case of s is a delay state, based on the semantics of PTRebeca, delay states have only one outgoing delay transition. So, in its corresponding state s' in $\mathcal{A}_{\mathcal{M}}$ there is only one outgoing transition with the probability of one, results in changing the formula of computing the expected reward reachability from $\frac{\rho'(s')}{E(s')} + \sum_{t' \in S'} P(s', t') \cdot v(t')$ to $\frac{\rho'(s')}{E(s')} + v(t')$. As during conversion from $\mathcal{T}_{\mathcal{M}}$ to $\mathcal{A}_{\mathcal{M}}$ a delay value d_s is changed to $1/d_s$, there is $\frac{\rho'(s')}{E(s')} + v(t') = d_s \times \rho(s) + v(t)$. Here, we assumed that there are states $t \in S$ and $t' \in S'$ such that $s \overset{d_s}{\Rightarrow} t$ and $s' \overset{1/d_s}{\Rightarrow}' t'$. In a nutshell, the minimum expected reward reachability in all three cases of Equation (3) for state s is the same as the minimum expected reward reachability in all three cases of Equation (4) for state s'.

6. The toolset and case studies

To illustrate the applicability of the approach presented in this paper, we developed a toolset and analyzed two different case studies, which are accessible from Rebeca home page [36]. The architectural overview of the toolset is depicted in Fig. 9. As shown in the figure, Afra IDE serves as the front-end of the toolset and IMCA [24] is the back-end model checking engine of the toolset.

Using the Afra IDE, a number of C++ files are generated for a PTRebeca model. These C++ files are compiled and linked by g++ compiler, which results in an executable file. Running the executable file generates the TMDP of the model (i.e. the state space of the model). In the PTRebeca models, the size of message bags is bounded. The state space of a PTRebeca model is finite when the model shows a recurrent behavior. We used the time-shift equivalence approach, proposed in [12], to make the state space finite.

The *TMDP-MA* tool is developed to convert the TMDP of the model to the input language of IMCA model checker. To perform the conversion, the generated TMDP and the specification of the *goal states* of the model are input to *TMDP-MA* and one Markov automaton is generated. The obtained MA is imported to the IMCA for model checking.

Evaluation of the toolset IMCA provides algorithms for expected time and expected reward reachability analysis, long-run average analysis, time-bounded probabilistic reachability and probabilistic reachability analysis of Markov automata. Since IMCA is used as the backend model checker for PTRebeca models, we investigate which properties are preserved by the conversion (refer to Definition 7), and so can be evaluated by our developed toolset.

In Section 5, we proved that expected time reachability and expected reward reachability properties are preserved by the conversion. By using a dedicated time action in the TMDP (equivalently in its corresponding MA) and because of the ability of assigning rewards to the transitions in IMCA, expected reachability properties can be computed for a PTRebeca model. Therefore, our toolset can be used for the evaluation of expected reachability properties of PTRebeca models.

According to Definition 7, probabilistic transitions in the TMDP are directly converted to probabilistic transitions in the MA. Obviously, probabilistic reachability properties are preserved in this conversion, and so can be checked for PTRebeca models. The rate of a Markovian transition in the MA is estimated by the inverse of the integer value of a corresponding delay transition in the TMDP. Because of this estimation, time-bounded probabilistic reachability properties are not preserved by the conversion. So, we are not able to evaluate this type of properties for PTRebeca models. We believe that long-run average properties are preserved by the conversion, but its mathematical proof remains as a future work.

In the following sections, we choose two case studies to cover the evaluation of two types of properties: expected reachability and probabilistic reachability. The probabilistic reachability property is checked for toxic gas sensing system, and the expected time reachability is calculated for network on chip case study. These case studies show the applicability of the toolset for the performance evaluation of systems.

6.1. Performance analysis of a toxic gas sensing system

In the conference paper [18], we examined toxic gas sensing (named sensor network in the conference paper) case study and used PRISM as the back-end model checker. Here, we perform the experiments again and use IMCA for performance evaluation. We obtain the same results via PRISM and IMCA. The system consists of a lab environment in which the level of a toxic gas changes over time. There is one scientist in this lab. If the toxic gas level raises above a certain threshold, the scientist's life is in danger. A sensor in the lab periodically measures the amount of toxicity in the air, and send the measurements to a central controller. The central controller periodically checks whether the scientist is in danger. If so, it notifies the scientist about the danger. The scientist should acknowledge the notification; if the scientist fails to do so in a timely manner, the central controller notifies a rescue team. When the team reaches the lab it notifies the controller that the scientist has been rescued. If the controller does not receive this notification, it means that the scientist has lost his life.

PTRebeca model The PTRebeca model of this system is shown in Fig. 10 and contains five different reactive classes: `Environment`, `Rescue`, `Controller`, `Sensor`, and `Scientist`. The toxic level of the environment changes periodically by a probabilistic assignment of line 23. The sensor periodically measures the level of toxic gas by sending `giveGas` message to the environment (which is modeled in line 44). After sensing, the sensor reports the measured data to the controller. The sensor may fail to report the measured data as shown by the probabilistic assignment of line 46. Upon receiving the measured data from a sensor (in the `report` message server), the controller stores the value in `sensorValue0` (line 82).

Periodically, in the `checkSensors` message server, the controller checks if the reported value is above the normal amount. In case of detecting high toxicity, the controller informs the scientist by sending `abortPlan` message (line 93), and checks the scientist's acknowledgment after a specified amount of time (line 95). If the controller does not receive an `ack` message from the scientist, the rescue team is informed about the situation. If this process takes more than the value of `scientistDeadline` units of time, the scientist will die and this is modeled by sending a message `die` to the scientist by the environment. This message is scheduled immediately after changing the gas level to the dangerous level (line 25).

In this model, the network delay is assumed to be one time unit. In different experiments, we consider different values for the period in which the sensor measures the toxic level of the lab. The value ranges from 1 to 25. The period of the controller checking the sensor's data is 5 time units.

Experimental results We study one property for this model which is "what is the probability of the death of the scientist?". Since the model includes non-deterministic behaviors, the model checker computes the maximum and the minimum probabilities over all paths in the generated state space.

Fig. 11(a) shows the maximum and the minimum probabilities of the scientist death when the value of variable *checkingPeriod* of sensor changes. If the sensor checks the environment with a high frequency (i.e. the value of variable *checkingPeriod* is low), the message server *checkGasLevel* is sent frequently (Line 49) which incurs frequently sending the message server *doReport*. So, the probability of sensor failure increases, resulting in high probability of the scientist death.

For example, when the sensor checks the environment once every unit of time, i.e. the value of *checkingPeriod* equals one (Line 49), the environment is checked five times before the first change in the environment. The first change in the environment occurs at time 5 as the value of variable *changingPeriod* equals 5 (Line 18). Therefore, the probability of sensor failure increases because of working a lot during time. When the sensor frequency is low, the environment changes cannot be detected on time; resulting in a high probability of the scientist death.

There is an optimum value for the variable *checkingPeriod* (i.e. sensor frequency) which is five according to the obtained results reported in Fig. 11(a) for the minimum probability of the scientist death. The value of variable *scientistDeadline* equals 10. As the results show, the trend of changes in the value of the maximum probability of the scientist death is normal but at times 5, 10, 15, 20, and 25 it jumps to one. At these times because of concurrency between time related behaviors in the system, there is a scenario in which the dangerous level is reported too late to the administrator and the scientist will die. At these times, the execution sequence of the following messages is important and causes the special behavior: (1) checking the sensor value by the controller (it is repeated periodically after 5 units of time), (2) changing the toxic level of the environment to a dangerous level (period is 5 units of time), (3) checking the environment by the sensor (Fig. 11(a) shows the probability of the scientist death for different value of this period), and (4) sending a message *die* to the scientist (after 10 units of time) when the environment is dangerous.

In Fig. 11(b), the value of variable *scientistDeadline* equals 12; the scientist has more time to be saved before being killed by the toxic environment. The maximum probability of the scientist death is not equal to one at times 5, 10, 15, 20, and 25, but because of concurrency between time related behaviors, there is a scenario in which the dangerous level is reported too late and consequently the maximum probability of the scientist death increases. The same as the previous case, there is an optimum value for the variable *checkingPeriod*, i.e. sensor frequency, which is ten in this experiment.

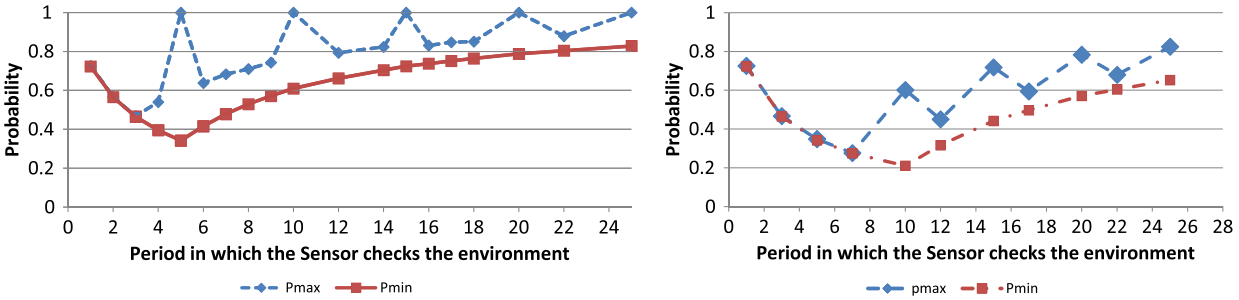
6.2. Performance analysis of network on chip

Our second example is a model of a network on chip (NoC). NoC has emerged as a promising architecture paradigm for many-core systems. As complexity grows in NoCs, functional verification and performance evaluation in the early stages

```
env byte scientistDeadline = 10;
env byte rescueDeadline = 5;
env byte netDelay = 1;
env byte controllerCheckDelay = 5;
env byte sciAckDeadline = 5;
env byte rescuedeadline = 10;

reactiveclass Environment(10){
  knownrebecs{
    Sensor sensor;
    Scientist scientist;
  }
  statevars{
    byte gasLevel, changingPeriod;
    boolean meetDangerousLevel;
  }
  Environment(){
    changingPeriod = 5;
    gasLevel = 2; // 2 = safe, 4 = dangerous
    self.changeGasLevel() after(changingPeriod);
  }
  msgsrv changeGasLevel(){
    if(gasLevel == 2) gasLevel=?(0.98:2, 0.02:4);
    if(gasLevel > 2 && !meetDangerousLevel) {
      scientist.die() after(scientistDeadline);
      meetDangerousLevel = true;
    }
    self.changeGasLevel() after(changingPeriod);
  }
  msgsrv giveGas(){
    if(sender==sensor) sensor.doReport(gasLevel);
  }
}
reactiveclass Sensor(7) {
  knownrebecs {
    Controller controller;
    Environment environment;
  }
  statevars { int checkingPeriod; }
  Sensor(int myPeriod) {
    checkingPeriod = myPeriod;
    self.checkGasLevel();
  }
  msgsrv checkGasLevel() {environment.giveGas();}
  msgsrv doReport(byte value) {
    boolean working = ?(0.01:false,0.99:true);
    if(working){
      controller.report(value) after(netDelay);
      self.checkGasLevel() after(checkingPeriod);
    }
  }
}
reactiveclass Scientist(7) {
  knownrebecs { Controller controller; }
  statevars {boolean isDead, isOutEnv, ackSent;}
  msgsrv die(){ if(!isOutEnv) isDead = true; }
  msgsrv abortPlan() {
    isOutEnv = true;
    if(!ackSent) controller.ack()
        after(netDelay);
    ackSent = true;
  }
  msgsrv leftEnv(){isOutEnv = true;}
}
reactiveclass Rescue(7) {
  knownrebecs {Controller controller;}
  msgsrv go() {
    delay(2); //unexpected obstacle
    controller.rescuereach() after(netdelay)
        deadline(rescuedeadline-netdelay);
  }
}
reactiveclass Controller(13) {
  knownrebecs {
    Sensor sensor;
    Scientist scientist;
  }
  statevars {
    int sensorValue0;
    boolean scientistAck,scientistDead,ackIsSent;
  }
  Controller() { self.checkSensors(); }
  msgsrv report(int value) {
    if (sender == sensor) sensorValue0 = vale;
  }
  msgsrv rescueReach() {
    scientistReached = true;
    scientist.leftEnv();
  }
  msgsrv checkSensors() {
    boolean danger = false;
    if (sensorValue0 > 3) danger = true;
    if(!scientistAck){
      if (danger) {
        scientist.abortPlan() after(netDelay);
        if(!ackIsSent)
          self.checkScientistAck()
              after(sciAckDeadline);
        ackIsSent = true;
      }
      self.checkSensors()
          after(controllerCheckDelay);
    }
  }
  msgsrv ack() {scientistAck = true;}
  msgsrv checkScientistAck() {
    if (!scientistAck)
      rescue.go() after(netDelay);
    scientistAck = false;
  }
}
main {
  Environment environment(sensor, scientist):();
  Sensor sensor(controller,environment):(10);
  Scientist scientist(controller):();
  Controller controller(sensor, scientist,
      rescue):();
  Rescue rescue(controller):();}
```

Fig. 10. The model of toxic gas sensing system.



(a) The value of variable *scientistDeadline* is 10.

(b) The value of variable *scientistDeadline* is 12.

Fig. 11. The maximum and minimum probabilities that the scientist eventually dies, when the sensor frequency changes.

of the design process are suggested as ways to reduce the fabrication cost. Globally Asynchronous Locally Synchronous (GALS) NoC [37] has gained many attentions in designing of such systems. As an example of a NoC, we model and analyze ASPIN (Asynchronous Scalable Packet switching Integrated Network), which is a fully asynchronous two-dimensional GALS NoC design using XY routing algorithm. Using this algorithm, packets can only move along X direction first, and then along Y direction to reach their destination. In ASPIN, packets are transferred through channels, using four-phase handshake communication protocol. The protocol uses two signals, namely *Req* and *Ack* to implement four-phase handshaking protocol. To transfer a packet, first, the sender sends a request by rising *Req* signal, and waits for an acknowledgment which is raising *Ack* from the receiver. All the signals return to zero after a successful communication. There are four adjacent routers to each router and also four internal buffer for storing the incoming packets of different neighbors.

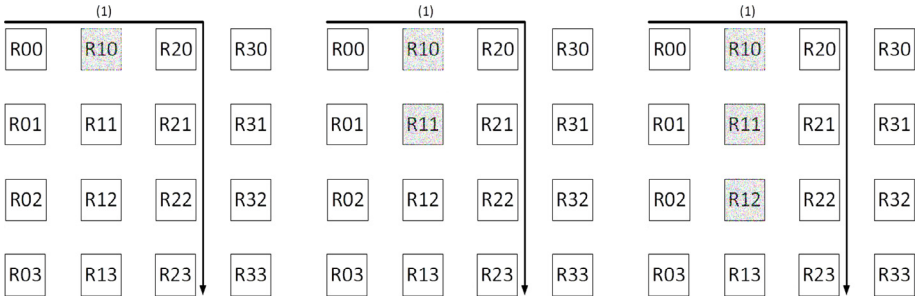
The timed version of ASPIN was investigated in [13] using simulation and model checking. The Timed Rebeca language was used for modeling of ASPIN, and Afra tool-set [36] was applied to the model for estimating the maximum end-to-end latency through model checking. Here, we add faulty routers to ASPIN, and examine the model for different traffic patterns and faulty routers. In the PTRebeca model, all nodes are working correctly with probability one except a few of them which are specified in their constructors. For example, the node with Xid = 1 and Yid = 0 is supposed to be faulty (Line 15 of Fig. 12). A faulty node fails to forward received packets with a specified probability. The probabilistic version of the case study is similar to the timed version presented in [13]. The way we model channels, the topology of the communication, routing algorithm, buffer status, and communication protocol in the model is the same as in [13].

PTRebeca model The simplified version of PTRebeca model of ASPIN is shown in Fig. 12, which contains two different reactive classes: `Manager` and `Router`. The `Manager` does not exists in real NoCs. Here, it is used as the starter of the model. It sends `init` message to routers to ask them for generating packets. This way, different traffic patterns are created by modifying only `Manager`. The `Router` is the model of a router in an ASPIN. So, its definition contains four known rebecs which are its neighbor routers (line 7), its id in XY manner (Xid and Yid in line 8), its buffer variables which show that the buffer is enable or busy (line 10), a variable which shows that whether it works properly or not (line 11), and variables that show whether its neighbors are faulty or not (line 12). The communication channel functionalities among neighbors are modeled by message passing in Rebeca. Four-phase handshake protocol is modeled using three message servers: `reqSend`, `giveAck`, and `getAck`. A router calls its `reqSend` message server to send a request to its neighbor. The `XY-routing` algorithm is implemented inside `reqSend` (lines 25–56) and determines to which neighbor router the packet is sent. If the neighbor router is faulty, a dynamic XY-routing algorithm presented in [38] is used to reroute the packet. The congestion links are not considered in our algorithm. The packet is rerouted to an operative neighbor by calling function `reRoute` (e.g. line 31). In lines 40–46 of Fig. 12, the details of routing a packet with $Xtarget > Xid$ and $Ytarget > Yid$ is shown. If the packet must be sent to the router's east neighbor and the east neighbor is not faulty (line 41), the function `routeToEast` is called (line 42). In this function message `giveAck` is sent to the east neighbor and the internal state of the sender router is changed. The `giveAck` message server first checks the address of the destination of the newly received packet. If the address is the same as the current router, then the packet is consumed (line 85). Otherwise, if the router's buffer is not full (line 75), the packet will be stored and an acknowledgment is sent to the sender router by calling its `getAck` message server (line 80). If the incoming buffer of the neighbor is full (line 73), the router must wait for some amount of time and try sending later, which is modeled by sending to itself (line 74).

To model the behavior of router buffers, we use the rebec's queue to store all packets received by a router and only keep track of the length of north, south, east and west buffers to have buffer status at all time. The variable `bufSize` specifies the buffer size in each direction of routers. In the experiments, `bufSize` equals two (Line 1). Each router has an array `bufNum` which keeps the number of sent packets in each direction for which their `ack` signals haven't been received yet. When a message is sent to a direction, the number of sent messages to that direction is increase by one (Line 76). When an `ack` signal is received from a direction, the number of sent messages to that direction is decreased by one. The complete PTRebeca model of ASPIN is accessible from Rebeca home page [36].

```
env byte bufSize = 2;
reactiveclass Manager(10){
  knownrebecs{Router r00, r10, ... , r33;}
  msgsrv reset(){ r00.init(); r23.init();}
}
reactiveclass Router(10) {
  knownrebecs {Router N, E, S, W;}
  statevars { byte Xid, Yid;
    byte[4] bufNum;
    boolean[4] full, enable, outMutex;
    boolean recieved, isWorking;
    boolean [4] neighborIsWorking; // 0=N, 1=E,
        2=S, 3=W
  }
  Router(byte X, byte Y){ Xid = X; Yid = Y;
      recieved = false;  //specifying faulty
      nodes
    if(Xid == 1 && Yid == 0) self.coreIsFaulty();
    ...
  }
  msgsrv coreIsFaulty (){ isWorking = ?
      (0.95:true,0.05:false);}
  msgsrv init(){ ... }
  void routeToSouth( ){...}
  void routeToNorth(){...}
  void routeToWest(){...}
  void routeToEast(){...}
  void reRoute(){...}
  msgsrv reqSend(byte Xtarget, byte Ytarget, int
      dirS, int packId, boolean routing) {
    if (enable[directionS]){
      boolean sent = false; int hardwareDelay;
          hardwareDelay = 26;
      if (Xid == Xtarget){
        if(Ytarget > Yid){
          if(neighborIsWorking[2] && senderR !=
              2){routeToSouth();}
          else{reRoute();}
        }
        else if (Ytarget < Yid ){...}
      }
      else if (Yid == Ytarget){
        if(Xtarget > Xid){...}
        else if (Xtarget < Xid){...}
      }
      else{ //first move through horizontal
          channels
        if(Xtarget > Xid && Ytarget > Yid){
          if(neighborIsWorking[1] && senderR !=
              1){
            routeToEast();}
          else if(neighborIsWorking[2] &&
              senderR != 2){
            routeToSouth();}
          else{
            reRoute();}
        } else if(Xtarget < Xid && Ytarget >
            Yid){...}
        else if(Ytarget < Yid && Xtarget >
            Xid){...}
        }
        else if(Ytarget < Yid && Xtarget <
            Xid){...}
      }
    }
    else
      self.reqSend(Xtarget,Ytarget,directionS,
          packId,senderR) after(1);
  }
  msgsrv isFaulty(int senderCore,byte Xtarget,
      byte Ytarget, int directionS, int
      packId){...}
  msgsrv getAck(int dirS){ ... }
  msgsrv giveAck(byte Xtarget, byte Ytarget,int
      dirS, int dirD, int msgSender, int
      packId, boolean routing) {
    int MSGSender;
    if(sender == N) MSGSender = 0;
    else if (sender == E) MSGSender = 1;
    ...
    else MSGSender = msgSender;
    if(!isWorking){
      if(MSGSender == 0)
        N.isFaulty(2,Xtarget,
            Ytarget,directionS, packId);
      else if(MSGSender == 1)
        E.isFaulty(3,Xtarget,
            Ytarget,directionS, packId);
      ...
    }
    if(!(Xtarget == Xid && Ytarget == Yid)) {
      if (full[dirD])
        self.giveAck(Xtarget, Ytarget,dirS,dirD,
            MSGSender, packId, routing)
            after(2) deadline(5);
      else {
        bufNum[dirD] = (byte)bufNum[dirD] + 1;
        if (bufNum[dirD] == bufSize)
          full[dirD] = true;
        self.reqSend(Xtarget, Ytarget, dirD,
            packId,routing) after(1);
        if(MSGSender == 0) N.getAck(dirS);
        else if(MSGSender == 1) E.getAck(dirS);
        ...
      }
    }
    else if((Xtarget == Xid && Ytarget == Yid))
        { ... if (packId == 1) recieved = true;
        ...}
  }
}
main {
  Manager m(r00,r10, ... ,r33):();
  Router r00(m,r03,r10,r01,r30):(0,0);
  Router r10(m,r13,r20,r11,r00):(1,0);
  ...
  Router r33(m,r32,r03,r30,r23):(3,3);
}
```

Fig. 12. The model of ASPIN network.



(a) R10 is faulty in scenario 1 (b) R10 and R11 are faulty in scenario 2 (c) R10, R11, and R12 are faulty in scenario 3

Fig. 13. The 4×4 ASPIN model: The traffic in experiment 1.

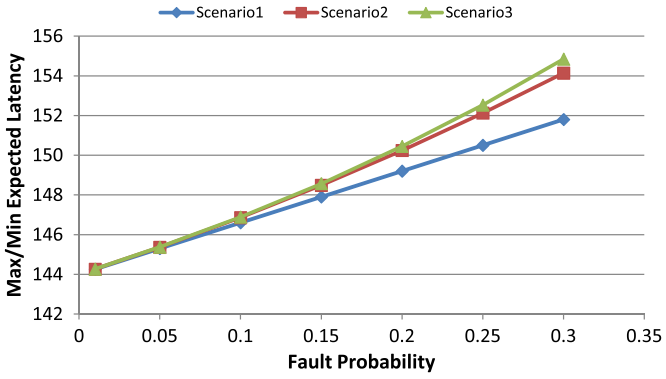


Fig. 14. Experiment 1: the min/max expected latency for different scenarios.

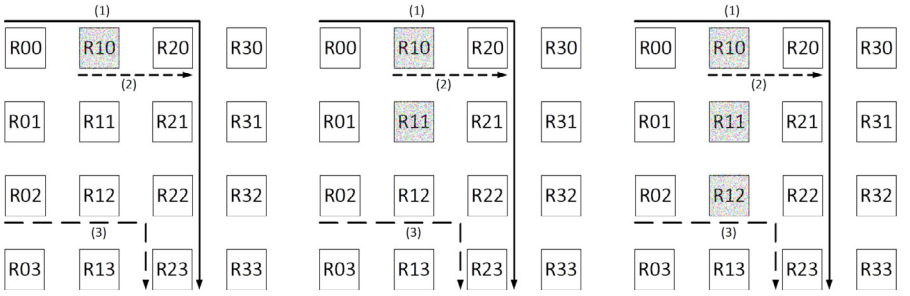
Experimental results We performed three experiments in a 4×4 ASPIN model. In each experiment we consider different scenarios each of which including different faulty nodes to measure the end-to-end latency. The traffic pattern of all scenarios of each experiment is identical. In the following we explain each experiment and the obtained results in more details.

In all experiments, the minimum and maximum expected latencies of packet (1) are reported. The expected latency shows the needed time for delivery of packet (1) to its destination. The minimum and maximum probabilities of reaching packet (1) to its destination are equal to one for all scenarios. In other words, there is no scenario in which packet (1) does not reach to its destination.

As shown in Fig. 13, in the first experiment, packet (1) is sent from R00 to R23 and there is no other packet in the network. In scenario 1, router R10 is faulty. In scenario 2, routers R10 and R11 are faulty, and in scenario 3, routers R10, R11 and R12 are faulty. The results are presented in Fig. 14, where the minimum expected latency is the same as the maximum expected latency for all scenarios. Scenario 3 has the highest expected latency since there are more faulty routers and the packet is rerouted more times in comparison to other scenarios. Also, scenario 1 has the least expected latency as there are less faulty nodes in the network.

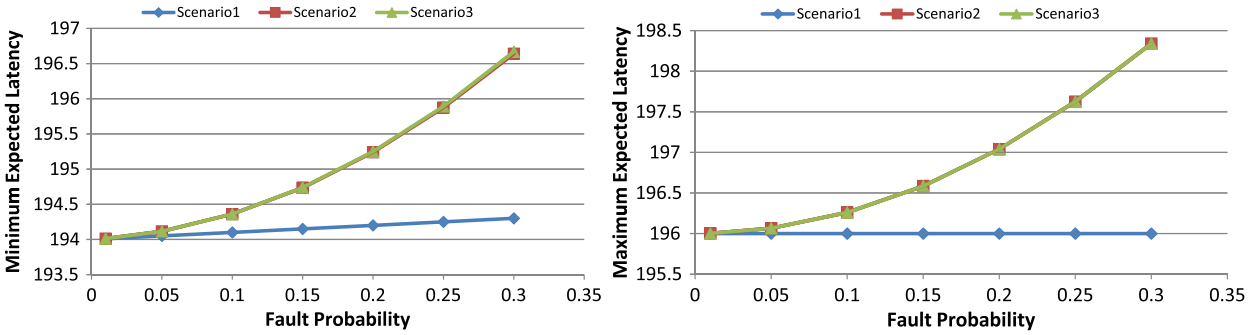
In the second experiment as shown in Fig. 15, router R10 generates packet (2) as soon as it receives packet (1), thus packet (2) may cause disruption to packet (1). On the other hand, R02 produces packet (3) in a way that it reaches R22 at the same time as packet (1), so packet (1) may be delayed by packet (3) too. In scenario 1, router R10 is faulty. In scenario 2, routers R10 and R11 are faulty, and in scenario 3 routers R10, R11 and R12 are faulty. The results are presented in Fig. 16. Scenario 3 has the highest and scenario 1 has the least expected latency. The reason is the same as the one explained for the first experiment.

In the third experiment, packet (1) is disrupted by packet (2), and packet (2) is itself disrupted because of congestion in R21. On the other hand, congestion in R23 leads packet (5) to be blocked until packet (4) leaves the input port of R22. This may results in disruption of packet (1) by packet (5), if they reach R32 at the same time (Fig. 17). In scenario 1, router R10 is faulty. In scenario 2, routers R10 and R11 are faulty. The results are presented in Fig. 18. In contrast to experiments 1 and 2, increasing the probability of fault decreases the maximum expected latency. The reason is that rerouting packet (1) from the normal path (i.e. R00 $\rightarrow$ R10 $\cdots$ $\rightarrow$ R30 $\rightarrow$ R31 $\cdots$ $\rightarrow$ R33) to an alternative path (i.e. R00 $\rightarrow$ R01 $\rightarrow$ R02 $\rightarrow$ R12 $\cdots$ $\rightarrow$ R32 $\rightarrow$ R33 in case of in scenario 2), avoids the congestion caused by packets (3), (5), and (7). So, the total latency decreases.



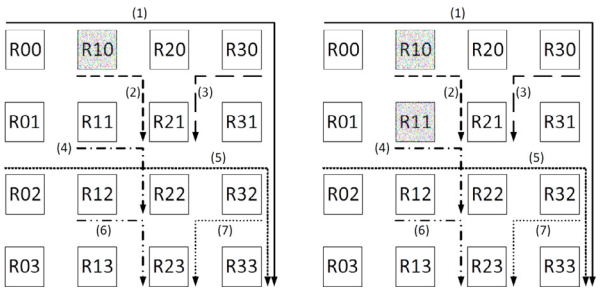
(a) R10 is faulty in scenario 1 (b) R10 and R11 are faulty in scenario 2 (c) R10, R11, and R12 are faulty in scenario 3

Fig. 15. The 4 × 4 ASPIN model: The traffic in experiment 2.



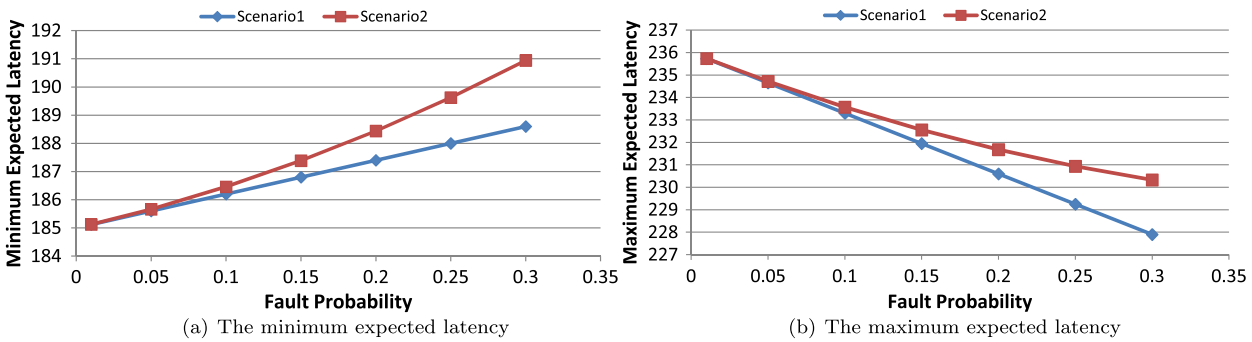
(a) The minimum expected latency (b) The maximum expected latency

Fig. 16. Experiment 2: the expected latency for different scenarios.



(a) R10 is faulty in scenario 1 (b) R10 and R11 are faulty in scenario 2

Fig. 17. The 4 × 4 ASPIN model: The traffic in experiment 3.



(a) The minimum expected latency (b) The maximum expected latency

Fig. 18. Experiment 3: the expected latency for different scenarios.

Table 1
The time and memory needed to evaluate different case studies with PRISM-based and IMCA-based approaches. NA means not available.

Problem		#states	#trans	Using PRISM		Using IMCA	
				time (sec)	memory	time (sec)	memory
NoC	**Exp 1-Scenario 1**	84	109	23.52	NA	0.000711	∼12.945 KB
	Exp 1-Scenario 2	484	909	167.392 + 0.02	NA	0.017655	∼79.07 KB
	Exp 1-Scenario 3	507	666	307.439 + 0.03	NA	0.02387	∼77.172 KB
	Exp 2-Scenario 1	342	379	161.835 + 0.03	NA	0.033238	∼51.187 KB
	Exp 2-Scenario 2	2184	3045	1031.11 + 0.17 (∼17 min)	NA	0.509681	∼337.226 KB
	Exp 2-Scenario 3	5220	9922	2955.611 + 0.94 (∼49 min)	NA	1.768932	∼857.344 KB
	Exp 3-Scenario 1	10032	15915	3228.434 + 0.94 (∼54 min)	NA	10.631929	∼1.561 MB
	Exp 3-Scenario 2	43290	71106	**crashed**	NA	136.916137	∼4.842 MB
Toxic Gas Sensing System	**1 sensor**	506	1170	222.787 + 0.01	NA	0.032514	∼87.383 KB

6.3. Comparing PRISM-based and IMCA-based approaches

In this section we investigate the time and memory necessary to run experiments using the two approaches based on PRISM and IMCA. Table 1 presents the results for different case studies with different sizes. The experiments of PRISM-based approach are run on a laptop with Windows 7, 4 GB RAM, and Intel Core i5-2430M CPU @2.4 GHz. To run the experiments of IMCA-based approach, Ubuntu 12.04.5 LTS is installed on the same laptop, but RAM is restricted to 1 GB. In the PRISM-based approach, the TMDP of a PTRebeca model is input to PRISM as a single MDP module. The MDP module is defined using the standard PRISM input language. The IMCA-based approach was explained at the beginning of this section.

As Table 1 shows, two numbers are reported for time when using the PRISM-based approach. The first one is the time needed via PRISM to construct the model, and the second one is the computation time to model check the model. The time for model construction is considerable, and this makes the approach inefficient even for small case studies like scenario 1 of experiment 2 in NoC case study (refer to `Exp 2-Scenario 1` in Table 1). In the IMCA-based approach, the Markov automaton of a PTRebeca model is input using a state-based language. So, the time for model construction is negligible, and the approach is efficient for large PTRebeca models like scenario 2 of experiment 3 in NoC case study (refer to `Exp 3-Scenario 2` in Table 1). PRISM crashed when trying to construct the model for `Exp 3-Scenario 2`. The needed memory for model checking is not reported by PRISM, so it's not available to be compared with the IMCA-based approach.

7. Related work

IMCA IMCA is a powerful model checker for analyzing interactive Markov chains (IMCs) and Markov automata (MA). IMCA has a state-based input language and lacks high-level programming constructs. Expected time and long-run average objectives, time-bounded probabilistic reachability and probabilistic reachability properties are supported for MA and IMC models [24,27].

In contrast to IMCA, PTRebeca provides high-level programming constructs and primary data structures, which makes modeling easier. In modeling we use the capabilities of PTRebeca, and in analysis we use IMCA for the evaluation of probabilistic timed properties. The semantics of PTRebeca is defined in TMDP. To be able to use IMCA, the TMDP of a PTRebeca model is converted to a Markov automaton. This conversion preserves all the above mentioned objectives except time-bounded reachability.

UPPAAL SMC In [39], authors introduce UPPAAL SMC in which systems are represented via networks of automata. In UPPAAL SMC, each component of the system is modeled with an automaton whose clocks can evolve with various rates. To provide efficient analysis of probabilistic properties, statistical model checking is used as a technique for fully stochastic models. The work supports modeling and performance analysis of systems with continuous time behaviors and dynamical features.

PTRebeca has a Java-like syntax which makes the language easy to use for practitioners. In PTRebeca time is discrete and discrete probability distributions are used to model probabilistic behaviors. In this work, we use the stochastic model checking algorithms for performance evaluation of systems via the IMCA model checker.

PRISM PRISM is a well-established and powerful model checker with a state-based input language. An input model of PRISM is composed of a number of modules which can share variables and interact with each other. PRISM is well equipped with theories and reduction techniques [23], but lacks high-level programming constructs like loops, and primary data structures like arrays, which makes modeling hard.

In contrast, PTRebeca provides high-level object-based programming features and asynchronous message passing, which makes modeling easier. In modeling we benefit from capabilities of PTRebeca, and in analysis we use the capabilities of the PRISM and the IMCA model checkers. As we showed earlier, using IMCA, we are able to model check larger PTRebeca models comparing to PRISM as the backend model checker.

Modest Modest [30] is a high-level and convenient language for describing stochastic timed and hybrid systems. It supports loop constructs, structs and arrays, exception handling, and other advanced programming constructs. It also supports various model checking approaches. For the probabilistic timed fragment of Modest, model checking can be performed using a digital time semantics [40] or by a direct mapping to probabilistic timed automata. Both approaches use PRISM as a backend model checker.

In contrast to Modest, PTRebeca supports object-based programming features, and follows the asynchronous message passing paradigm of actors, while Modest relies on synchronous message passing. In the conference paper we used PRISM for the analysis of systems, which is similar to Modest with respect to the analysis. In this work IMCA is used to overcome the problem of time-consuming model checking of large PTRebeca models.

ProbMela ProbMela is a probabilistic version of Promela [41]. The operational semantics of ProbMela is defined as an MDP [42]. In [43], ProbMela is used as input language for the MDP model checker LiQuor which provides qualitative and quantitative analysis of LTL properties. There is also a mapping from ProbMela to the PRISM language, which makes probabilistic analysis possible [44].

PTRebeca is an event-driven and actor-based language whereas ProbMela is process-based. Both languages are asynchronous in spirit. We proposed a semantics of PTRebeca as TMDP (or PTA with digital clocks), enabling the analysis of timing and probabilistic behaviors of asynchronous systems via PRISM. In this work, the TMDP obtained from a PTRebeca model is converted to a Markov automaton and the IMCA model checker can be used for the performance evaluation analysis.

PMaude PMaude extends standard rewriting theories of Maude with probability [45]. There is an actor extension of probabilistic rewriting theories for PMaude which removes non-determinism. A statistical technique is provided to analyze quantitative aspects of systems using discrete-event simulation. In comparison with PMaude, modeling asynchronous systems is more straightforward in PTRebeca language as it is an actor-based language. Also PTRebeca supports non-determinism in the model and there is no need to resolve it by assuming distribution on different choices of non-determinism. It is because of the probabilistic model checking facilities which are provided by PRISM and IMCA.

Actor languages Some work has been done on the development of actor frameworks based on familiar languages such as C/C++, Smalltalk, Python, Ruby, .NET and Java. To mention a few examples, Scala Actors library [6], Kilim [46], and ActorFoundry [47] are Java implementations of the actor model. More examples of actor frameworks for the above languages can be found in [48].

Comparing to the above actor-based programming languages, we are using a model-driven development approach in PTRebeca language. We can start with small models and use model checking and simulation to find possible correctness problems in our core algorithms, and also find how to improve the performance by changing some parameters while the code is still small, understandable, and easily manageable.

PCreol Creol is an object-oriented modeling language based on concurrent objects, communicating by asynchronous message passing [49]. PCreol is the probabilistic extension of Creol, oriented towards quantitative analysis [50]. PCreol is integrated with VeStA [51] which enables the statistical model checking and quantitative analysis of PCreol models. Using VeStA, the full state-space exploration is replaced by Monte Carlo simulation, controlled by means of statistical hypothesis testing.

To have more accurate results, probabilistic model checking is provided for PTRebeca models via PRISM and IMCA model checkers, which allows functional correctness and performance evaluation of PTRebeca models. Both languages are similar with respect to asynchronous message passing among concurrent objects.

Summary In PMaude, probability distribution functions (rates and stochastic functions) are provided for modeling probabilistic behaviors. Also, PMaude implements stochastic continuous-time. In ProbMela, probabilities are drawn from discrete probability distributions, and passage of time can be modeled using a timer process. Modest enables a direct high-level modelling of PTA and more complex models. In all aforementioned languages, non-deterministic behavior can be modeled. In analysis, PMaude resolves non-determinism, and uses statistical model checking to verify properties which results in inaccurate results. In the analysis of ProbMela and Modest, non-determinism is not resolved. Modest also provides the option of a digital clock semantics, which, just like we did in the conference paper, is handed over to PRISM for model checking.

Our focus in designing PTRebeca has been on ease of modeling and efficiency of analysis mainly for asynchronous applications. To this end, we use discrete time model and discrete probability distributions. These decisions showed to be effective in modeling different applications that we have targeted. Moreover, resolving non-determinism by a discrete probability distribution generates inaccurate estimations, so, we avoided that by choosing TMDP as the semantics of PTRebeca. We were able to formalize the advance of time in our model using a single integer-valued variable. The language design of PTRebeca and its analysis approach when using PRISM, is closest to the Modest approach, apart from the latter not being object-oriented and not being asynchronous by design.

We also converted the TMDP resulted from a PTRebeca model to a Markov automaton. This way, we are able to use the IMCA model checker for large PTRebeca models. In Markov automata delays are governed by exponential distributions while

in PTRebeca time is discrete. In the conversion, the rate in a Markovian transition of the Markov automaton is approximated by the integer value of the corresponding delay transition in the TMDP. To ensure the approximations are correct, we mathematically proved that expectation properties are preserved by this conversion.

8. Conclusion

In this paper we introduced the syntax and semantics of Probabilistic Timed Rebeca (PTRebeca) for modeling and verification of probabilistic real-time actor systems. Semantics of PTRebeca is presented in SOS rules. As the model of time in PTRebeca is discrete, we decided to use discrete-time MDP with an integer-valued time variable for the semantics of PTRebeca. PTRebeca models can thus be analyzed against PCTL, expected reachability, and probabilistic reachability properties.

In the conference paper [18], we used PRISM as the back-end model checker for performance evaluation of PTRebeca models. As the TMDP of a PTRebeca model is input as one MDP module to PRISM, only small models like ticket service can be input via PRISM input language. To support the modeling of larger PTRebeca models, we used the explicit engine of PRISM which works with an intermediate transition matrix representation. Using this method, we could analyze larger models like sensor network, but PRISM does not support all the features for this format. So, we could analyze models only against probabilistic reachability properties. To overcome this shortage, in this paper we examined parallel composition approach in which each PTRebeca component is converted to a probabilistic timed automaton. The parallel composition of all PTAs represent the model behavior. The resulting PTA can be input to PRISM for performance analysis. We showed that this approach creates larger state space comparing to the TMDP semantics. So, it is not efficient for performance analysis of PTRebeca models.

To provide probabilistic reachability and expected reachability properties for larger models, we proposed an approach in which the TMDP of a PTRebeca model is converted to one Markov automaton. The MA is input to IMCA model checker for performance evaluation. We developed a toolset for automatic mapping of the TMDP to one MA. We examined two case studies to show the applicability of our approach. The toxic gas sensing system, called sensor network previously, was examined in the conference paper using explicit engine of PRISM. Here, we obtained the identical results via mapping the TMDP to one MA and using IMCA model checker, but in less amount of time. We also modeled a case study of NoC network using PTRebeca, and evaluated the expected time properties by using the developed toolset.

In addition to the benefits of using TMDP semantics for analysis of PTRebeca models, our technique is based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. Hence, the proposed semantics is general enough to be applied to similar computation models where there is message-driven communication and autonomous objects as units of concurrency, and there exists discrete probabilistic behaviors in the model such as agent-based systems.

Acknowledgement

The work on this paper was supported by the project “Timed Asynchronous Reactive Objects in Distributed Systems: TARO” (nr. 110020021) of the Icelandic Research Fund.

Appendix A. Proof of Theorem 1

We show that

$$L(eT^{min}(s, \Diamond G)) = eT^{min}(s, \Diamond G) \tag{A.1}$$

for all $s \in S$. To this aim, we distinguish three cases which are $s \in DS \setminus G$, $s \in PS \setminus G$, and $s \in G$.

- in case of $s \in DS \setminus G$, the left-hand side of (A.1) is:

$$L(eT^{min}(s, \Diamond G)) = d_s + eT^{min}(t, \Diamond G), \tag{A.2}$$

where d_s is delay time reaching state t from state s. This delay time is deterministic. On the other hand,

$$eT^{min}(t, \Diamond G) = \inf_D \mathbb{E}_{t,D}(V_G) = \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{t,D}(\pi) \tag{A.3}$$

Combining (A.2) and (A.3), we have

$$\begin{aligned} L(eT^{min}(s, \Diamond G)) &= d_s + \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{t,D}(\pi) \\ &= \inf_D \sum_{Paths} (V_G(\pi) + d_s) \cdot Pr_{t,D}(\pi) \\ &= \inf_D \sum_{Paths} (V_G(\pi)) \cdot Pr_{s,D}(\pi) \\ &= eT^{min}(s, \Diamond G) \end{aligned} \tag{A.4}$$

Note that in (A.4), in the second line, paths, start from t, whereas, in the third line, paths start from s.

- in case of $s \in PS \setminus G$ there is:

$$\begin{aligned}
eT^{min}(s, \Diamond G) &= \inf_D \mathbb{E}_{s,D}(V_G) = \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{s,D}(\pi) \\
&= \inf_D \sum_{s \xrightarrow{\alpha,\mu,0} t} D(s)(\alpha) \cdot \mathbb{E}_{t, D(s \xrightarrow{\alpha,\mu,0} \cdot)}(V_G) \\
&= \inf_D \min_{s \xrightarrow{\alpha} \mu^s_\alpha} \sum_{t \in S} \mu^s_\alpha(t) \cdot \mathbb{E}_{t, D(s \xrightarrow{\alpha,\mu,0} \cdot)}(V_G) \\
&= \min_{s \xrightarrow{\alpha} \mu^s_\alpha} \inf_D \sum_{t \in S} \mu^s_\alpha(t) \cdot \mathbb{E}_{t, D(s \xrightarrow{\alpha,\mu,0} \cdot)}(V_G) \\
&= \min_{s \xrightarrow{\alpha} \mu^s_\alpha} \inf_D \sum_{t \in S} \mu^s_\alpha(t) \cdot \mathbb{E}_{t,D}(V_G) \\
&= \min_{s \xrightarrow{\alpha} \mu^s_\alpha} \sum_{t \in S} \mu^s_\alpha(t) \cdot eT^{min}(s, \Diamond G) \\
&= \min_{\alpha \in Act(s)} \sum_{t \in S} \mu^s_\alpha(t) \cdot eT^{min}(s, \Diamond G) \\
&= L(eT^{min}(s, \Diamond G))
\end{aligned}$$

- in case of $s \in G$, based on the definition there is $eT^{min}(s, \Diamond G) = \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{s,D}(\pi) = 0$, which is the same as the value of the Bellman operator for goal states.

Appendix B. Proof of Theorem 2

From [33], $cR^{min}(s, \Diamond G)$ is the unique fixpoint of the bellman operator L' defined as

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s,\alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s,\alpha,s').v(s') + \sum_{s' \in G} \mathbf{P}(s,\alpha,s').g(s') \right\}. \tag{B.1}$$

Now we show that the Bellman operator L defined in Theorem 1, and the Bellman operator L' defined in (B.1) for $ssp_{et}(\mathcal{M})$ are the same. By Definition 8, for each $s \in S$, $g(s) = 0$, therefore

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s,\alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s,\alpha,s').v(s') \right\}. \tag{B.2}$$

Consider three cases, $s \in DS \setminus G$, $s \in PS \setminus G$ and $s \in G$.

- *Case (I)*: Assume $s \in DS \setminus G$, by Definition 8, $c(s,\alpha) = d_s$ and

$$\mathbf{P}(s,\alpha,s') = \begin{cases} 1 & s' \text{is reaching state from } s \text{ by delay } d_s \text{ and } \alpha = \perp, \\ 0 & \text{otherwise} \end{cases}$$

Only action belongs to $Act(s)$ is $\perp$, furthermore only state after s is state s', thus, from (B.2),

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s,\alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s,\alpha,s').v(s') \right\} = d_s + v(s'),$$

where s' is deterministic reaching state from s by delay time d_s. However from Theorem 1, $d_s + v(s') = [L(v)](s)$, for each $s \in DS \setminus G$, so in this case, theorem is proved.

- *Case (II)*: Assume $s \in PS \setminus G$. By Definition 8, $\mathbf{P}(s,\alpha,s') = \mu^s_\alpha(s')$ and $c(s,\alpha) = 0$. Therefore,

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s,\alpha) + \sum_{s' \in S} \mathbf{P}(s,\alpha,s').v(s') \right\} = \min \left\{ \sum_{s' \in S} \mu^s_\alpha(s').v(s') \right\}$$

However in this case

$$[L(v)](s) = \min\left\{\sum_{s'\in S} \mu^s_\alpha(s').v(s')\right\},$$

Therefore $[L'(v)](s) = [L(v)](s)$, and the proof of the theorem in this case is complete.

- *Case (III)*: Assume $s \in G$. Here in addition to $g(s) = 0$, we have $c(s,\alpha) = 0$, for each action α, and $\mathbf{P}(s,\alpha,s') = 0$, for each $\alpha \in Act(s)$ and $s' \in S$. Therefore

$$[L'(v)](s) = \min_{\alpha\in Act(s)}\left\{c(s,\alpha) + \sum_{s'\in S\setminus G} \mathbf{P}(s,\alpha,s').v(s')\right\} = 0 = [L(v)](s).$$

Now the proof is complete.

Appendix C. Proof of Theorem 3

We show that

$$L(eR^{min}(s,\Diamond G)) = eR^{min}(s,\Diamond G) \tag{C.1}$$

for all $s \in S$. To this aim, we distinguish three cases which are $s \in DS \setminus G$, $s \in PS \setminus G$, and $s \in G$.

- in case of $s \in DS \setminus G$, the left-hand side of (C.1) is:

$$L(eR^{min}(s,\Diamond G)) = \rho(s) \times d_s + eR^{min}(t,\Diamond G), \tag{C.2}$$

where $\rho(s)$ is the reward of staying in s. This reward is deterministic. On the other hand,

$$eR^{min}(t,\Diamond G) = \inf_D \mathbb{E}_{t,D}(R_G) = \inf_D \sum_{Paths} R_G(\pi)\cdot Pr_{t,D}(\pi) \tag{C.3}$$

Combining (C.2) and (C.3), we have

$$\begin{aligned} L(eR^{min}(s,\Diamond G)) &= \rho(s)\times d_s + \inf_D \sum_{Paths} R_G(\pi)\cdot Pr_{t,D}(\pi)\\ &= \inf_D \sum_{Paths} (R_G(\pi) + \rho(s)\times d_s)\cdot Pr_{t,D}(\pi)\\ &= \inf_D \sum_{Paths} (R_G(\pi))\cdot Pr_{s,D}(\pi)\\ &= eR^{min}(s,\Diamond G) \end{aligned} \tag{C.4}$$

Note that in (C.4), in the second line, paths, start from t, whereas, in the third line, paths start from s.

- in case of $s \in PS \setminus G$ there is:

$$\begin{aligned} eR^{min}(s,\Diamond G) &= \inf_D \mathbb{E}_{s,D}(R_G) = \inf_D \sum_{Paths} R_G(\pi)\cdot Pr_{s,D}(\pi)\\ &= \inf_D \sum_{s \xrightarrow{\alpha,\mu,0} t} D(s)(\alpha)\cdot \mathbb{E}_{t,D(s\xrightarrow{\alpha,\mu,0}\cdot)}(R_G) + r(s,\alpha)\\ &= \inf_D \min_{s\xrightarrow{\alpha}\mu^s_\alpha} \sum_{t\in S} \mu^s_\alpha(t)\cdot \mathbb{E}_{t,D(s\xrightarrow{\alpha,\mu,0}\cdot)}(R_G) + r(s,\alpha)\\ &= \inf_D \min_{s\xrightarrow{\alpha}\mu^s_\alpha} r(s,\alpha) + \sum_{t\in S} \mu^s_\alpha(t)\cdot \mathbb{E}_{t,D(s\xrightarrow{\alpha,\mu,0}\cdot)}(R_G)\\ &= \min_{s\xrightarrow{\alpha}\mu^s_\alpha} \inf_D r(s,\alpha) + \sum_{t\in S} \mu^s_\alpha(t)\cdot \mathbb{E}_{t,D(s\xrightarrow{\alpha,\mu,0}\cdot)}(R_G)\\ &= \min_{s\xrightarrow{\alpha}\mu^s_\alpha} \inf_D r(s,\alpha) + \sum_{t\in S} \mu^s_\alpha(t)\cdot \mathbb{E}_{t,D}(R_G)\\ &= \min_{s\xrightarrow{\alpha}\mu^s_\alpha} r(s,\alpha) + \sum_{t\in S} \mu^s_\alpha(t)\cdot eR^{min}(s,\Diamond G) \end{aligned}$$

$$= \min_{\alpha \in Act(s)} r(s,\alpha) + \sum_{t \in S} \mu_\alpha^s(t) \cdot eR^{min}(s, \Diamond G)$$
$$= L(eR^{min}(s, \Diamond G))$$

- in case of $s \in G$, based on the definition there is $eR^{min}(s, \Diamond G) = \inf_D \sum_{Paths} R_G(\pi) \cdot Pr_{s,D}(\pi) = 0$, which is the same as the value of the Bellman operator for goal states.

References

[1] C. Hewitt, Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot, MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT, Apr. 1972.
[2] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1990.
[3] G. Agha, I. Mason, S. Smith, C. Talcott, A foundation for actor computation, J. Funct. Program. 7 (1997) 1–72.
[4] G. Agha, The structure and semantics of actor languages, in: REX Workshop, 1990, pp. 1–59.
[5] Erlang, Erlang programming language homepage, http://www.erlang.org.
[6] P. Haller, M. Odersky, Actors that unify threads and events, in: Coordination Models and Languages, Springer, 2007, pp. 171–190.
[7] Naos, The Naos engine homepage, http://www.naos-engine.com.
[8] M. Sirjani, A. Movaghar, An actor-based model for formal modelling of reactive systems: Rebeca, Tech. Rep. CS-TR-80-01, Tehran, Iran, 2001.
[9] M. Sirjani, A. Movaghar, A. Shali, F. de Boer, Modeling and verification of reactive systems using Rebeca, Fundam. Inform. 63 (4) (Dec. 2004) 385–410.
[10] M. Sirjani, M.M. Jaghoori, Ten years of analyzing actors: Rebeca experience, in: Formal Modeling: Actors, Open Systems, Biological Systems, 2011, pp. 20–56.
[11] L. Aceto, M. Cimini, A. Ingólfsdóttir, A.H. Reynisson, S.H. Sigurdarson, M. Sirjani, Modelling and simulation of asynchronous real-time systems using Timed Rebeca, in: FOCLASA'11, 2011, pp. 1–19.
[12] E. Khamespanah, Z. Sabahi Kaviani, M. Sirjani, R. Khosravi, M.-J. Izadi, Timed Rebeca schedulability and deadlock freedom analysis using bounded floating-time transition system, in: Journal of Science of Computer Programming, 2014.
[13] Z. Sharifi, M. Mosaffa, S. Mohammadi, M. Sirjani, Functional and performance analysis of network-on-chips using actor-based modeling and formal verification, in: Proceedings of AVOCS'13, 2013.
[14] Z. Sharifi, S. Mohammadi, M. Sirjani, Comparison of NoC routing algorithms using formal methods, in: Proceedings of PDPTA'13, 2013.
[15] Ehsan Khamespanah, Kirill Mechitov, Marjan Sirjani, Gul Agha, Schedulability analysis of distributed real-time sensor network applications using actor-based model checking, in: Proceedings of SPIN'16, 2016.
[16] L. Linderman, K. Mechitov, B.F. Spencer, TinyOS-based real-time wireless data acquisition framework for structural health monitoring and control, Struct. Control Health Monit. 20 (6) (June 2013) 1007–1020.
[17] M. Varshosaz, R. Khosravi, Modeling and verification of probabilistic actor systems using prebeca, in: Proceedings of the 14th International Conference on Formal Engineering Methods, ICFEM'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 135–150.
[18] A. Jafari, E. Khamespanah, M. Sirjani, H. Hermanns, Performance analysis of distributed and asynchronous systems using probabilistic timed actors, in: Proceedings of AVoCS'14, 2014.
[19] M. Kwiatkowska, G. Norman, D. Parker, J. Sproston, Performance analysis of probabilistic timed automata using digital clocks, Form. Methods Syst. Des. 29 (2006) 33–78.
[20] N. Coste, H. Hermanns, E. Lantreibecq, W. Serwe, Towards performance prediction of compositional models in industrial GALS designs, in: Proceedings of the 21st International Conference on Computer Aided Verification, CAV'09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 204–218.
[21] C. Baier, B.R. Haverkort, H. Hermanns, J.-P. Katoen, Performance evaluation and model checking join forces, Commun. ACM 53 (9) (2010) 76–85.
[22] G.D. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sep. 1981.
[23] A. Hinton, M.Z. Kwiatkowska, G. Norman, D. Parker PRISM, A tool for automatic verification of probabilistic systems, in: Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06, in: Lect. Notes Comput. Sci., Springer-Verlag, 2006, pp. 441–444.
[24] D. Guck, T. Han, J.-P. Katoen, M.R. Neuhäußer, Quantitative timed analysis of interactive Markov chains, in: Proceedings of the 4th International Conference on NASA Formal Methods, NFM'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 8–23.
[25] C. Eisentraut, H. Hermanns, L. Zhang, On probabilistic automata in continuous time, in: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11–14 July 2010, Edinburgh, United Kingdom, 2010, pp. 342–351.
[26] H. Hermanns, Interactive Markov Chains: And the Quest for Quantified Quality, Springer-Verlag, Berlin, Heidelberg, 2002.
[27] D. Guck, H. Hatefi, H. Hermanns, J. Katoen, M. Timmer, Modelling, reduction and analysis of Markov automata, in: Quantitative Evaluation of Systems – Proceedings of the 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27–30, 2013, pp. 55–71.
[28] C. Derman, Finite State Markovian Decision Processes, Academic Press, Inc., Orlando, FL, USA, 1970.
[29] R. Segala, Modeling and verification of randomized distributed real-time systems, Ph.D. thesis, Cambridge, MA, USA, 1995.
[30] E.M. Hahn, A. Hartmanns, H. Hermanns, J.-P. Katoen, A compositional modelling and analysis framework for stochastic hybrid systems, Form. Methods Syst. Des. 43 (2) (2013) 191–232.
[31] L. Lamport, Real-time model checking is really simple, in: Proceedings of the 13 IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods, CHARME'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 162–175.
[32] D. Guck, H. Hatefi, H. Hermanns, J. Katoen, M. Timmer, Analysis of timed and long-run objectives for Markov automata, Log. Methods Comput. Sci. 10 (3) (2014).
[33] D.P. Bertsekas, J.N. Tsitsiklis, An analysis of stochastic shortest path problems, Math. Oper. Res. 16 (3) (1991) 580–595.
[34] L. de Alfaro, Computing minimum and maximum reachability times in probabilistic systems, in: Concurrency Theory, Proceedings of the 10th International Conference, CONCUR'99, Eindhoven, The Netherlands, August 24–27, 1999, pp. 66–81.
[35] D. Guck, M. Timmer, H. Hatefi, E. Ruijters, M. Stoelinga, Modelling and analysis of Markov reward automata, in: Automated Technology for Verification and Analysis – Proceedings of the 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3–7, 2014, pp. 168–184.
[36] Rebeca, Rebeca homepage, http://www.rebeca-lang.org.
[37] International technology roadmap for semiconductors – ITRS, http://www.manmaker.com/manual/, 2011.
[38] A. Hosseini, T. Ragheb, Y. Massoud, A fault-aware dynamic routing algorithm for on-chip networks, in: International Symposium on Circuits and Systems, ISCAS 2008, 18–21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA, 2008, pp. 2653–2656.
[39] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, Uppaal SMC tutorial, Int. J. Softw. Tools Technol. Transf. 17 (4) (2015) 397–415.

[40] A. Hartmanns, H. Hermanns, A modest approach to checking probabilistic timed automata, in: QEST, IEEE Computer Society, 2009, pp. 187–196.
[41] G.J. Holzmann, The model checker SPIN, Softw. Eng. J. 23 (5) (1997) 279–295.
[42] C. Baier, F. Ciesinski, M. Groesser, Probmela: a modeling language for communicating probabilistic processes, 2004.
[43] F. Ciesinski, C. Baier, LiQuor: a tool for qualitative and quantitative linear time analysis of reactive systems, in: Proc. 3rd International Conference on Quantitative Evaluation of Systems, QEST'06, IEEE CS Press, 2006, pp. 131–132.
[44] F. Ciesinski, C. Baier, M. Groesser, D. Parker, Generating compact mtbdd-representations from probmela specifications, in: Proceedings of the 15th International Workshop on Model Checking Software, SPIN'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 60–76.
[45] G. Agha, J. Meseguer, K. Sen, PMaude: rewrite-based specification language for probabilistic object systems, Electron. Notes Theor. Comput. Sci. 153 (2) (2006) 213–239.
[46] S. Srinivasan, A. Mycroft, Kilim: isolation-typed actors for Java, in: ECOOP 2008 – Object-Oriented Programming, Springer, 2008, pp. 104–128.
[47] M. Astley, The Actor Foundry: A Java-based Actor Programming Environment, University of Illinois at Urbana-Champaign: Open Systems Laboratory.
[48] R.K. Karmani, A. Shali, G. Agha, Actor frameworks for the JVM platform: a comparative analysis, in: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM, 2009, pp. 11–20.
[49] E.B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Softw. Syst. Model. 6 (1) (2007) 39–58.
[50] L. Bentea, O. Owe, A probabilistic framework for object-oriented modeling and analysis of distributed systems, in: Formal Verification of Object-Oriented Software, Springer, 2012, pp. 105–122.
[51] K. Sen, M. Viswanathan, G.A. Agha, VESTA: a statistical model-checker and analyzer for probabilistic systems, in: 2nd International Conference on the Quantitative Evaluaiton of Systems, QEST 2005, 19–22 September, IEEE Computer Society, Torino, Italy, 2005, pp. 251–252.