Journal of Advances in Computer Research
Quarterly ISSN: 2008-6148
Sari Branch, Islamic Azad University, Sari, I.R.Iran
(Vol. 3, No. 4, November 2012), Pages: 1-16
www.jacr.iausari.ac.ir

# Modeling and Evaluation of Stochastic Discrete-Event Systems with RayLang Formalism

**Ali Khalili, Mohammad Abdollahi Azgomi**✉

*School of Computer Engineering,*
*Iran University of Science and Technology, Tehran, Iran*
Khalili.ir@gmail.com; azgomi@iust.ac.ir

**Received: 2012/07/24; Accepted: 2012/09/13**

**Abstract**

*In recent years, formal methods have been used as an important tool for performance evaluation and verification of a wide range of systems. In the view points of engineers and practitioners, however, there are still some major difficulties in using formal methods. In this paper, we introduce a new formal modeling language to fill the gaps between object-oriented programming languages (OOPLs) used by engineers and the formalisms used for evaluation and verification purposes. We propose the syntax and semantics of a new object-oriented modeling language for discrete-event systems called RayLang. We have designed the syntax of RayLang similar to OOPLs. In RayLang models, objects that are instantiated from classes, run concurrently and can communicate with each other by requesting services. Every object in RayLang models has some internal state variables and some service handlers for executing the requests of other objects. We have shown that Markovian RayLang models can be transformed into continuous-time Markov chains (CTMCs) and then can be solved by existing solution techniques. For modeling, discrete-event simulation and analytic solution of RayLang models, we have implemented these models in the PDETool framework.*

***Keywords:*** *Formal Modeling Language, Object-Oriented Modeling, Discrete-Event Systems, Performance and Dependability Evaluation*



## 1. Introduction

In spite of the recent successful application of formal methods, there are still clear needs of further research to develop and design new formalisms. In many notations, it is not easy to understand the meaning and properties of the symbols and how they may and may not be manipulated, and to gain fluency in using them, to express new problems, solutions and proofs [5]. The major problem is the difference in the abstraction level and the existing gap between the programming languages used by programmers and software engineers and the modeling languages used for performance evaluation and verification purposes. Most of the existing formalisms cannot be easily used by software engineers. On the other hand, languages used by software engineers cannot be used directly for performance and dependability evaluation purpose and are too informal or too heavy to be analyzed by a verification tool [24].

The natural idea behind the object-oriented paradigm is to consider the system we intend to model, simulate or develop, as a collection of active objects which collaborate

with each other. The term active object denotes an autonomous entity equipped with its own behavior and some internal data which corresponds to the state of object and of course may change during its lifetime [9].

The aim has been to introduce a new formal modeling language to fill the gaps between object-oriented programming languages (OOPLs) used by engineers and the formalisms used for performance evaluation and verification purposes.

This paper presents the syntax and semantics of a new object-oriented modeling language for discrete-event systems called RayLang. Based on the above aim, we have designed the syntax of RayLang similar to OOPLs. In RayLang, objects that are instantiated from classes, run concurrently and can communicate with each other by requesting services. RayLang has a well-defined formal semantics and incorporates several ingredients of programming languages and light-weight notations. Every object in a RayLang model has some internal state variables and some service handlers for executing requests of other objects. The state space of RayLang models can be generated. We have shown that Markovian RayLang models can be transformed into continuous-time Markov chains (CTMCs) and then be solved by existing solution techniques.

The remainder of this paper is organized as follows. Section 2 explains the motivations of this work. Section 3 briefly introduces the related works. In Section 4, the core language of RayLang and its syntax and semantics are presented. The analysis of RayLang models are discussed in Section 5. Two illustrative examples are given in Section 6. Section 7 compares RayLang with other formalisms. And finally, Section 8 concludes the paper.

## 2. Motivations

The main motivation of proposing a new formalism has been to provide an object-oriented language for modeling, performance evaluation and verification of stochastic discrete-event systems with a well-defined and easy to use syntax and semantics. The proposed formalism should have the basic necessary constructs in a programming language-like syntax (instead of using mathematical notation), which makes it easy to use for practitioners. Thus, the formalism will provide an effective and simple approach to model discrete-event systems.

The intended formalism may also integrate model-based performance evaluation techniques, model checking techniques and software engineering concepts. To model a discrete-event system using the new formalism, a modeler can exploit performance evaluation techniques based on the specified rewards using analytic solving techniques or discrete-event simulation. Having these features, makes the new formalism a light-weight notation with a rigid semantics that tries to fill the gap between traditional formal methods and software engineering tools.

## 3. Related Work

In the literature, there has been a lot of effort to propose modeling languages for modeling concurrent and distributed systems. In this section, we will briefly review those models, which are closely related to the work presented in this paper.

Petri nets (PNs) [23] which have been introduced for modeling concurrent and distributed systems are well-known models that have graphical representation in

addition to the formal definitions and structures. Due to their modeling power and simplicity, Petri nets have been extended with several features. Stochastic extensions of Petri nets, such as stochastic Petri nets (SPNs) [21], generalized stochastic Petri nets (GSPNs) [2] and stochastic activity networks (SANs) [22] are the most famous models used in performance and dependability evaluation of systems.

In addition, several high-level and object-oriented extensions of Petri nets have been introduced to make them more appropriate and easy to use. Coloured Petri nets (CPNs) [14], stochastic well-formed nets (SWNs) and language for object-oriented Petri nets (LOOPN) [17], object Petri nets (OPNs) [16] and concurrent object-oriented Petri nets (CO-OPN) [6] are the most useful examples of such extensions.

Modeling, specification and evaluation language (MOSEL) [3] is a description language which depicts a network like model with a controlled flow of tokens, expresses the two-dimensional network structure, the token flow-control mechanisms as well as the routing probabilities and the stochastic delay information in a one-dimensional textual notation.

Maude [18] is a language based on rewriting logic with modules use rewrite theories, while computation with such modules corresponds to efficient deduction by rewriting. Real-time Maude [8] is a language and tool supporting the formal specification and analysis of real-time and hybrid systems in timed modules and timed object-oriented modules, which can be transformed into equivalent Maude modules.

Classical process algebras are abstract languages used for the specification and design of concurrent systems. The most widely known process algebras are the calculus of communicating systems (CCS) [20] and communicating sequential processes (CSP) [12]. There exist several timed and stochastic extensions of process algebra with the aim of performance evaluation. Performance evaluation process algebra (PEPA) [11] extends the classical process algebra with the capacity of assigning rates to activities to quantify time and uncertainty, which are described in an abstract model of a system.

The specification language $\chi$ [13] which is developed as a modeling and simulation tool for design of concurrent systems is inspired by CSP and the guarded command language (GCL). Similar to CSP, the behavior of system components is described by processes that communicate via channels. Communication in $\chi$ is synchronous, unidirectional and timeless and variables are typed.

The modeling language MoDeST [4] reasons about stochastic timed systems based on the stochastic timed automata (STA) formalism by combining the features of probabilistic GCL (pGCL), process algebras with TCSP-like synchronization over common actions, and other language constructs such as urgent actions, exception handling and process instantiation.

The Actor model of computation was originally proposed by Hewitt [10] and further developed by Agha [1] into a concurrent object-based model. Actors are self-contained, concurrently interacting entities of a distributed computing system. They communicate via asynchronous message passing which is fair and can be dynamically created and the topology of actor system can change dynamically. Reactive objects language (Rebeca) [25] is an actor-based language with a formal foundation supported by a front-end tool for the translation of models into the input languages (*e.g. PROMELA*) of the existing modelchecking tools (*e.g.* SPIN).

## 4. Definitions of RayLang

RayLang is an object-oriented stochastic modeling language for discrete-event systems, *i.e.* systems which are in a state during some time interval, after which an atomic event might happen that changes the state of the system immediately. RayLang has two different settings:

- *Non-deterministic setting.* In this setting, the events occur in a nondeterministic manner and no timing parameter is required. The application of this setting is on model checking.
- *Stochastic setting.* In this setting, the time parameters are specified for the events, which indicate the completion time of the actions related to the events (which are called services here). The application of this setting is on performance and dependability evaluation.

The focus in this paper is only on the stochastic setting. For this purpose, we will firstly present the informal definition of RayLang. Then, the syntax and formal definitions and behavior of the model will be presented.

***An Informal Description of RayLang***

In RayLang, a model is composed of a finite set of reactive, self-contained and communicating *objects*, which are executed concurrently. An object is instantiated from a *class*, consisting of some *state variables* and *services*. A state variable has a *type* (such as, *integer, short, Boolean,etc.*) which defines the values that variable can hold and should have an *initial value*. A special kind of variables are called *condition variables* that are of type Boolean. *References* are another kind of variables which denote object acquaintances and can be used for object communication purpose.

Services model the behavior of an object change its state variables. Unlike methods in programming language, services have not any return value. There are two types of services: *ordinary* and *immediate*. Ordinary services are executed by passing an asynchronous message (called *service request*) to them. Each ordinary service has an unbounded buffer, identified as *service queue*, for arriving service requests. When a request at the head of a queue of an ordinary service is serviced, its service handler is invoked and the request message is deleted from the corresponding queue after a time based on the service *probability distribution function* (PDF). Each ordinary service could have a precondition which is a Boolean expression and acts as a guard for enabling/disabling the execution of service. If the precondition of an ordinary service is evaluated to true, it can be triggered by removing a request message from the top of its corresponding queue and results in an atomic execution of its body which cannot be interleaved by any other ordinary service execution. For an ordinary service, a user-defined PDF must be specified which models the service execution time (the time that must be elapsed while the service is enabled until it finishes serving a request in its requests queue).

On the other hand, immediate services do not have any service queue, but rather act like methods or functions in programming languages, *i.e.* when an immediate service is called (requested), it is executed immediately (synchronously). An immediate service could not have a precondition and is always enabled and can be executed immediately whenever it is called. Requesting an immediate service can be viewed as a synchronous (local or remote) method invocation.

In a RayLang model, computation takes place by means of requesting services (a kind

of message passing) and the execution of services. An object can communicate and interact with other objects when it ``*knows*'' their name, which are said to be their acquaintances. An object knows its acquaintances by having references to them.

***The Syntax of RayLang***

The Java-like syntax of RayLang classes (object templates), objects (class instantiations), and models (parallel composition of objects) in a BNF-like notation is presented in Figure 1. Note that the words in italic show terminals (such as language keywords, operators and identifiers).

In each class, after declaring variables, some services will be defined. Service body instructions allow a probabilistic case (*case*) or some (remote or local, ordinary or immediate) service requests (*servRequest*), broadcast service requests (*brdcstRequest*), value assignments (*assgmntStatement*), conditional statements (*condStatement*) and sequential compositions.

In service request statements, the ID of callee (object which receives the request) and the service name can be specified that are followed by actual parameters. If the callee is not specified (or keyword *this* is used), the caller and callee are the same which models a local service request. If the requested service is an ordinary service, this can be viewed as an asynchronous message passing that message contains the parameters passed to the corresponding service handler in callee which represents a request for service. This could be regarded as a method call (similar to conventional object-oriented programming languages) if the requested service is an immediate service. Sometimes, this can be used for object synchronization.

| | |
|---|---|
| model | → (class)+ main rewards |
| class | → *'class'*ClassName*'{'* (varDefinitions)? servDefinitions*'}'* |
| varDefinitions | →varType*VarName* (*','VarName*)* *';'* |
| varType | →refVariableType \| conditionVariableType \| stateVariableType |
| refVariableType | →*ClassName* |
| condVariableType | →*'condition'* |
| stateVariableType | →*'int'* \| *'short'* \| *'byte'* \| *'bool'* |
| servDefinitions | → service+ |
| service | →(*'immediate'*)? *'service'ServiceName'('*(parameters)?*')'* Dist? *'{'* (precondition)? serviceBody*'}'* |
| parameters | → parameter (*','* parameters )? |
| parameter | →*parTypeparName* |
| precondition | →*'precondition' ':'*booleanExpression |
| Dist | →*':' distributionFunction'('* actualParameters*')* |
| serviceBody | →caseStructre \| ( statement *';'* )* |
| caseStructute | →*'case''{'* (option)+ *'}'* |
| option | →*ProbValue':'* ( statement *';'* )+ |
| statement | →assgmntStatement \| ifStatement \| serviceRequest \| brdcstRequets \| case \| varDef |
| brdcstRequets | →*className'.'servName'('* (actualParameters)? *')'* |
| serviceRequest | → ((*objectID* \| *'this'*) *'.'* )? *servName'('* (actualParametes)? *')'* |
| actualParam | → ( *value* \| *varName* \|*'this'*) {*','* ( *value* \| *varName* \| *'this'*) }* |
| assgmntStatement | →*varName'++'* \| *varName '--'* \| *varName'='* Expression |
| ifStatement | →*'if'* '(' booleanExpresion ')' *'{'*statemensts*'}'* (*'else' '{'*statemensts*'}'* )? |
| varDef | →varDefinitions |
| main | →*'main' '(' ')' '{'*objectDef * *'}'* |
| objectDef | →*classNameobjName'('*initializationParameters*')' ';'* |

***Figure 1.The syntax of RayLang***

Ordinary services could be exploited in broadcast service requests. A broadcast request statement $className.serviceName(p_1,\dots,p_n)$ causes broadcasting a request message to the ordinary service *serviceName* of all objects in the model with type *className* where $p_1,\dots,p_n$ are the parameters of the message. In this way, a request will be sent to an ordinary service of all objects with a special type defined in the model.

A (probabilistic) case statement (*case*) can be used for the execution of some alternatives and causes one of its corresponding choices to execute probabilistically. Case statements can only be used in ordinary services. After defining the classes, there is a part for specifying model configuration (*main*). This part is specified with the keyword *main* followed by the definition of the model configuration which is defined as a finite number of objects that must be created and then run concurrently. Each class definition has a constructor with a name same as the class.

When an object is defined in main, a request is sent to its constructor to create an object with proper parameters. This constructor must assign the initial values for all local (state, condition and reference) variables and could only request local ordinary services (immediate, remote and broadcast service requests as well as case statements are not allowed in constructor). The *Constructor* must be defined as immediate services. After definition of all objects in main part of a model and thus immediate execution of their constructors, the model will be in its initial state.

Some non-terminals (such as expressions and parameters of services) that are not given in the grammar are like as in programming languages (*e.g.* Java). Like programming languages, the model uses scope rules for using identifiers. An identifier (of a variable) declared in a class is ``*known*'' in all services of that class. When an identifier in service parameters has the same name as an identifier in the class, the identifier in the class is ``*hidden*'' and could be accessed only by the keyword *this*. Because the state and reference variables in an object are private, they could not be used or modified in other objects. Condition variables of an object could be used as read-only Boolean variables via the name of its object (by using its reference) in precondition or statements in service body of other objects. For the sake ofeasy modeling, the modeler can also use local variables in the body of services which do not affect the object's local configurations (state variables, reference variables,*etc.*). As in programming languages, local variables are created dynamically and are used onthe execution of a service and after that, the value of them will be destroyed and cannot be used in the next service execution.

**Semantics of RayLang**

In this subsection, the semantics of RayLang is presented. The variables defined in an object $o_i$ are typed (integer, boolean, char, condition, reference, *etc.*). We skip the detail of variable declarations as they are irrelevant to the purposes of this paper. We assume a finite set *Variables* of variables and a domain (type) *Dom(x)* for any variable *x*. We write *Values* to denote the set of all possible values for the variables, *i.e.*$Values = \bigcup_{x\in Variables} Dom(x)$. The initial value of each variable must be specified in the constructor. An object $o_i$ (with the unique identifier *i*) is a tuple $< V_i, Serv_i^*, iServ_i^*, Q_i >$ where $V_i$ is the set of variables (state variables, condition variables and reference variables), $Serv_i^* = \bigcup_{1\le j\le J} Serv_i^j$ is the set of all ordinary services $Serv_i^j$, and $iServ_i^* = \bigcup_{1\le k\le K} iServ_i^k$ is the set of all immediate services $iServ_i^k$ of object $O_i$ where *J* and *K* are the number of ordinary and immediate services,

respectively and $j$ and $k$ are called service identifier of the ordinary service $Serv_i^j$ and immediate service $iServ_i^k$. Each ordinary service $Serv_i^j$ is a tuple $< Pre, Dist, Body >$ where the $Serv_i^j.Pre$ is the precondition expression of the service $Serv_i^j$ (as described in syntax section), $Dist(.)$ is general probability distribution function which may have zero or more parameters and describes the time that the service needs to be executed while it is enabled, and $Body$ is a set of RayLang instructions. For an ordinary service, if the precondition is not specified explicitly, it will always be evaluated as true. Each ordinary service in an object has a queue which can be defined as a finite sequence of requests for that service. The service request queue of an object $O_i$, defined by $Q_i$, is like a multi-queue consisting of all the queues of its ordinary services and including all the requests that have been sent to the ordinary services and have not been serviced yet. $Q^*$ denotes the set of unbounded FIFO multi-queues that contains message requests of all (ordinary) services of all defined objects, *i.e.*$Q^* = \bigcup_{\forall i \in I} Q_i$ which $Q_i$ is the unbounded FIFO multi-queue corresponding to services request queue of object $O_i$. Let's $Q_i$ be a set of all unbounded FIFO queues $Q_i = \bigcup_{1 \le j \le J} q_i^j = \{q_i^1, \dots, q_i^J\}$) where each queue $q_i^j$ contains requests of the ordinary service $Serv_i^j$ in object $O_i$. For mapping each ordinary service of an object to one of these queues, function *queue*, $queue: Serv_i^j \to q_i^j$, is defined such that $q_i^j \in Q_i$.For a model Ω, there exists a universal set $I$ of all objects that are engaged in the model and $|I|$ denotes the number of all defined objects in the system.

The state space of the model (Γ) is defined as:

$$\Gamma = \prod_{1 \le i \le |I|} (\Gamma_i \times Q_i)$$

where $\Gamma_i: Variables \to Values$ is the local state of the object $o_i$, an evaluation function that maps each local variable to a value of the appropriate type ($\forall x \in Variables, \Gamma_i \in Dom(x)$), and $Q_i$ is the unbounded FIFO multi-queue of service requests for the object $O_i$ as mentioned before. However, all of these state need not be actually reachable.

Now, we can define the behavior of a model Ω as a labeled transition system $< \Gamma, L, \to, \gamma_0 >$ where:

- $\Gamma = \prod_{1 \le i \le |I|} (\Gamma_i \times Q_i)$ is the set of states as said earlier where $\Gamma_i$ is the set of possible values for internal object's (state, condition and reference) variables and $Q_i$ is an unbounded FIFO multi-queue of the service request in the object $O_i$,
- $L = \bigcup_{1 \le i \le |I|} Serv_i^j$ is the set of labels that are all possible ordinary service request that can be executed in Ω, where $Serv_i^j \in L$ means the execution of a request for service $Serv_i^j$ in object $O_i$,
- $\to \subseteq \Gamma \times L \times \Gamma$ is the set of transition relations on Γ where $\gamma_1 l \gamma_2$ iff $\gamma_1, \gamma_2 \in \Gamma$ and $l = Serv_i^j \in L$ is an enabled transition which means $\exists j: r = head(\gamma_1.q_i^j)$, $r$ is the request for the ordinary service $Serv_i^j$ on head of the queue and the precondition $Serv_i^j.Pre$ of the service is evaluated as true and $\gamma_2$ results from $\gamma_1$ by $l$ as follows:
  - Performing the actual transition from the state $\gamma_1$ to the state $\gamma_2$ (by the service $Serv_i^j$) needs time $t$ which depends on the probability distribution function of

the service ($Serv_i^j.Dist$) in the state $\gamma_1$ and the real execution will be completed when time $t$ is elapsed in which the service remains enabled. As we assume that the weight and the priority of all services are equal in the time $t$, if there exist multiple enabled services that are scheduled for execution, one of them is chosen for execution with equal probability.

- The request message is removed from the head of $\gamma_1.q_i^j$, *i.e.*$\gamma_2.q_i^j = tail(\gamma_1.q_i^j)$. This causes assigning the values of the formal parameters to the real parameters, too.
- An ordinary service execution that is caused by request $r$ leads the execution of service $Serv_i^j$ of the object $o_i$ as a sequence of instructions as an atomic operation as follows:
  - If the ordinary service is a multi-option service (*i.e.* the service has a probabilistic case statement in its top level of statements), each case option (internal statements of the option) leads in an atomic operation with the probability $p$ defined as the option's probability. If not, the execution of the service leads to unique outcome state,
  - The execution of an ordinary statement (*e.g.* assignment statements, conditional statements, *etc.*) in $Serv_i^j$ may change the value of some variables in object $o_i$,
  - The execution of each statement:

    $$((this|objectID)?.servName(actualParams?)$$

    where *servName* is an ordinary service, changes the service request queue $q_{j'}^{i'}$ where $i'$ is the object identifier of *objectID* and is same as $i$ when local service request (without specifying any *objectID* or with keyword *this*) is used and $q_{j'}^{i'} \in Q_{i'}$ is the queue corresponding to ordinary service *servName* when $j'$ is the service identifier,
  - Execution of each ordinary service broadcast request statement (*className.servName(actualParams?)*), changes all service request queues $q_{j'}^{i'}$ where $i'$ is object identifier of all objects with type *className*, $i' \in \{t|1 \le t \le |I| \wedge Dom(o_t) = className\}$, and for each $i'$, $j'$ will be the service identifier of *servName* in object $o_{i'}$. Immediate services could not be used in broadcast requests,
  - Execution of each immediate service request statement

    $$((this|objectID).servName(actualParams?)$$

    (where the corresponding *servName* is an immediate service), causes the execution of $iServ_{i'}^{j'}$ (like a method call in programming language) where $i'$ is the object identifier of *objectID* and is same as $i$ when local service request (without specifying any *objectID* or with keyword *this*) is used and this causes atomic execution of some statements, respectively.

- $\gamma_0$ is the initial state of the model. Variables must be initialized to their default values according to their types in constructors, and $Q_0$ is defined such that in the beginning of the model, for all $i$, constructor of object $O_i$ is executed. It is obvious that $\gamma_0 \in \Gamma$.

## 5. Analysis of RayLang Models

RayLang is a language for modeling discrete-event systems. The state space of RayLang models can be generated by model execution in which every ordinary service can be viewed as an event and the value of object variables in addition to the content of service queues represent the state of the model (as mentioned earlier). The state space of the model can be considered as a labeled transition system which could be used in the analysis process. The initial state of the model is the situation where only the constructors of all objects have been executed. In each state, some ordinary services may be enabled, *i.e.* there exist at least one request in their request queue and their preconditions evaluated to true. The execution of each enabled ordinary service causes a transition to another state (if the modeler uses probabilistic case statement in a service, the execution of the service results in probabilistic transitions into different states). Generating the state space can be done in a depth-first or breadth-first order. Using the defined semantics, if the state space of the model starting from the initial state is finite, we can define the reachability graph as a labeled directed graph based on the reachability set (the set of all reachable states). This reachability graph can be used for model checking. If all ordinary services of a model have exponential PDFs, the reachability graph can also be transformed into a continues-time Markov-chain (CTMC), which can be used for analytic solution of the model.

Analytical approaches usually suffer from the state space explosion problem. When the state space of a model is very large or larger than to be handled, discrete-event simulation techniques can be used to analyze RayLang models. A simulation path is a (random) path between some states starting from the initial state. Each event in the path is the (atomic) execution of an enabled ordinary service.

***Reward Specification in RayLang Models***

For evaluation of a RayLang model, the modeler must define some reward variables which are specific measures of system behavior based on his/her interests. Rewards that are supported here are based on [19] which defines a unified approach to the specification of performance, dependability and performability. A reward variable is a set of one or more functions (with type *double*) which return reward value and are defined on a state or transition between states in the model, *i.e.* there are two types of reward functions: impulse reward functions and rate reward functions that specify impulse and rate rewards, respectively. Rate rewards are used to define rewards based on the time in each state and the corresponding function defines the measurement that the reward should evaluate. The impulse reward functions that define impulse rewards are evaluated when the specified ordinary service of an object in the model is executed. For each reward, type of rewards must be defined which determines when, in system time, the reward functions must be evaluated and includes *steadyState*, *instanceOfTime*, *intervalOfTime* and *averagedIntervalOfTime* as presented in the syntax of reward specification (Figure 2).

```
→'rewards''{'(reward)* '}'                    rewards
→rewardID';'rewardType'{'rateReward?                    reward
(impulseReward)* '}'
→'-' 'execute''('ObjID'.'OrdinaryService ')' '{' Body'}' impulseReward
→'-' 'if' '{' booleanExpresion? '}' '{' Body '}'        rateReward
```

***Figure 2.The syntax of the reward specification in RayLang models***

***Implementation of RayLang in PDETool***

PDETool is a multi-formalism modeling tool which provides features for construction and translation of models into the XML-based input language of an SDES-based simulation engine, called SimGine [15] . PDETool is easily extensible to support a wide range of graphical and non-graphical formalisms. Furthermore, it facilitates the construction, animation and analysis of models. A formalism can be implemented in PDETool if a mapping can be provided to the input language of SimGine.

We have implemented RayLang in PDETool by developing a translator, which converts RayLang models into the input language of SimGine. It makes modelers enable to evaluate RayLang models using discrete-event simulation technique. If all events' delay functions are exponentially distributed, the tool can also be used to analytically solve the model. In this direction, the model is transformed into a CTMC by generating reduced reachability graph, which can be used in both transient and steady-state analysis based on the specified reward variables.

The translation of RayLang model into the input language of SimGine is performed by a two passes *LL(K)* parser. In a nutshell, this process is as follows. Objects' states and services are flattened in the first step of translation. For each object, internal state-variables will be translated into the model's state variable. Each ordinary service is mapped into an event in addition to a queue (as a state-variable) which contains the service's requests. All immediate services are translated into auxiliary functions, which can be called during model execution and has a special parameter *objectID* to determine the object in which the service is called.

## 6. Illustrative Examples

In this section, we present two examples to illustrate the syntax and semantics of RayLang.

***Example 1. A Client-Server Model***

Consider a client-server system, presented in Figure 3, which has a server that serves some clients. Each client sends a request to the server in the service *sendReq* (with exponential rate 1) and waits for the server to respond it. The server has a queue (corresponding to a service say*srvReq*) contains all received requests that is not served yet. The process of requesting and generating a proper response take a time which is distributed exponentially. The generated response sent to the client, which generates the corresponding request and after that, the client *rcvRsp* is executed after 3.0 time units.

```
class Client{
        Server theServer;
        immediate service Client(Server theServer){
                this.theServer=theServer;
                arrive();
        }
        service arrive():Exp(1){
                theServer.enterInQueue();
    arrive();
        }
}
class Server{
        immediate service Server(){          }
        service enterInQueue():Exp(5){
                serv();
  }
        immediate service serv(){}
}
main(){
        Server s();  Client c(s);
}
rewards{
        numberOfWaitedCustomers:steadyState{
                -if(true){return s.enterInQueue#;}
        }
}
```

***Figure 3.A client-server model in RayLang***

There are two classes, *Client* and *Server*, which are templates of the buffer and some clients. Each client has a reference *s* to the object server for interacting with it. The server interacts with its clients via a reference which is passed to the service *rcvReq* by the parameter *c* with of the type of client.

Objects' definition is followed by classes. Each object has a type and proper initialization parameters, which are passed to the constructor of that object. Here, a server and four clients are defined. After initializing all objects by executing their constructors, the model would be in its initial state. The impulse reward variable *reward1* is specified to investigate how many requests are served between time 10 and 20 in the server.

***Example 2.A Readers and Writers Model***

As the second example, we modify the last example to model probabilistic cases and mutual exclusion. Assume that there exist a buffer and some customers that access to the buffer for reading from/writing to it. Multiple readers can read from the buffer simultaneously but a writer could write into it only if there is no readers or writers (any customers generally). Figure 4 shows a RayLang model for this problem.

There are two classes, buffer and customer, which are templates of the buffer and some customers. The state variables *readers* and *writer* in object buffer represent the number of simultaneous readers and writer in the buffer (*writer* is always equal to zero or one). Condition variables *customerExist* and *writerExist* will be true if any customers (readers or writer) and any writer exist in the buffer, respectively.

At the beginning, each object customer will have a request for executing *accessBuffer* to access the buffer. This will cause to try reading from/writing to the buffer with probability *0.9/0.1* which is done by using a probabilistic case statement. Generating such a request is performed during the time exponentially distributed with rate *1*. The object *buffer* serves the requests of customers. A service *startRead* can be executed if

the buffer is not performing a write request and a service *startWrite* can be started only if the buffer is not performing any other (read/write) request. Executing a service *startRead*/*startWrite* and *finishRead*/*finishWrite* show that the buffer starts to perform a read/write request and finishes performing a read/write request, respectively. Start process of the request takes place as early as possible (the PDF of the service is deterministic with the parameter zero and it will be done whenever it is enabled). The immediate service *getResult* in each customer object is performed when the request of the object in the buffer is served and change the state of customer from waiting into active.

In reward specification part, *avgNR* is defined as a rate reward which is intended to evaluate the steady state average number of requests exist for reading in request queue of service *startRead* in object *b*. It uses operator # which means the number of requests in the service queue. This property can also be evaluated by reward *avgNR2* which examine the property using variable *readers* in object *b* that explicitly models (counts) the number of read requests. The average number of the write operation, the average number of requests waiting for a read operation and write operation can be evaluated by rewards *avgNW*, *avgNWR* and *avgNWW*, respectively. The simulation results with 99% of confidence level within the confidence interval 0.1 compared to the analytic results (both obtained using PDETool) are presented in Table 1.

```
class customer{
        buffer b;
        immediate service customer(buffer b){
                        this.b=b;
                        accessbuffer();
        }
        service accessbuffer():Exp(2){
                case{
                        0.9: b.startread(this);
                        0.1: b.startwrite(this);
                }
        }
        immediate service release(){
                accessbuffer();
        }
}

class buffer{
        int readers, writer;
        condition icustomers, iwriter;
        immediate service buffer (){
                readers=0;
                writer=0;
                icustomers=false;
                iwriter=false;
        }
        service startread(customer
c):Deterministic(0){
                precondition: not iwriter;
                readers++;
                icustomers=true;
                finishread(c);
        }
}
        service startwrite(customer c):Deterministic(0){
                precondition: not icustomers;
                writer++;
                iwriter=true;
                icustomers=true;
                finishwrite(c);
        }
        service finishread(customer c):Exp(1){
                readers--;
                if (readers==0){icustomers=false;}
                c.release();
        }
        service finishwrite(customer c):Exp(1){
                writer--;
                iwriter=false;
                icustomers=false;
                c.release();
        }
}

main(){
        buffer b();
        customer c1(b),c2(b),c3(b),c4(b);
}

rewards{
        avgNR:steadyState {  -if(true){return b.finishread#;}
}
        avgNW:steadyState {  -if(true){return
b.finishwrite#;} }
        avgNR2:steadyState {  -if(true){return b.readers;} }
        avgNW2:steadyState {  -if(true){return b.writer;}}
        avgNWR:steadyState {  -if(true){return
b.startread#;} }
        avgNWW:steadyState { -if(true){return
b.startwrite#;}
}
```

***Figure 4.A Readers-Writers model in RayLang***

***Table 1.Evaluation of rewards in Readers-Writers modelusing PDETool***

| Reward ID | Simulation result | Variance | Analytic result |
|---|---|---|---|
| avgNR | 2.5715 | 0.0274 | 2.5344 |
| avgNW | 0.0978 | 0.0067 | 0.0998 |
| avgNR2 | 2.5715 | 0.0274 | 2.5344 |
| avgNW2 | 0.0978 | 0.0067 | 0.0998 |
| avgNRW | 0.1622 | 0.0133 | 0.0164 |
| avgNWW | 0.6618 | 0.0158 | 0.7060 |

## 7. Comparisons

In contrast to most modeling formalisms, such as Petri nets and process algebras, in which the modelers deal with some low-level primitives (*e.g.* transitions, places, tokens and channels *etc.*), in RayLang, the important primitives such as objects, variables and services are high-level. This makes RayLang appropriate for modeling systems with a proper abstraction level. Now, we compare RayLang with Rebeca, which is the most similar formalism. Then, we compare RayLang, with some other existing models.

***RayLang vs. Rebeca***

As mentioned in Section 2, Rebeca is an actor-based language for modeling concurrent and distributed systems. If we ignore stochastic (timing) concept of RayLang, it has some conceptual and syntactic similarity with Rebeca. Even, one may think RayLang as an (stochastic) extension of Rebeca with deep modifications. In RayLang, we have *objects*, *services* and *references* instead of *rebecs*, *massage servers* and *known objects* in Rebeca, respectively. The major differences between these two languages can be explained as follows:

- In Rebeca, each rebec has its own message queue and its internal message servers and all requests for the rebec come into this queue. For the request on the head of the queue, the rebec executes the proper message server. But in RayLang as an object-oriented modeling language, each service can be considered as an active thread which has its own request queue that contains the requests sent for the corresponding service.
- RayLang has two types of services: ordinary services and immediate services. In contrast of Rebeca in which all communications take place by asynchronous message passing and a message server may be executed when a request exists on top of the rebec's queue, in RayLang, ordinary services could be executed if there exists a request in its request queue and its precondition is enabled. Immediate services act like (synchronous remote or local) method calls in programming language and execute immediately whenever they are called.
- RayLang supports additional useful modeling facilities (like broadcast service requests and dynamic objects relationship) which may help modeling realistic systems like service-oriented systems and computer networks.

***Comparison with Other Formalisms***

A comparison between RayLang and other existing related formalisms is presented in Table 2.

***Table 2.A Comparison of RayLang with Other Related Models***

| Criteria / Model | $\chi$ [13] | SPNL [26] | OSAN [27] | LOOPN [17] | MoDeST [4] | CSP/CCS [12][20] | Rebeca [25] | RayLang |
|---|---|---|---|---|---|---|---|---|
| Similarity with programming languages | × | × | × | × | ✓ | | ✓ | ✓ |
| Object-based/oriented | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ |
| Synch./asynch. comm. | ✓/× | -/- | -/- | -/- | -/- | ×/✓ | ✓/✓ | ✓/✓ |
| Having non-deterministic extension | ✓ | × | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| Having probabilistic extension | × | × | ✓ | × | ✓ | × | × | ✓ |
| Having timed/stochastic extension | ✓ | ✓ | ✓ | ✓ | ✓ | × | × | ✓ |
| Model checking tool support | ✓ | × | ✓ | × | × | ✓ | ✓ | ✓ |
| Performance evaluation tool support | ✓ | ✓ | ✓ | ✓ | ✓ | × | × | ✓ |

## 8. Conclusions

In this paper, we introduced an object-oriented modeling language called RayLang for stochastic discrete-event systems and formally defined its syntax and semantics. Object-oriented modeling can help modelers through encapsulated constructs. On the other hand, model-based performance and dependability evaluation and formal verification can be used to design more dependable systems. In RayLang, there are active objects (which are instantiated from classes) run concurrently and communicate with each other by requesting services. Each object has its internal variables and thus its internal states. After defining classes and objects, the modeler can evaluate interested measures specified by some reward variables.

The stochastic setting of RayLang models, which we discussed in this paper, can be used for performance and dependability evaluation of stochastic discrete-event systems. The non-deterministic setting of the language can be used for modeling and verification of discrete-event systems via model checking techniques of the model against some specified properties.

We have implemented RayLang within the PDETool by developing a translator which maps RayLang models into the input language of the tool's simulation engine, called SimGine. Numerical analysis of RayLang models (where the model satisfies Markovian properties) can be exploited by generating the state space of the model in the tool and solving the obtained CTMC.

Currently, we allow simple data types. More object-oriented properties (*e.g.* inheritance) and supporting rich data types can be added to the language in future extensions of RayLang.

## 9. References


[1] G. Agha,I. Mason, S. Smith and C. Talcott, "A Foundation for Actor Computation," Journal of Functional Programming, Vol. 7, No. 01, pp. 1-72, 1997.

[2] M. AjmoneMarsan,G. Balbo and G. Conte, Performance Models of Multiprocessor Systems, MIT Press, pp. 142-188, 1986.

[3] K. Al-Begain, G. Bolch and H. Herold, Practical Performance Modeling: Application of the MOSEL Language, Kluwer Academic Pub., 2001.

[4] H. Bohnenkamp,H. Hermanns,J. Katoen and R. Klaren,"The MoDeSTModeling Tool and Its Implementation", Lecture Notes in Computer Science, Vol. 4224 pp. 116-133, 2003.

[5] J. Bowen and M. Hinchey, "Ten Commandments of Formal Methods... Ten Years Later",IEEE Computer, Vol. 39, No. 1, pp. 40-48, 2006.

[6] D. Buchs and N. Guel,"A Concurrent Object-Oriented Petri Nets Approach for System Specification", In Proceedings of the 12thInternational Conference on Application and Theory of Petri Nets, pp. 432-454, 1991.

[7] G. Chiola,C. Dutheillet, G. Franceschinis and S. Haddad, "Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications", IEEE Transactions on Computers, Vol. 42, No. 11, pp. 1343-1360, 1993.

[8] P. CsabaÖlveczky and J. Meseguer,"Real-Time Maude: A Tool for Simulating and Analyzing Real-Time and Hybrid Systems", Electronic Notes in Theoretical Computer Science, Vol. 36, pp. 361-382, 2000.

[9] N. Guel, O. Biberstein, D. Buchs, E. Canver, M. Gaudel,F. von Henke and D. Schwier, Comparison of Object-Oriented Formal Methods, TechnicalReport, Second Year Report of Esprit Long Term Research Project 20072, 1997.

[10] C. Hewitt, Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, TechnicalReport, Artificial Intelligence Technical Report 258, Department of Computer Science, MIT, 1972.

[11] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, 1996.

[12] C.A. Hoare, Communicating Sequential Processes, Communications of the ACM, Vo. 21, pp. 666-677, 1978.

[13] A. Hofkamp and J. Rooda, Chi 1.0 Reference Manual, Systems Engineering Group, Eindhoven University of Technology, Rev-1322 Edition, 2007.

[14] K. Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Monographs in Theoretical Computer Science,Vol. 1-3, An EATCS Series Springer-Verlag, Vol. 97, 1992.

[15] A. Khalili, A. Bidgoly and M. Abdollahi Azgomi, "PDETool: A Multi-Formalism Modeling Tool for Discrete-Event Systems Based on SDES Description", In Proceedings of the 30thInternational Conference on Application and Theory of Petri Nets and Other Models of Concurrency, Paris, France, Lecture Notes in Theoretical Computer Science, Vol. 5606,Springer, pp. 343-352, 2009.

[16] C. Lakos,"From Coloured Petri Nets to Object Petri Nets", In Proceedings of the 16thInternational Conference on Application and Theory of Petri Nets, Turin, Italy, June 26-30, Springer-Verlag, pp. 278-297, 1995.

[17] C. Lakos and C. Keen, "LOOPN-Language for Object-Oriented Petri Nets, In Proceedings of the SCS Multi-Conference on Object-Oriented Simulation, Simulation, Series 23, Anaheim, California, pp. 22-30, 1991.

[18] P. Lincoln,N. Marti-Oliet and J. Meseguer, "Specification, Transformation, and Programming of Concurrent Systems in Rewriting Logic", In Specification of Parallel Algorithms, DIMACS Workshop,pp. 309-339, 1994.

[19] J.F. Meyer and W.H. Sanders, "A Unified Approach for Specifying Measures of Performance, Dependability, and Performability", In Proceedings of the Dependable Computing for Critical Applications,Vol. 4, Springer Verlag, pp. 215-237, 1991.

[20] R. Milner,ACalculus of Communicating Systems, Springer-Verlag, 1982.

[21] M.Molloy,"Performance Analysis Using Stochastic Petri Nets", IEEE Transactions on Computers, Vol. 100, No. 31, pp. 913-917, 1982.

[22] A. Movaghar, Performability Modeling with Stochastic Activity Networks, Ph.D. Thesis, University of Michigan, Ann Arbor, 1985.
[23] J. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall,1981.
[24] M. Sirjani, F. de Boer, A. Movaghar and A. Shali, "Extended Rebeca: AComponent-Based Actor Language with Synchronous Message Passing, In Proceedings of the 5th International Conference on Application of Concurrency to System Design, pp. 212-221, 2005.
[25] M. Sirjani,A. Movaghar, A.Shali and F. de Boer, "Modeling and Verification of Reactive Systems Using Rebeca,FundamentaInformaticae, Vol. 63, No. 4, pp. 385-410, 2004
[26] C. Hirel,B. Tuffin and K.S. Trivedi, "SPNP: Stochastic Petri Nets, Version 6.0", In Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS'00), Boudewijn R. Haverkort, Henrik C. Bohnenkamp, and Connie U. Smith (Eds.), Springer-Verlag, London, UK, UK, pp. 354-357, 2000.
[27] M. Abdollahi Azgomi and A. Movaghar, "Modeling and Evaluation with Object Stochastic Activity Networks", In Proceedings of theFirstInternational Conference on the Quantitative Evaluation of Systems (QEST'04), pp. 326-327, 2004.