Modeling and Verification of Probabilistic Actor Systems using pRebeca

Mahsa Varshosaz and Ramtin Khosravi

School of Electrical and Computer Engineering,
College of Engineering, University of Tehran, Tehran, Iran
`{m.varshosaz,rkhosravi}@ece.ut.ac.ir`

Abstract. *Quantitative verification has gained an increasing attention as a promising approach for analysis of systems in various domains, especially for distributed systems, where the uncertainties of the environment cause the system to exhibit probabilistic and nondeterministic behavior. In this paper, we introduce pRebeca, an extension to the high-level actor-based modeling language Rebeca, that is used to model distributed and reactive systems with probabilistic and nondeterministic nature. We propose a simple syntax suitable for describing different aspects of a probabilistic system behavior and provide a formal semantics based on Markov decision processes. To model check a pRebeca model, it is converted to a Markov decision process and verified using the PRISM model checker against PCTL properties. Using a couple of examples, we show how a probabilistic system can be expressed in pRebeca in a simple way, while taking advantage of the PRISM model checker features.*

Keywords: probabilistic model checking, actor model, pRebeca, Rebeca

1 Introduction

As a broad range of computer systems exhibit probabilistic and nondeterministic behavior, there has been an increasing interest in employing and developing quantitative verification techniques to analyze and evaluate various properties of such systems. Distributed systems have been among motivating domains gaining much attention for application of quantitative techniques. The widespread and rapid growth of distributed systems such as sensor networks, web-based systems, etc., make it worth to analyze various quantitative and qualitative properties about them. As a starting point of analysis, modeling of such systems could take some effort due to structural and behavioral complexities. Thus, using a computational model compatible with the domain can effectively reduce this effort by expressing the concepts and entities of the domain directly. A modeling language based on the proper computational model, avoids putting extra effort to model the basic computations by providing proper primitives and level of abstraction, making the model more readable and maintainable. To this end, we introduce pRebeca which is an extension to the Rebeca (Reactive Objects

Language) [1] language. It is a high-level object-based modeling language based on the actor model, suitable for describing asynchronous distributed systems with probabilistic behavior.

The actor model is one of the pioneering computing models for concurrent and asynchronous distributed systems. An actor model consists of a number of actors, which are the universal primitives in this model, communicating via asynchronous message passing. Since the introduction of actor as an agent-based language by Hewitt [2] and its development by Agha [3, 4] to a concurrent object based language, various interpretations and extensions of this model have been developed and are widely used in theory and practice [5–7]. Rebeca is an operational interpretation of actor model in the form of a high-level modeling language with simple Java-like syntax, formal semantics, and model checking tools which makes it proper and easy to use especially for asynchronous distributed systems. In a Rebeca model, rebecs (reactive objects) are actors, communicating via asynchronous message passing. Each rebec can perform computation on its local variables, send messages or create new rebecs in response to a received message.

In spite of widespread use of the actor model in both industry and academia, there has been little work done on modeling and verification of actor-based systems with probabilistic features. To the best of our knowledge, pRebeca is the first actor based modeling language with probabilistic modeling features. The only related work is PMAUDE [8] which is a specification language not based on the actor model but implementing actors with distribution probabilities on time of message passing and computation, without nondeterminism [8]. We extended Rebeca to make it capable of modeling actor-based systems with probabilistic and nondeterministic behavior. The simple and easy to learn syntax of pRebeca, makes it a suitable modeling language in practice. In a nutshell, in this work we present:

- Support for modeling probabilistic actor systems, via language constructs for:
  - Probabilistic choice between alternative behavior
  - Unreliable message passing between actors
- pRebeca syntax as an extension to the Rebeca syntax
- pRebeca model semantics based on Markov decision processes (MDPs)
- A method to model check pRebeca models by means of the PRISM model checker [9]

To verify pRebeca models, we use a two step method. In the first step pRebeca model is converted to an MDP by means of the state-space generation engine of Afra [10], a Rebeca model checker. In the second phase, the resulting MDP is expressed in the PRISM language and Probabilistic Computation Tree Logic (PCTL) properties can be checked against it. Hence, modelers using pRebeca can take advantage of using a high-level modeling language along with using PRISM features as a powerful model checking tool to verify their models.

The rest of this paper is organized as follows. Section 2 contains the related work and some explanation on differentiating points of this work from existing ones. Some preliminary definitions and concepts are explained in section 3. Section 4 consists of explanations of the syntax and semantics of pRebeca. Section 5 presents the method proposed to model checking pRebeca models and also includes a case study, and the work is concluded in section 6.

2 Related Work

There have been a number of process algebras [11–17], modeling languages, and model checking tools [18, 9, 19, 20] introduced so far to model and model check probabilistic systems. PRISM [9] is a well-known powerful probabilistic model checker with a guarded-command input language. A PRISM model is composed of reactive modules which can interact with each other. Although PRISM is a powerful model checker, the PRISM language does not support high-level constructs such as conditional or loop statements and just has a few primary data types, thus it is not suitable for high-level modeling of systems. ProbVerus [21] is another probabilistic model checker which is an extension to Verus verifier and implements symbolic techniques for PCTL model checking. ProbVerus does not support nondeterminism in its models. The ProbVerus language provides constructs such as assignment, conditional and loop statements and a probabilistic choice as a probabilistic feature and its semantics is based on Markov chains. PMAUDE is a specification language based on probabilistic rewriting theories with tool support for discrete event simulation and statistical verification. There is an implementation of actor modules in PMAUDE, in which the probabilistic distribution of time for message passing and computation is considered. The nondeterminism have been removed from the models for the sake of statistical model checking. Probmela [22] is a variant of Promela modeling language with tool support for quantitative Linear Temporal Logic (LTL) model checking. There has been also a direct mapping from Probmela to PRISM language. The Probmela Language constructs are close to our work in which there are high-level constructs such as conditionals, loops, probabilistic choices and primitives to model unreliable message passing.

The main difference of our work with others is that we extended an object based modeling language which is based on actor model and containing concepts such as classes and methods in Java-like style. pRebeca provides high-level constructs such as loops, conditional statements, assignment and also probabilistic primitives to model probabilistic choice between alternative behavior and unreliable message passing. All these facilities make pRebeca a suitable language for modeling asynchronous distributed systems. The method that we use for model checking keeps generated PRISM models in a rational size. This method prevents generation of models with extra overhead which may be resulted by direct mapping between two languages.

3 Preliminaries

3.1 Rebeca

Rebeca is a high-level modeling language based on actor model. The behavior of a Rebeca model is resulted from fairly interleaving of some self contained, reactive objects running on separate threads. Rebecs communicate only via asynchronous message passing and they have no shared state. Each rebec has an unbounded FIFO queue called its message queue to automatically receive messages. In Rebeca, rebecs are instances of reactive classes.

The structure of a Rebeca model consists of reactive class definitions and a "main" block. Each reactive class takes an integer value as an upper bound to its message queue which is used for model checking. There are three main parts in a reactive class:

– Definition of known rebecs denoting rebecs that can be receivers of the sending messages. Known rebecs define the infrastructure network topology.
– Definition of state variables that can be manipulated along processing of messages. State variables can be typed as integer, character, boolean or array.
– Definition of message servers which play the role of methods of the reactive class. The body of a message server contains a set of local variables and a sequence of statements. Messages in a Rebeca model are calls to message servers in the form of r.m(params) where r denotes one of the known rebecs of the sender, or **self** in case of sending a message to self. The message name m determines the message server of r that should be executed with the parameter list params. The statements in the body of a message server can be loops, conditional statements, assignment and (non-blocking) message send statements. Each rebec should have an **initial** message server which is invoked at the beginning of the model running. The execution of message servers is atomic. Thus, a computation step in a Rebeca model is the execution of a message server.

The main block in a Rebeca model consist of declaration of rebecs as the instances of reactive classes and binding of the declared rebecs to their known rebecs. Also, some parameter values can be sent to the initial message server of each rebec. At the beginning of the running of a model, instances of reactive classes are made and an invocation to the initial message server of each rebec is put into its message queue. Then, one of the rebecs is selected nondeterministically and the message at the head of its queue is popped out and executed and after that another rebec takes the turn and so on.

3.2 Markov Decision Process

In oreder to explain the semantics of pRebeca based on a Markov decision process we need to explain some definitions and concepts related to this structure. All the notations are adopted from [23].

Definition 1. *A distribution function over a set of states S can be denoted by ρ: $S \to [0,1]$ where $\sum_{s \in S} \rho(s)=1$. The set of all possible distribution functions over S is denoted by Distr(S).*

Definition 2. *A Markov decision process can be defined by a tuple $M=(S,S_{init},Act,\tau,AP,L)$ where:*

- *S is a set of states.*
- *S_{init} is the set of initial states.*
- *Act is the set of actions.*
- *$\tau \subseteq S \times Act \times Distr(S)$ is the probabilistic transition relation.*
- *AP is the set of atomic propositions.*
- *$L \subseteq S \times 2^{AP}$ is the labeling function.*

In each state s, transitions take place by first nondeterministically choosing an action $a \in Act(s)$ where $Act(s)$ denotes the set of possible actions in state s. Then, there is a corresponding transition distribution function which determines the probability of the transition to other states in S. The transition relation $s \to^a \rho$ denotes that ρ is the distribution function in state s by selecting action a (simply, $\rho_{s,a}$ denotes the transition distribution function in state s by selecting action a). All $s^{'} \in S$ for which $\rho(s^{'})>0$ are called a-successors of s. A transition such as $s \to^a \rho_t^1$ denotes that the only a-successor of s is t and the probability of transition to t is one and this relation can be viewed as $s \to^a t$ in a Labeled Transition System.

4 pRebeca

In this section, we explain the syntax and semantics of pRebeca and provide an example in order to better understand the language constructs.

4.1 Syntax

The syntax of pRebeca is an extension to the syntax of the Rebeca language. In order to provide a concise syntax, we investigated possible probabilistic aspects that could exist in an actor based system. In the actor model, actors are executed independently and communicate just via asynchronous message passing. Thus, the probabilistic features in the actor model can be considered as follows:

- An actor can exhibit different alternative behaviors with some probability. Such as a node in the network which can behave differently in each of the safe or failure states.
- Messages can be lost being sent via unreliable communication media. Unreliability is a usual feature of communication media in the real world networks that should be considered in modeling.

To address the mentioned probabilistic features, we extended the Rebeca syntax by adding the following features.

- *Probabilistic alternative behavior:*
  pAlt{
  prob(p_1):{statement;*}
  $\vdots$
  prob(p_n):{statement;*}
  }
  where $p_i \in [0,1]$, $i \in \{1,..., n\}$.
  A pAlt statement denotes probabilistic choice between alternative behavior. In a pAlt structure, each block of statements may be executed by the mentioned probabilities. If the sum of the probabilities is less than one, the remaining probability indicates doing nothing with probability $p_{default}$=1-$\sum_{i=1}^{n} p_i$. As an example, with execution of the following structure:
  pAlt {prob(0.3): {x=x+1} prob(0.6): {x=x-1;}}, the value of x may be incremented by one with probability 0.3 or it would be decremented by one with probability 0.6 and remain unchanged with probability 0.1.

- *Probabilistic assignment:*
  x=?(p_1:v_1,..., p_n:v_n), where $p_1+\ldots+p_n=1$ and v_1,..., v_n are possible values for the variable x.
  A probabilistic assignment denotes assigning values v_1,..., v_n to x by respective probabilities p_1,..., p_n. As an example, the statement x=?(0.2:5, 0.8:2); assigns values 5 and 2 to x with probabilities 0.2 and 0.8 respectively. This primitive can be implemented by means of a **pAlt** construct but is included for the sake of simplicity, as a syntactic sugar.

- *Probabilistic message sending:*
  r.m() **probloss**(p)
  This statement denotes sending messages with loss probability of p. This feature is added in order to model unreliable communications. As an example, the execution of r.m()probloss(0.2) may result in adding a message to the message queue of rebec r with probability 0.8 and no changes would happen with probability 0.2.

The grammar of pRebeca is presented in Figure 1. Complete explanations on the syntax of the Rebeca language can be found in [1].

As client-server architecture is one of the common architectures used for distributed systems, we choose an unreliable client server system as a running example. In this model, there are two reactive classes Client and Server. Three instances of Client and one instance of Server are declared. All clients can send requests to the server and do not communicate directly. Instances of Server have two operating modes: *safe* mode in which the server exhibits its expected functionality and *failure* mode in which the server is unable to respond to the requests correctly. A client exhibits probabilistic behavior in the way that it

Model::= reactiveClass* Main
reactiveClass::= **reactiveclass** C(I){knownRebecs stateVars msgSrv*}
Main::= {rebecDcl*}
knownRebecs::= **knownrebecs**{krDcl*}
msgSrv::= **msgsrv**(<T v>*){stmt*}
stateVars::= **statevars**{varDcl*}
varDcl::= T <v>$^+$;
krDCL::= C c;
stmt::= v=e;|r=new C(<e>*);| call;| conditional| pAlt| pAssignment
call::= r.M(<e>*)|r.M(<e>*)**lossprob(p)**
conditinal::= if(e){stmt$^+$}| if(e){stmt$^+$} else{stmt$^+$}
rebecDcl::= C r(<r>*):(<c>*);
pAlt::= **pAlt**{prob(p$_1$){stmt$^+$ },..., prob(p$_n$){stmt$^+$}}
pAssignment::= v=?(<p$_i$:c>$^+$);

Fig. 1. BNF grammar for pRebeca language. Superscript + denotes repetition of one or more times and superscript * denotes zero or more times repetition. Using angle brackets with repetition denotes comma separated lists. The Symbols C, T, I, v, c, r, m, and e denote reactive class, type, constant, variable, class name, rebec, message server, and expression respectively. Expressions in pRebeca are the same as expressions in Java.

chooses one of the existing client identifiers probabilistically and then, sends a message containing the selected id to the server. The number of repetitions of this message sending is determined via another probabilistic choice. Clients ignore the received messages which have different receiver ids. The server checks the id in the message and if in safe mode, sends the message to the corresponding id. If it is in the failure mode, it either ignores the message or sends it to a wrong client. The communication media in this system is lossy. This example does not describe any specific and realistic system and is just used for the sake of better representing the syntax and semantics of the language. The pRebeca model of unreliable client-server system is presented in Figure 2.

4.2 Semantics

Before explaining the semantics of a pRebeca model we need to define some notations and basic given sets.
In a pRebeca model P, we assume to have the following sets.

- Id_P is the set of rebec identifiers.
- Val_P is the set of all possible values for all state variables.
- MS_P is the set of all message servers in the model.

```
reactiveclass Server() {
  knownrebecs { Client cl1,cl2,cl3; }
  statevars { boolean mode; }
  msgsrv initial() {}
 msgsrv takeRequest(int Id) {
  mode=false;
  pAlt{
   prob(0.7){ //server in safe mode
      mode=true;
       if(Id==1)
            cl1.receive(Id)probloss(0.3);
       if(Id==2)
            cl2.receive(Id)probloss(0.2);
       if(Id==3)
            cl3.receive(Id) probloss(0.4);}
    prob(0.2){ //server in failure mode
      if(Id==1){
            cl2.receive(Id)probloss(0.2);
            cl3.receive(Id)probloss(0.4);}
      if(Id==2){
            cl1.receive(Id)probloss(0.3);
            cl3.receive(Id)probloss(0.4);}
      if(Id==3){
            cl1.receive(Id) probloss(0.3);
            cl2.receive(Id) probloss(0.2);}
    }
  }
}
main{
Server Server1(client1,client2,client3):();
Client client1(server1):(1,2,3);
Client client2(server1):(2,1,3);
Client client3(server1):(3,2,1);
}
```

```
 reactiveclass Client() {
   knownrebecs {Server mServer;}
   statevars {
         int rand,Id1,Id2,myId,rep;
         boolean ignore;}
  msgsrv initial(int m,int n,int k){
    myId = m;
    Id1 = n;
    Id2 = k;
     self.send();
  }

  msgsrv send() {
      rand=?(0.3:Id1,0.7:Id2)
      rep=?(0.5:1,0.5:2)
      while(rep>0)
      {
        mServer.takeRequest(rand)
     probloss(0.2);
        rep−−;
       }
      self.send();
  }
  msgsrv receive(int recId){
      If(recId!=myId)
      ignore=true;
  }
}
```

Fig. 2. The pRebeca model of unreliable client-server system.

We formalize the structure and state of a rebec in a pRebeca model P, as follows:

Definition 3. *The structure of a rebec, r_i (where $r_i \in Id_P$), can be described by a tuple (SV_i, KR_i, MS_i) where:*

- *SV_i is the set of the state variables of r_i.*
- *KR_i is the set of the known rebecs of r_i.*
- *MS_i is the set of the message servers in r_i.*

Definition 4. *The structure of a message received by rebec r_i can be formalized by a tuple Msg=(senderId, M) where:*

- *senderId $\in Id_P$.*
- *$M \in MS_i$.*

We denote the message queue of rebec r_i by q_i and define the following functions for a message queue:

Definition 5. *head(q_i) denotes the message at the head of message queue q_i. append(q_i, msg) denotes adding msg to the end of message queue q_i.*

Definition 6. *The state of a rebec r_i, can be defined by means of an evaluation function s_i: $SV_i \rightarrow Val_P$. The function s_i determines the value of each state variable in r_i at the current state.*

We separate the contents of the message queue of a rebec from its state for the sake of better matching and explaining the semantics based on an MDP structure. Hence, we define an evaluation function determining the content of the message queues in the model.

Definition 7. *The message queue evaluation function is defined as the function ζ:$(\bigcup q_i) \rightarrow (Id_P \times MS_P)^*$ where q_i denotes the message queue of r_i and $(Id_P \times MS_P)^*$ denotes all possible sequences of messages in a message queue.*

The operational semantics of a pRebeca model is based on an MDP in which, each state is a combination of the states of the rebecs included in the model and the message queue evaluation function. The transitions in this MDP can take place by first, nondeterministic selection of one of the rebecs with nonempty message queue and popping out the first message which determines the action of the transition, and then, a distribution function corresponding to the action determines the probability of transitions to the succeeding states.

Definition 8. *The operational semantics of a pRebeca model, P with n rebecs, can be defined as an MDP M_P=(S_P, S_{0P}, Act_P, τ_P, AP_P, L_P) where:*

- *S_P is the set of states where each state is a tuple of the form:*

$$(s_1, ..., \ s_n, \ \zeta) \tag{1}$$

  where s_i denotes the state of rebec i and ζ is the evaluation function of message queues.
- *S_{0P} is the initial state where all rebecs are in their initial states $s_{init\ i}$. In the initial state of each rebec, all the state variables have their initial values and the only message in each rebec's message queue is an initial message.*
- *$Act_P \subseteq Id_P \times MS_P$ is the set of actions in M_P which is the set of all possible messages that can be sent in P.*
- *AP_P is the set of atomic propositions which is a subset of $Id_P \times (\bigcup SV_i)$. We consider the Boolean-valued variables of the rebecs as labels. The atomic proposition (r_i, v) corresponds to variable v of rebec r_i.*
- *L_P is the labeling function $S_M \rightarrow 2^{AP_P}$, which assigns the variables with true value to the states.*
- *$\tau_P \in S_P \times ACT_P \times Dist(S_P)$ is the set of transitions in M_P. A transition in M is defined by:*

$$\frac{head(q_i) = a_i}{(s_1, ..., \ s_n, \ \zeta) \rightarrow^{a_i} \upsilon} \tag{2}$$

  where s_i is the state of rebec r_i and a_i is an invocation of one of the message servers of rebec r_i. This formula denotes the nondeterministic selection of action a_i at state (s_1,..., s_n, ζ). the function υ is the distribution function determining the probability of transition to the a_i-successors of state (s_1,..., s_n,

ζ). The probability of transitions is determined from the effect of statements in the body of the corresponding message server (as will be described shortly).

Processing action a_i can only make changes to the state of rebec r_i and also can change the contents of the message queues in the model. As in a pRebeca model a message is an invocation of a message server and the execution of message servers is atomic we need to specify the effect of each statement separately and afterwards the effect of executing a sequence of statements.
Assignment, conditional statements and loop statements have the same meaning as in Rebeca and a complete explanation of the semantics of these statements can be found in [1]. Thus, we continue with explaining the semantics of probabilistic primitives of the pRebeca language only. As said before, processing a message by rebec r_i only affects s_i and ζ. Hence, in the following we restrict our notation to rebec r_i only. To be more precise, all of the notations defined bellow must be subscripted by i, but we drop the subscript to make it more readable.

Definition 9. *The set of local states combined with the message queues is defined as IState= $(SV_i \to Val_P) \times (\bigcup q_i \to (Id_P \times MS_P)^*)$.
For σ=$(s \times \zeta) \in$ IState, s denotes the state of r_i and ζ is the contents of the queues.*

Definition 10. *A distribution function specifying the probability of transition to the successor IStates can be defined as: φ: IState$\to$[0,1].*

In order to recursively compute the probabilities of transitions from an IState after the execution of a sequence of statements, we need to define the sum of two distribution functions in the case that equal IStates are generated from different paths of computations which may be a result of different combination of probabilistic choices during the execution of probabilistic statements.

Definition 11. *If φ=$\{\sigma_1 \mapsto p_1,..., \sigma_k \mapsto p_k\}$ and φ'=$\{\sigma'_1 \mapsto p'_1,..., \sigma'_k \mapsto p'_k\}$ be two distribution functions over IStates, then sum of the functions is defined as:
$\varphi+\varphi'$=$\{\sigma \mapsto p \mid \sigma \in \{\sigma_1,..., \sigma_k\} \bigcup \{\sigma'_1,..., \sigma'_k\}\}$
where p=$\frac{\varphi(\sigma)+\varphi'(\sigma)}{TotalProb}$, TotalProb= $\sum$ p , p $\in \{p_1,..., p_k, p'_1,..., p'_k\}$. We assume $\varphi(\sigma_j)$=0 for σ_j which is not in the domain of φ and $\varphi'(\sigma_j)$=0 for σ_j which is not in the domain of φ'. We also use the notation: $\sum_{j=1}^{n} \varphi_j$=$\varphi_1+...+\varphi_n$.*

We also define the scalar product of a probability p $\in$ [0,1] in a distribution function as follows:

Definition 12. *If φ=$\{\sigma_1 \mapsto p_1,..., \sigma_k \mapsto p_k\}$ and p $\in$ [0,1] then p.φ=$\{\sigma_1 \mapsto p.p_1,..., \sigma_k \mapsto p.p_k\}$.*

Definition 13. *The effect of a statement stmt on an IState σ is defined by means of the function Effect: IState $\times$ stmt $\to$ (IState $\to$ [0,1]). In fact, Effect(σ, stmt)=$(\sigma_1 \to p_1,..., \sigma_k \to p_k)$ where σ_i is one of the possible successors of σ with transition probability p_i.*

Definition 14. *The effect of a sequence of statements is defined by Effect(σ, stmt;stmtlist)=$\sum$ p_j.Effect(σ_j, stmtlist), $1 \leq j \leq k$, where Effect(σ, stmt)=($\sigma_1 \mapsto p_1$,..., $\sigma_k \mapsto p_k$) and stmtlist::= stmt | stmt;stmtlist denotes a sequence of statements.*

Now we define the effect of the probabilistic statements in pRebeca model:

- *Effect*(σ, pAlt{ prob(p_1): {stmtlist$_1$},..., prob(p_n): {stmtlist$_n$}})= $\sum$ p_j.Effect(σ, stmtlist$_j$).
- *Effect*(σ,x=?(p_1: v_1,..., p_n: v_n))=(σ[x:=v_1]$\mapsto$ p_1,..., σ[x:=v_n]$\mapsto$ p_n) where σ[x:=v_i] denotes a successor state of σ in which the value of the state variable x is changed to v_i.
- *Effect*(σ, r_i.m()probloss(p))=(σ[q_i=append(q_i, m( ))] $\mapsto$ (1-p), $\sigma \mapsto$ p) where σ[q_i=append(q_i, m())] denotes the state which differs from σ only in the contents of message queue q_i where a new message m is added to the end of the message queue.

Now we can explain the transition relation in the mentioned MDP. In transition (s_1,..., s_n,ζ)$\rightarrow^{a_i}$ υ where rebec r_i processes message a_i, we have:

-υ(s'_1,..., s'_n, ζ')=0 for the states where $s'_j \neq s_j$ for $i \neq j$.

-υ(s'_1,..., s'_n, ζ')=*Effect*(σ, Body(a_i)) where $s'_j = s_j$ for $i \neq j$.

where σ=(s_i, ζ) and Body(a_i) denotes the sequence of statements of the message server invoked by a_i.

5 Model Checking pRebeca

In this section, we explain our method for model checking pRebeca models. We chose PRISM as the model checker for pRebeca models since it is a powerful tool supporting analysis of models based on MDPs and other Markovian structures. We consider two different methods to analyze pRebeca models by means of PRISM. The first is the direct mapping of pRebeca models to PRISM models which is very complicated since the PRISM modeling language does not support most of the required high level constructs both in data and control aspects. There are many entities in a pRebeca model that can not be easily implemented in the PRISM language such as message queues and loops.

The PRISM language does not support array data type. Thus we need to model a message queue by defining a number of variables which model the locations of the queue and a variable to keep the number of messages in the queue. Popping out a message from a message queue in pRebeca which is automatically done can be modeled in PRISM by shifting the value of all variables towards the head and update the value of variable that keeps the number of filled elements. This part of code should be rewritten for different possible number of messages in a message queue. Besides, there is another problem with keeping parameters that can be sent through a message because there is no support for data structures in PRISM. There would be similar problems in implementing other high level constructs of pRebeca by PRISM language. The direct mapping as explained above may require to define a large number of variables which may cause extraordinary growth in model size and makes the model checking impossible.

As the direct mapping of pRebeca models to PRISM models would be overbearing and may cause extraordinary growth in model size and makes the model checking impossible we provided a two step method which makes significantly less overhead. This method has two main steps that can be explained as follows:

1. The pRebeca model is converted to an MDP using state space engine of Afra which is a Rebeca model checker.
2. The generated MDP is converted to a PRISM model.

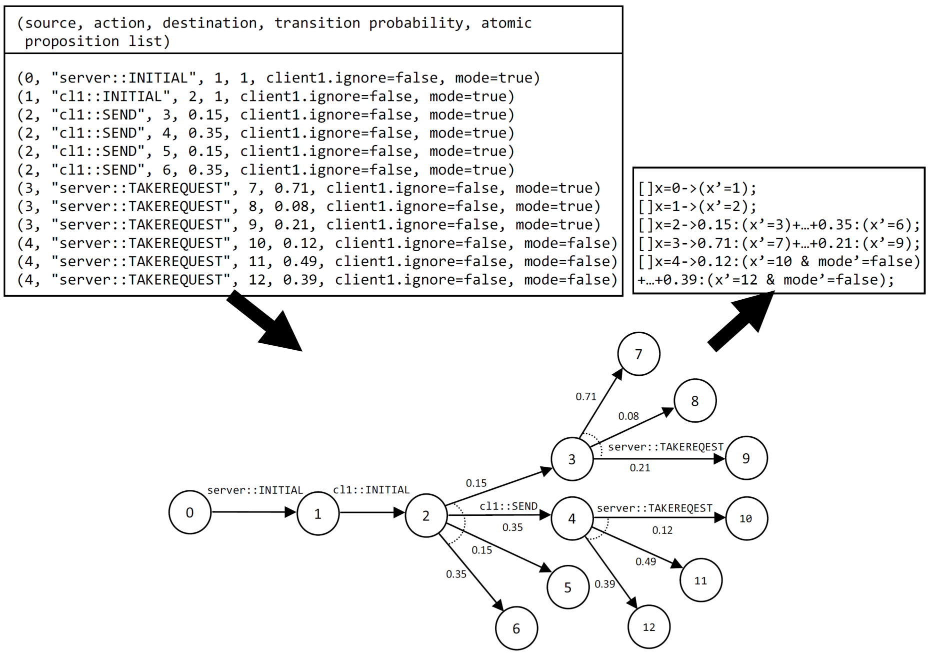
The result of the first step is an MDP which contains the states, transitions, and the probability of each transition. Before explaining the description of an MDP by means of the PRISM language, we give a brief explanation about PRISM language. A command in the PRISM language is in the form:
[ ]guard → prob$_1$: update$_1$ + ... + prob$_n$: update$_n$;

If the guard holds, each one of the updates may take place by the corresponding probability. If the guards of two or more commands hold simultaneously, one of them is executed nondeterministically.

In order to describe an MDP structure in PRISM, we assign each state of the MDP a number and the probability of transitions would be the corresponding probabilities with update parts. An update is the change of current state number which shows transition to another state. Figure 3 represents the transformation of a part of the state space of our running example, generated by Afra, in the form of an MDP and the corresponding PRISM specification. Using this method based on our experience has less overhead and keeps the size of the resulting models in an acceptable order to be checked by PRISM.

5.1 Case Study

We choose asynchronous leader election algorithm based on [24] as a case study. Leader election is one of the well-known algorithms used in distributed systems. In this variant of the algorithm, the nodes do not have any identifiers and are identical, spread over a network. In our case study, the network has ring topology, however the network topology can be easily changed by making a few changes to known rebecs part. Each node has a left and a right neighbour and only sends messages to its right neighbour. There are two modes for each node, active and inactive. At the beginning, all nodes are active. To elect a leader at the starting



```
(source, action, destination, transition probability, atomic
 proposition list)

(0, "server::INITIAL", 1, 1, client1.ignore=false, mode=true)
(1, "cl1::INITIAL", 2, 1, client1.ignore=false, mode=true)
(2, "cl1::SEND", 3, 0.15, client1.ignore=false, mode=true)
(2, "cl1::SEND", 4, 0.35, client1.ignore=false, mode=true)
(2, "cl1::SEND", 5, 0.15, client1.ignore=false, mode=true)
(2, "cl1::SEND", 6, 0.35, client1.ignore=false, mode=true)
(3, "server::TAKEREQUEST", 7, 0.71, client1.ignore=false, mode=true)
(3, "server::TAKEREQUEST", 8, 0.08, client1.ignore=false, mode=true)
(3, "server::TAKEREQUEST", 9, 0.21, client1.ignore=false, mode=true)
(4, "server::TAKEREQUEST", 10, 0.12, client1.ignore=false, mode=false)
(4, "server::TAKEREQUEST", 11, 0.49, client1.ignore=false, mode=false)
(4, "server::TAKEREQUEST", 12, 0.39, client1.ignore=false, mode=false)
```

```
[]x=0->(x'=1);
[]x=1->(x'=2);
[]x=2->0.15:(x'=3)+…+0.35:(x'=6);
[]x=3->0.71:(x'=7)+…+0.21:(x'=9);
[]x=4->0.12:(x'=10 & mode'=false)
+…+0.39:(x'=12 & mode'=false);
```

Fig. 3. Transforming pRebeca transitions (top-left) to PRISM specification (top-right) based on the equivalent MDP (bottom).

point, each node flips a coin and sends the result to its right neighbour. If a node has a head (denoted by false) in flipping coin and its left neighbour gets tail (denoted by true) it becomes inactive, and it repeats the process otherwise. An inactive node can only pass messages it receives to its right neighbour. Nodes initially have the number of active nodes and in the case that a node becomes inactive, it decreases the number of active nodes by one. The number of active nodes will be passed through messages making it possible for nodes to update their local information. The execution will continue until the number of active nodes equals 1. The pRebeca code of leader election protocol is presented in Figure 4.

We have modeled this system for 3, 4 and 5 nodes and the resulting PRISM models have the proper size to be checked by the PRISM model checker. We have model checked some PCTL properties such as the maximum probability of electing a leader in 2N rounds where N is the number of nodes and the maximum probability of finally electing a leader. We have also analyzed a number of well-known protocols and algorithms such as randomized dinning philosophers (with 6 philosophers) and IPV4 zeroconf protocol and the time/space consumption of model checking was reasonable.

```
reactiveclass Node() {
 knownrebecs {
   Node rNeighbour;
 }
 statevars {
    boolean leader;
    int nActive;
    boolean active;
     boolean x;
  }
  msgsrv initial(int num) {
     nActive=num;
     active=true;
     leader=false;
     self.flip();
  }
  msgsrv flip() {  //flipping the coin and sending result to right neighbour
     x=?(0.5:false,0.5:true);
     rNeighbour.elect(x,nActive);
  }
  msgsrv elect(boolean i,int n) {
     if(active==true){
            if(nActive==1){
                  leader=true;
                  rNeighbour.leaderElected()
            }
            else{
                  if(nActive>n)  //updating the number of active nodes if needed
                       nActive=n;
                  if(x==false ∧ i==true){
                       active=false;
                       nActive−−;
                  }
                  else{
                       self.flip();
                  }
            }
     }
     else{  //just passing received messages in case of being inactive
              rNeighbour.elect(i,n);
     }
  }
  msgsrv leaderElected(){
        if(leader==false){  //informing right neighbours that a leader has been elected
             active=false
             rNeighbour.leaderElected();
        }
  }
}
main{  //declaration of rebecs and passing the number of rebecs to initial message servers
      Node n0(n1):(3);  //the number of nodes is passed to initial message servers
      Node n1(n2):(3);
      Node n2(n0):(3);
}
```

Fig. 4. The pRebeca model of randomized leader election.

6 Conclusion and Future Work

In this paper, we presented pRebeca, an extension to an object-based high-level modeling language based on actor model, which is suitable for modeling asynchronous distributed systems. We proposed the syntax of pRebeca which is an extension of the Rebeca syntax and also provided the semantics of this language based on MDPs. A two step method was presented to model check pRebeca models by means of the well-known probabilistic model checker PRISM. Using the proposed approach, the modeler of a probabilistic distributed system can effectively model the system in a high-level, readable, and maintainable language. These benefits all come from the object encapsulation and elegant concurrency paradigm in actor model, as well as familiar, high-level Java-like syntax of Rebeca. Our effective way to generate PRISM models from pRebeca enables to have the mentioned benefits, while using a powerful model checking engine like PRISM. Although our method reduces the complexity of converting pRebeca model to a proper input model described by the PRISM language, we are planning to develop methods and tools to better and more efficiently analyze pRebeca models.

References

1. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inf. **63**(4) (June 2004) 385–410
2. Hewitt, C.: 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 (April 1972)
3. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
4. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7** (1998) 1–72
5. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. PPPJ '09, New York, NY, USA, ACM (2009) 11–20
6. Hewitt, C.: Orgs for scalable, robust, privacy-friendly client cloud computing. IEEE Internet Computing **12**(5) (September 2008) 96–99
7. Hewitt, C.: Actorscript(tm): Industrial strength integration of local and nonlocal concurrency for client-cloud computing. CoRR **abs/0907.3330** (2009)
8. Agha, G., Meseguer, J., Sen, K.: Pmaude: Rewrite-based specification language for probabilistic object systems. Electron. Notes Theor. Comput. Sci. **153**(2) (May 2006) 213–239
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G., Qadeer, S., eds.: Proc. 23rd International Conference on Computer Aided Verification (CAV'11). Volume 6806 of LNCS., Springer (2011) 585–591

10. Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: Sysfier: Actor-based formal verification of systemc. ACM Trans. Embed. Comput. Syst. **10**(2) (January 2011) 19:1–19:35
11. Baier, C., Kwiatkowska, M.Z.: Domain equations for probabilistic processes. Electr. Notes Theor. Comput. Sci. **7** (1997) 34–54
12. den Hartog, J., de Vink, E.P.: Mixing up nondeterminism and probability: a preliminary report. Electr. Notes Theor. Comput. Sci. **22** (1999) 88–110
13. Hansson, H.A.: Time and Probability in Formal Design of Distributed Systems. Elsevier Science Inc., New York, NY, USA (1994)
14. Larsen, K.G., Skou, A.: Compositional verification of probabilistic processes. In: Proceedings of the Third International Conference on Concurrency Theory. CONCUR '92, London, UK, UK, Springer-Verlag (1992) 456–471
15. Lowe, G.: Probabilistic and prioritized models of timed csp. Theor. Comput. Sci. **138**(2) (1995) 315–352
16. Penczek, W., Szalas, A., eds.: Mathematical Foundations of Computer Science 1996, 21st International Symposium, MFCS'96, Cracow, Poland, September 2-6, 1996, Proceedings. In Penczek, W., Szalas, A., eds.: MFCS. Volume 1113 of Lecture Notes in Computer Science., Springer (1996)
17. Tofts, C.M.N.: A synchronous calculus of relative frequency. In: Proceedings of the Theories of Concurrency: Unification and Extension. CONCUR '90, London, UK, UK, Springer-Verlag (1990) 467–480
18. Ciesinski, F., Baier, C.: 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) 131–132
19. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The Ins and Outs of The Probabilistic Model Checker MRMC. In: Quantitative Evaluation of Systems (QEST), IEEE Computer Society (2009) 167–176 `www.mrmc-tool.org`.
20. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: Modest: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. **32**(10) (2006) 812–830
21. Katoen, J.P., ed.: Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings. In Katoen, J.P., ed.: ARTS. Volume 1601 of Lecture Notes in Computer Science., Springer (1999)
22. 2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings. In: MEMOCODE, IEEE (2004)
23. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
24. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation **88**(1) (1990)