SimGine: A simulation engine for stochastic discrete-event systems based on SDES description

Ali Khalili, Mohammad Abdollahi Azgomi and Amir Jalaly Bidgoly

SIMULATION published online 19 March 2013
DOI: 10.1177/0037549712473512

Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
0(0) 1–17
© 2013 The Society for Modeling and Simulation International
DOI: 10.1177/0037549712473512
sim.sagepub.com

SimGine: A simulation engine for stochastic discrete-event systems based on SDES description

Ali Khalili[1], Mohammad Abdollahi Azgomi[1] and Amir Jalaly Bidgoly[1]

Abstract
Discrete-event systems have gained a lot of interest due to their wide range of applications, and discrete-event simulation is a useful method for the performance evaluation of such systems. In this domain, model-based evaluation methods play an important role and there are many formalisms and realistic experiments using these methods. In this paper, we introduce SimGine, a multi-formalism simulation engine for stochastic discrete-event systems based on SDES, which is a unified abstract description for stochastic discrete-event systems. The engine is also capable of rare-event simulation of models using the importance sampling technique, which makes it the first multi-formalism simulation tool with rare-event simulation capability. The XML-based input language of SimGine allows for definition of the required methods. The body of each method is expressed by codes in a high-level programming language and this provides a powerful and flexible approach for defining events with complex behavior. For the simulation of an existing model, a tool for translating models into the SimGine input language should be prepared. SimGine can be used as a stand-alone simulation tool or as a simulation engine in other tools.

Keywords
Stochastic discrete-event systems, simulation engines, formal methods, discrete-event simulation

1. Introduction

The behavior of many systems which are increasingly important in our lives could be described as discrete-event systems. These systems have gained a lot of interest due to their wide range of applications. Numerous models and analysis algorithms have been developed and successfully put into practice. Stochastic discrete-event systems which can capture some randomness in event delays and probabilistic behavior are developed for evaluating quantitative measures of systems (such as dependability and performance).

There are many formalisms for stochastic discrete-event systems, such as stochastic extensions of Petri nets and stochastic process algebras.These methods can be used for the performance and dependability evaluation of many systems. All the various proposed discrete-event model classes share some common characteristics and many of the techniques that have been developed for one class are in principle applicable to all or most of them. SDES is a unified abstract description for stochastic discrete-event systems which can be regarded as a blueprint for an abstract data type with virtual elements instantiated for a certain model class by substituting the attributes with net-class dependent values and functions.[1]

For the evaluation of these discrete-event models, there are at least two common approaches: analytic solution and model simulation. Analytic methods, which usually are capable of providing exact solutions, require the models to conform to certain criteria. One of the major difficulties that analytic approaches suffer from is the state-space explosion. Another problem is that analytic approaches, which use numerical techniques, can only deal with models that satisfy the Markovian property. As a result, discrete-event simsimulationulation plays an important role in the evaluation of systems and remains a popular method despite the development of new analytic methods.

[1]School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

A. Khalili is now at the Italian Institute of Technology (IIT), Genova, Italy.

Corresponding author:
M.A. Azgomi, School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran 16846-13114, Iran.
Email: azgomi@iust.ac.ir

There are many modeling tools available and each of them is able to create and evaluate models of a special formalism. In this paper, we introduce SimGine, a simulation engine for discrete-event systems based on SDES description. This engine can be used as a framework for the simulation of stochastic discrete-event systems. To utilize this engine, the formalism and configuration (i.e. all required information including events, guards, reward variables, etc.) should be translated into the input language of the engine. The engine can be used as a stand-alone tool or as a simulation library in other tools for the evaluation of stochastic discrete-event systems. It is also capable of rare-event simulation of models using the importance sampling (IS) technique. Rare-event simulation is a key technique in mission- and safety-critical applications in which the traditional simulation techniques are not applicable. As far as we know, SimGine is the first multi-formalism simulation tool which is capable of simulating rare events.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the related works. Section 3 gives the motivations of the work. Section 4 describes SimGine's architecture and its input language. Enabling SimGine to simulate rare events is described in Section 5. The results of simulating two examples are given in Section 6. Section 7 presents a comparison of SimGine with other multi-formalism simulation tools. Finally, some concluding remarks are presented in Section 8.

2. Related work

The modeling and simulation of complex systems requires the availability of appropriate modeling formalisms and tools. There are many modeling tools available, most of which support a single high-level formalism to evaluate the models based on one or more solution methods. Many of these tools not only support a limited number of formalisms, but also are specific to some application areas. TimeNet[2] and UltraSAN[3] are some famous examples of simulation or modeling tools supporting single formalism based on some specific extensions of stochastic Petri nets.

However, in many cases, no single analysis and modeling method can successfully cope with all aspects of a complex system and a multi-formalism approach which tries to achieve formalisms and techniques integration is very appealing.[4] The earliest attempt in this direction, to the best of our knowledge, was the combination of multiple modeling formalisms in SHARPE.[5] It is a tool for specifying and analyzing performance, reliability, and performability models, and is capable of evaluating multiple model types including generalized stochastic Petri nets (GSPNs), product form queuing networks, reliability block diagrams, and Markov chains, by simulation or analytic solution methods.

SMART[6] is another software tool that integrates multiple modeling formalisms. It supports the analysis of models expressed in stochastic Petri nets and queuing networks. The aim of SMART is to implement the tool in a way that permits the easy integration of new solution techniques.

Möbius[7] is an environment in which multiple modeling formalisms and solvers can interact, and a framework for building a software tool for dependability and performance evaluation of complex discrete-event systems. The aim of the Möbius framework was to develop a tool in which a number of different modeling formalisms and solution techniques can be integrated into a single modeling tool or software environment. The simulation engine of the Möbius modeling framework is a formalism-independent simulator to support different execution policies and to provide a graphical interface to launch and monitor simulation progress.[8]

OsMoSys[4] is a multi-formalism, multi-solution, object-oriented modeling framework based on meta-modeling, which provides an effective means of achieving both explicit and implicit multi-formalism, as well as compositional modeling within a single formalism. In this framework, the meta-formalism is a language used to describe graph-based formalisms; in other words, formalisms whose elements are nodes and arcs, such as Petri nets, queueing networks, and fault trees.

MOSEL-2[9] is a modeling environment which comprises a high-level modeling language that provides a simple way to describe stochastic processes, which can evaluate the model by numerical analysis or simulation. The environment reuses existing tools for the system analysis by translating MOSEL-2 model specifications into the tool-specific system descriptions of some third-party modeling tools including MOSES, SPNP, and TimeNET.[10]

Discrete-event system specification (DEVS) formalism[11] is a general methodology for describing discrete-event systems whose states and input/output behavior can be described by sequences of events. It can be used to describe and simulate many classes of deterministic systems, including discrete-event systems. Stochastic behavior is achieved in DEVS simulations by modeling behavior resorting to pseudo-random generators (i.e. deterministic sequences with properties acceptable enough to consider them stochastic for the practical purposes at hand). In Sarjoughian and Elamvazhuthi,[12] the authors have developed an integrated modeling and simulation tool called 'component-based system modeler and simulator' (CoSMoS) to combine (visual and logical) model development and simulation execution. CoSMoS is not a simulator or a simulation engine but it facilities the design of systems by visually developing logical models, and is able to translate models into Parallel-DEVS notation, which can be simulated by DEVS-Suit.[13] The same approach can be applied to other modeling approaches and simulation engines,[12] and for the sake of generality and uniformity of SDES notation, SimGine might be considered as an alternative for this purpose.

Stochastic DEVS specification (STDEVS)[14] is a generalization of DEVS for stochastic systems based on the use of probability spaces. It provides a formal framework with a strong mathematical basis for modeling generalized nondeterministic discrete-event systems. It has been demonstrated that DEVS is a particular case of STDEVS for deterministic models. Therefore, computational implementations of DEVS simulation models resorting to pseudorandom number generators used in practice to mimic stochasticity are theoretically well formulated in the context of probability theory by means of STDEVS. In practice, STDEVS is an alternative to SDES, considered as the unified notation for the analysis of discrete-event systems in a simulation tool. When comparing SDES with STDEVS, we note that the SDES description only supports single models and does not handle model hierarchy in its original description (nevertheless, we are still able to extend it with a kind of model composition, e.g. by sharing variables among models), and to evaluate hierarchical high-level models, we have to flatten hierarchical compositions into a single SDES model. However, STDEVS has the inherent capability of composing models hierarchically, being therefore very suitable for modeling systems with different communicating components. On the other hand, STDEVS does not explicitly deal with features like immediate activities (with global probabilistic collision control) and activities' reactivation. Despite the fact that DEVS gains lots of interest in the literature, which shows its generality, elegant design, and good theoretical basis, we believe that SDES is a flexible and general notation and a more convenient option to be used as a unified formalism in a simulation engine (where the formalism is likely aimed by an automated translation).

Considering the capabilities and limitations of the mentioned works, the next section describes our aims of developing SimGine and later, in Section 7, a comparison between SimGine and the most related of these works will be provided.

3. Motivations and aims

The aim of a multi-formalism modeling tool is to support several specification language, model composition, and solution or simulation methods within an integrated environment. We believe that it is practically impossible to build a tool which can support the increasing number of modeling formalisms. Despite the advantage of available multi-formalisms modeling tools, this goal is not yet achieved practically. Many of the current multi-formalism modeling tools are not extensible, or are not generally available to other users for possible extensions. A few of the others, like Möbius, provide the possibility of extension for users. However, adding a new formalism to these frameworks is not an easy task.

From another viewpoint, discrete-event simulation faces some challenges. In mission- and safety-critical applications, the existence of some interested rare events makes the traditional discrete-event simulation methods inefficient. Standard simulation techniques are not practical in such applications, due to the required long simulation time. In such situations, rare-event simulation techniques are employed, which increase the occurrence probability of rare events, usually by utilizing an accelerator. However, rare-event simulation techniques have not yet gained sufficient interest in modeling and simulation tools despite their obvious advantages which are vital for industrial applications.

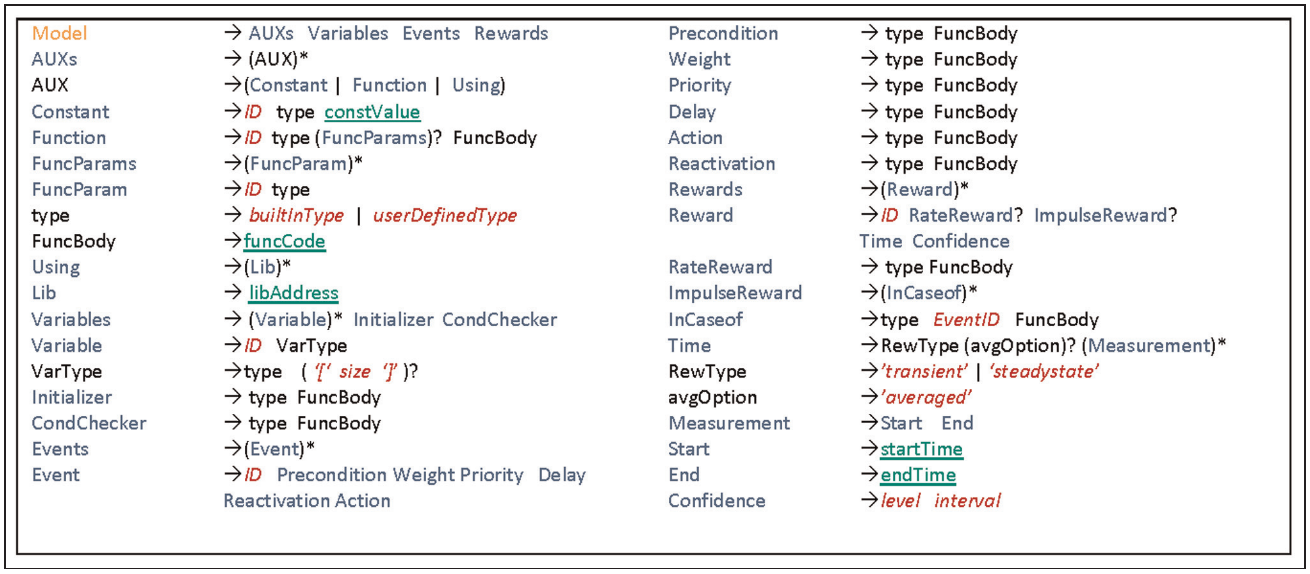
Our aim has been to develop a multi-formalism simulation engine, called SimGine, for stochastic discrete-event systems, which is extensible for use by third-party applications. The engine is based on SDES, a unified and easily understandable stochastic discrete-event description. SimGine can be used both as a stand-alone simulator and as a simulation library using the application programming interface (API). It uses an XML-based input language, which specifies the elements and the behavior of models using the programming language C#, which makes the engine suitable for modeling complex variables and behaviors. It facilities step-by-step simulation to help make model-debugging easier. In addition, it is capable of simulating rare events using IS, the most famous technique in rare-event simulation. To the best of our knowledge, SimGine is the first effort in utilizing rare-event simulation techniques in a multi-formalism modeling tool.

4. SimGine: The new simulation engine

In this section, we introduce SimGine, the simulation engine that is based on SDES description. As we mentioned before, most stochastic discrete-event formalisms can be translated into SDES description, and thus, they can be simulated by SimGine. For each formalism, a translator should be developed to translate the corresponding model into the input language of SimGine.

4.1. The input language

The syntax and semantics of SimGine's input language is designed to resemble the SDES description. Due to the benefits of extended mark-up language (XML), especially human- and machine-readability, representation of the input language is based on XML where each element of the language is represented by means of an XML tag that may contain some attributes and a value. Figure 1 shows the BNF-like grammar of the input language. In this figure, words in italics represent XML attributes and underlined words denote XML values in the language. Like any other



Model	→ AUXs Variables Events Rewards	Precondition	→ type FuncBody
AUXs	→ (AUX)*	Weight	→ type FuncBody
AUX	→(Constant | Function | Using)	Priority	→ type FuncBody
Constant	→*ID* type <u>constValue</u>	Delay	→ type FuncBody
Function	→*ID* type (FuncParams)? FuncBody	Action	→ type FuncBody
FuncParams	→(FuncParam)*	Reactivation	→ type FuncBody
FuncParam	→*ID* type	Rewards	→(Reward)*
type	→ *builtInType* | *userDefinedType*	Reward	→*ID* RateReward? ImpulseReward? Time Confidence
FuncBody	→<u>funcCode</u>		
Using	→(Lib)*	RateReward	→ type FuncBody
Lib	→ <u>libAddress</u>	ImpulseReward	→(InCaseof)*
Variables	→ (Variable)* Initializer CondChecker	InCaseof	→type *EventID* FuncBody
Variable	→*ID* VarType	Time	→RewType (avgOption)? (Measurement)*
VarType	→type (*'['* *size* *']'*)?	RewType	→*'transient'* | *'steadystate'*
Initializer	→ type FuncBody	avgOption	→*'averaged'*
CondChecker	→ type FuncBody	Measurement	→Start End
Events	→(Event)*	Start	→<u>startTime</u>
Event	→*ID* Precondition Weight Priority Delay Reactivation Action	End	→<u>endTime</u>
		Confidence	→*level* *interval*

Figure 1. BNF-like grammar of input language of SimGine (as XML format).

XML document, the grammar of the input language can also be expressed by a document type definition (DTD).

A model includes the definitions of the state variables of the system, events, and reward structures, corresponding to $S*$, $A*$, and $RV*$ in the SDES definition, respectively. In addition, a model may include some auxiliaries, which contain the definitions of the required constants, utility functions, and external source code needed in the simulation progress. Each utility function can be called by functions defined in the scope of the model. A utility function definition includes an *identifier, parameters*, and a *body*. The body of each function defines the type of return value of the function and a quoted string in an XML tag value which is written in a programming language and can include loops, local variable definitions, conditional statements, assignment statements, etc. The modeler can also import the required source code in some external files with the tag *Using* which specifies the location of the file containing the source code. This capability is useful for defining external functions (especially when they have numerous lines of code) and user-defined data types by means of class definition, which can also be used in other models; it makes them reusable and more structured. For example, the modeler can define some classes for queue, stack, and set data structures to be used in the model. In the current version of the engine, the programming language C#.NET is selected for the input language and all parts of the model definition should follow the syntax and semantics of this language. For example, naming convention, method calls rules, and scope rules are the same as for C#.

State variables of the model should be defined in the *Variables* section. Each variable within a special type has a unique variable name (identifier). The type of each state variable can be a built-in type (e.g. bool, int, char, byte, float, etc.), an array of a built-in type, or any type defined by the user in the auxiliary (i.e. *Using* part). State variables can be used in any functions, but assigning a new value to them is permitted only in the *action* function defined for each event. The initial state-space of the model (Val_0* in SDES notation) must be defined within the *initializer*, which is a function that does not return any value and initializes the state variables of the model. This method is executed before any trial of the simulation to initialize the model. The variables section may also include the *CondChecker* function to check and validate the values of state variables that will be executed after the execution of each event. This function, which implements $Cond^*$, can be used when all values belonging to a sort of state variable are not actually allowed.

The *Events* part includes events' definitions of the models (A^* in SDES notation). An event represents the basic unit of a model that facilitates changing the state of the system by modifying values of state variables. An event corresponds to a transition in stochastic Petri nets (like SPNs[15] and GSPNs[16]), an action in process algebras, an activity of a SAN,[17] or a server of a queue in queueing networks, for example. In the current version of the engine, the definition of events in the input language is the same as SDES actions but with two differences. First, we have limited the degree of each action to one and it has exactly one variant and thus, we do not distinguish between an action and its modes (see the definition of SDES in Appendix and Zimmermann[1]). And second, we have extended the definition of actions in SDES ($A*$) with the reactivation[18] concept. An action may reactivate as a state-dependent function while it is enabled. The action must start over when reactivation occurs and a new execution time must be chosen. Each event has a unique identifier.

Table 1. Types of functions' return values.

Function	Type (of return value)
initializer	void
condition checker	bool
precondition	bool
weight	float
priority	int
delay	float
action	void
reactivation predicate	bool
rate reward	float
impulse reward	float

The following methods must be defined for each event (Table 1 shows the type of each method):

- Precondition: the body of a boolean function that determines the precondition of an event (*Ena** in SDES notation) in each state evaluates whether the event is enabled or not. While the precondition of an event is *false*, the event will not be executed.
- Delay: the body of a function that specifies the execution time of the event (*Delay** in SDES notation), in other words, the time that must elapse before the event executes, while it is enabled. Among all enabled events, the event with the earliest execution time will complete first. If the delay of an enabled event in a state is equal to zero, that event is regarded as an immediate event (like an immediate transition in stochastic Petri nets).
- Priority: the body of a function which specifies the priority of the event (*Pri** in SDES notation). If two or more events are enabled and have the same execution time, the event with the highest priority executes first.
- Weight: if two or more events with the same execution time share the highest priority, a probabilistic decision determines which event completes first. The weight function (i.e. *Weight** in SDES notation) is mostly used for immediate events.
- Action: the body of the completion function of the event which may change the state of the model (*Exec** in SDES notation). If the model has more than one enabled event, the simulator needs a selection algorithm for choosing the next event to complete. The simulation algorithm, including this selection process, is shown briefly in Figure 2 (due to lack of space, the detailed algorithm is omitted and just the general procedure is in the figure. For more details, please see SimGine documentation[19]).
- Reactivation predicate: the body of a function that returns a boolean value as the reactivation predicate. If the reactivation predicate of an

```
Algorithm 1: Event-scheduling time-advance algorithm of SimGine
1. Initialize model's variables by executing function initializer
2. While the specified stop criteria is not reached, do:
    a) Determine the set of enabled events $EL$ in the current state, $EL = \{e | \forall e \in A^*, pre_e = true\}$.
       The set of newly enabled events $NL$ is $EL - EL'$ where $EL'$ is the set of enabled events in the previous round.
    b) Determine the set of reactivated events $RL$ which contains enabled events whose reactivation
       predicate are true, $RL = \{e | e \in EL \wedge reac_e = true\}$.
    c) Generate the execution time for newly enabled and reactivated events: $\forall e \in (NL \cup RL)$ :
       $T_e = delay_e$.
    d) Considering the remaining time of all enabled events ($EL$) and select events $SE$ with the
       least remaining time for completion ($SE = \{e | e \in EL \wedge \neg\exists e' \in EL : (T_{e'} > T_e \wedge e \neq e')\}$).
    e) If more than one event is selected ($|SE| > 1$), compute the priority of each event and select
       the events with the highest priority ($SE = \{e | e \in SE \wedge \neg\exists e' \in SE : (prio_{e'} > prio_e \wedge e \neq e')\}$).
    f) If still more than one event are selected ($|SE| > 1$), compute the weight of each event and
       select one of them probabilistically. The selection probability of each event $e$ is as follows:
                    $P(e) = \frac{weight_e}{\Sigma_{e' \in SE} weight_{e'}}$
       where $weight_e$ is the weight of event $e$ (after this selection, $SE$ would only contain the selected
       event $s$ and $|SE| = 1$)
    g) Observe the reward variables for the completion time of the selected event.
    h) Update the confidence mean of the reward variables.
    i) Fire the action method for the selected event by executing $action_s$.
    j) Advance the simulation clock to the completion time of the selected event (and decrease the
       remaining time for all enabled events, i.e. $\forall e \in EL, T_e = T_e - T_s$).
```

Figure 2. Event-scheduling time-advance algorithm of SimGine.

```
<Event ID="event1">
        <Precondition type="bool"> "{if(p1==p2)return true; else return false;}" </Precondition>
        <Weight type="float">"{return 1;}"</Weight>
        <Priority type="int">"{return 1;}"</Priority>
        <Delay type="float"> "{return Distributions.Exponential(1.5);}" </Delay >
        <Action type="void"> "{p2--;p1++;}" </Action>
        <Reactivation type="bool"> "{return false;}" </Reactivation >
</Event>
```

Figure 3. An event example.

enabled event holds in a state, the event restarts. Reactivation causes the event to behave like when it disables and then enables again immediately.

Figure 3 shows an example of a simple event named *event1* which has a precondition that makes it enabled when the state variable *p1* is equal to the state variable *p2*. The weight and the priority of *event1* are equal to 1. The delay function, as described before, must return a float value, and a predefined class (named *Distribution*) is available for the easy usage of the probability distributions. This class has some probability distribution functions, such as deterministic, exponential, Erlang, gamma, hyper-exponential, and lognormal. The execution rate of *event1* is exponential with the rate 1.5 and it causes the incrementation of *p1* and the decrementing of *p2* by one.

In the *Rewards* section, the modeler defines the interesting reward variables (i.e. $RV*$ in SDES notation) of the model to aid the evaluation of measures about the system. In this direction, SimGine supports reward variables at some instant or interval of time or in the steady-state.

The reward formalism supported by SimGine is similar to that in Sanders and Meyer.[20] There are two classes of reward functions: impulse and rate reward functions that specify impulse and rate rewards, respectively. For each reward variable, the following items are required to be specified:

- Rate reward: a rate reward function is defined by the tag *RR* and specifies the rate at which the reward is accumulated for any state of the model. When the model is in a specific state, if the rate reward function returns the float value *r*, it means that while the model remains in that state, the reward will be accumulated at the rate *r*.
- Impulse reward: impulse reward functions are specified by the tag *IR*. An impulse reward function specifies the amount of the reward earned when the state of the model changes by the execution of some particular events. For the observation of every desired event, a tag *InCaseOf* should be defined, which contains the body of a function that its return value characterizes the amount of the reward earned upon the execution of the event, where its name is specified in the tag's attribute *event*. Impulse rewards are usually used to count the number of executions of an event during an interval of time.
- Reward variable type: different problems need various classes of reward variables. Therefore, SimGine supports two types: *transient* and *steady-state*. The steady-state reward needs no more data, while the simulator requires more information when the transient type is selected, which must be defined in the tag *Measurement* that describes an interval of time that the reward must be earned in. The start and end points of an interval can be the same corresponding to an *instant-of-time* reward variable. The transient type can also have an option expressed as *averaged*. Using this option, the simulator computes the mean value of the reward in the specified time-interval (like an averaged interval-of-time reward variable in some other tools).
- Confidence: when the simulator reaches the desired confidence interval in the specific confidence level for all reward variables, the simulation progress will be stopped. The confidence level can be 0.95, 0.98, 0.99, or 0.999. Greater levels with smaller intervals lead to more reliable results.

Figure 4 shows an example of the reward structure in which a reward named *rew1*, composed of a rate reward and an impulse reward, is defined. The rate reward probes the state variable *p1* and causes *rew1* to earn one whenever it is equal to 3, in other words, the reward earned per unit of time in each state that holds the property '$p1 = 3$' is equal to 1. Also, the impulse reward causes the accumulation of *rew1* with 1 or 0.05 whenever *event1* or *event2* is executed, respectively.

4.2 The software architecture of SimGine

In this section, we briefly review the software architecture of SimGine. The overall software architecture of the tool, including its main components and their relationships, is depicted in Figure 5. The three main components of the engine are the model parser, code generator, and SDES

```
<Reward ID="rew1">
        <RateReward  type="float">  "{if(p1==3)return 1;return 0}"  </RateReward >
        < ImpulseReward >
                  <InCaseOF type="float" eventID="event1">  "{return 1;}"  </InCaseOF>
                  <InCaseOF type="float" eventID="event2">  "{return 0.05;}"  </InCaseOF>
        </ImpulseReward >
        <Time type="transient" option="averaged">
             <Measurement>
                   <Start>10</Start>
                   <End>100</End>
             </Measurement>
             <Measurement>
                   <Start>100</Start>
                   <End>120</End>
             </Measurement>
        </Time>
        <Confidence level="0.95" interval="0.1" />
 </Reward>
```

Figure 4. A reward structure example.

simulation manager. The input model is an XML file based on the input language, usually translated from a model in a high-level formalism. The translator, which is a third-party application, has the responsibility of translating high-level models into the input language. In the first step, the input model is checked by the model parser to validate syntax and semantic rules, in which the necessary information is prepared for the code generator. The code generator is responsible for constructing an executable code which includes the description of the input model, required for running the simulation. The simulation manager advances simulation progress and collects the information while running the simulation until it reaches a stop-criterion (such as the desired confidence interval, the maximum number of iterations, or any false value in the results of the condition checker). The SDES simulation manager acts as the underlying layer for simulator, animator, and rare-event simulator (described in the next section) sub-components. There is also a utility library which provides some utility functions available for all components of SimGine. For more information about software architecture, classes, and UML diagrams of SimGine, please see the SimGine developers' manual.[19]

SimGine can be used as a simulation engine library in third-party applications. For the simulation of a model, some objects and their methods provided by the library can be used. A graphical user interface (GUI) has also been developed that interacts with SimGine and so the engine can be used as a stand-alone simulation tool. This GUI allows users to edit SDES models and define simulation studies easily. In addition, the GUI provides features like step-by-step simulation (two screenshots of SimGine are shown in Figures 6 and 7). The SimGine GUI makes the modelers able to work with SDES models (as the input language of SimGine) and, as mentioned before, for working with high-level models, we need another application to

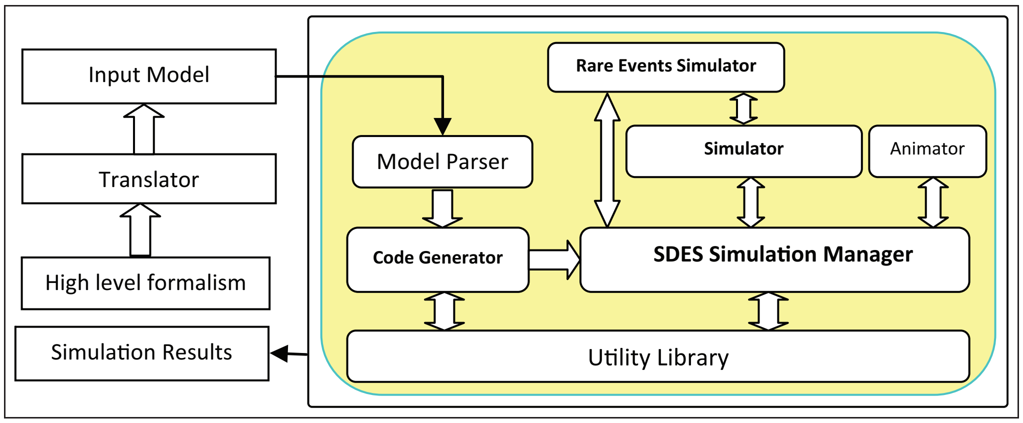
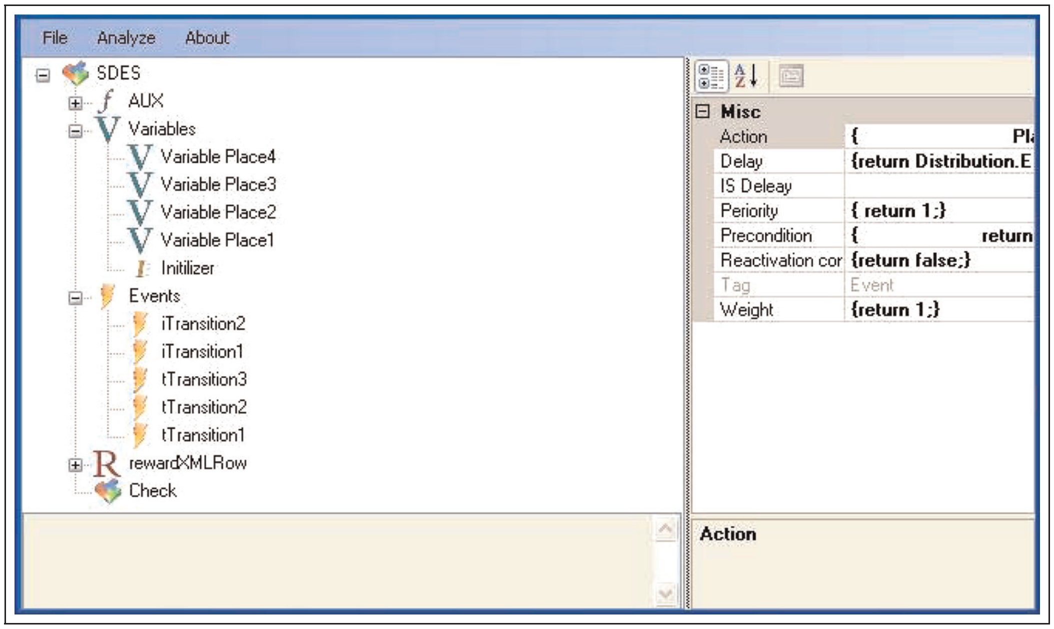


Figure 5. Software architecture of SimGine.

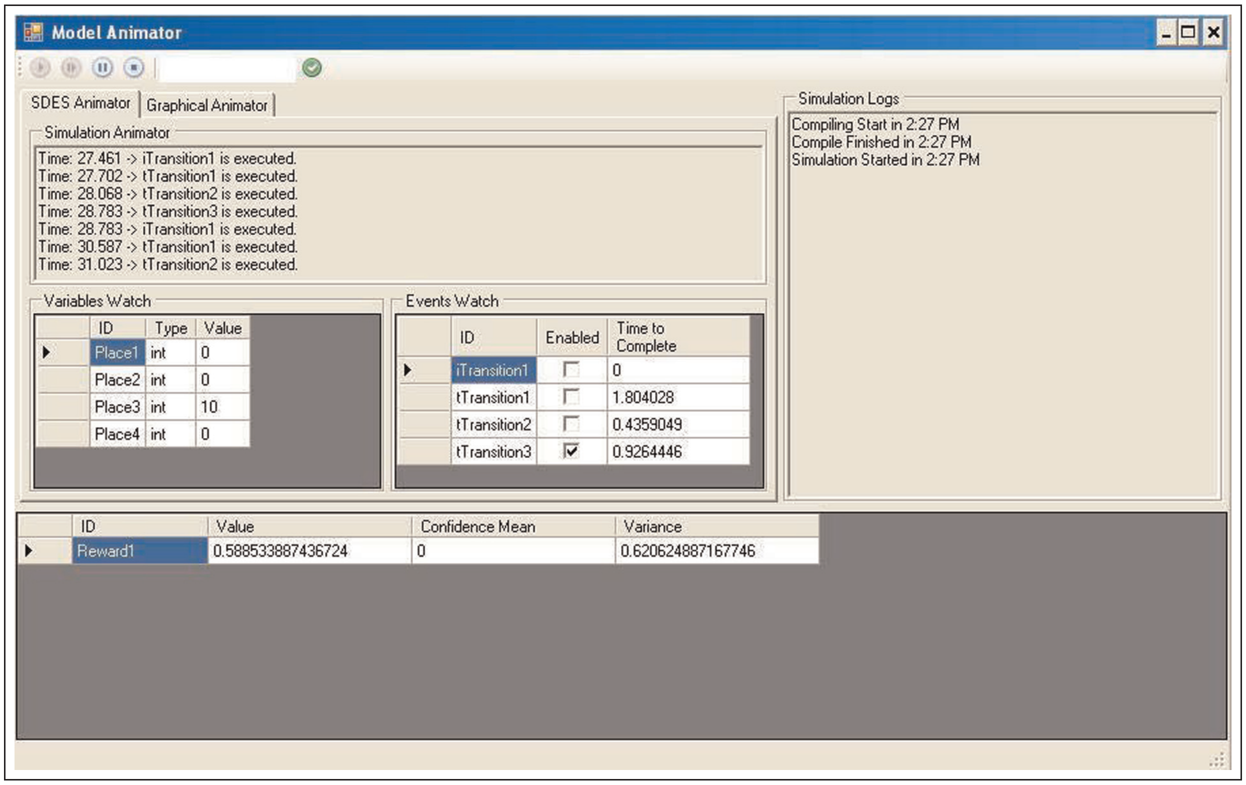


```
SDES
  AUX
  Variables
    Variable Place4
    Variable Place3
    Variable Place2
    Variable Place1
    Initilizer
  Events
    iTransition2
    iTransition1
    tTransition3
    tTransition2
    tTransition1
  rewardXMLRow
  Check
```

Misc	
Action	{ Pl
Delay	{return Distribution.E
IS Deleay	
Periority	{ return 1;}
Precondition	{ return
Reactivation cor	{return false;}
Tag	Event
Weight	{return 1;}

Action

Figure 6. A screenshot of the GUI of SimGine for editing models.



```
Time: 27.461 -> iTransition1 is executed.
Time: 27.702 -> tTransition1 is executed.
Time: 28.068 -> tTransition2 is executed.
Time: 28.783 -> tTransition3 is executed.
Time: 28.783 -> iTransition1 is executed.
Time: 30.587 -> tTransition1 is executed.
Time: 31.023 -> tTransition2 is executed.
```

```
Compiling Start in 2:27 PM
Compile Finished in 2:27 PM
Simulation Started in 2:27 PM
```

ID	Type	Value
Place1	int	0
Place2	int	0
Place3	int	10
Place4	int	0

ID	Enabled	Time to Complete
iTransition1	☐	0
tTransition1	☐	1.804028
tTransition2	☐	0.4359049
tTransition3	☑	0.9264446

ID	Value	Confidence Mean	Variance
Reward1	0.588533887436724	0	0.620624887167746

Figure 7. Another screenshot of the GUI of SimGine for the step-by-step simulation of models.

build models and translate them into SDES. In this direction, we have also developed PDETool[21] as an extensible modeling tool to build graphical and textual models and automatically translate them into the input language of SimGine. The aim of the tool is to provide some features for the construction of models and translating them into the input language of SimGine. It also provides features for the simulation and animation of high-level models.

PDETool currently supports stochastic reward nets (SRNs), GSPNs, some extensions of stochastic activity networks (SANs), and RayLang.[22]

5. Rare-event simulation within SimGine

As mentioned before, traditional simulation is not appropriate for systems with rare events due to the required long simulation time. In such situations, rare-event simulation techniques are used, which increase the occurrence probability of rare events, usually by utilizing an accelerator.

IS[23–25] and splitting[26–28] are famous variance reduction techniques for rare-event simulation. IS increases the probability of rare events by changing the probability laws that drive the evolution of the system. The model is then biased, but the bias is compensated for by the introduction of a function called the *likelihood ratio*, which is the ratio of the sample path in the original model to the altered model. The major difficulty in applying this method is to figure out how to change the probabilities, in which the IS estimator has much smaller variance than the original one. In the splitting method, however, the probability laws remain unchanged, but an artificial drift toward rare events is created by terminating some probability trajectories that seem to go away from it and by splitting (cloning) those that are going in the right direction.[27] *RESTART*[29–31] is also a class of splitting which considers a threshold and kills the trajectories when they cross below it.

In this section, we discuss extending SimGine to support rare-event simulation. Among rare-event simulation techniques, IS is chosen for this purpose, being one of the most famous and successful techniques. To do so, we have extended SimGine with a rare-event simulation technique which is capable of simulating models using the IS technique. In the following, IS and its background theory is briefly reviewed, and then we explain how we have extended SimGine to support rare-event simulation. In this regard, in Section 5.2, the required changes in the syntax of the input language of SimGine will be described in detail. Afterward, Section 5.3 gives a technical description of the implementation of IS in the engine.

5.1. Importance sampling

IS[23–25] is one of the most effective and significant methods for rare-event simulation. Consider a model with the sample space Ω, the probability measure P, and a rare event $\varepsilon \in \Omega$. The interest of evaluating the model is estimating the probability $P(\varepsilon)$. We define the indicator function $I(\varepsilon)$ to have the value 1 when outcomes belong to ε and the value 0 in other cases. Let γ denote the probability $P(\varepsilon)$. Consider a new distribution function $P*$ with the property that $P*(A) > 0$ where $P(A) > 0$ for $A \subset \varepsilon$, such that

$$\begin{aligned} P(\varepsilon) &= E_P[I(\varepsilon)] = \int I(\varepsilon) dP = \int I(\varepsilon) \tfrac{dP*}{dP*} dP \\ &= \int I(\varepsilon) \tfrac{dP}{dP*} dP* = \int I(\varepsilon) L dP* = E_{P*}[LI(\varepsilon)] \end{aligned} \tag{1}$$

Here, the quotient of the probability measure $L = dP/dP*$ is called the likelihood ratio. According to this, IS simulation for estimating γ can be performed as follows:[23] generate n independent samples $(I_1(\varepsilon), L_1)$, $(I_2(\varepsilon); L_2)$, $\ldots$, $(I_n(\varepsilon); L_n)$ of $(I(\varepsilon); L)$. Then use the following equation as an unbiased estimator of γ:

$$\gamma_n(P*) = \tfrac{1}{n} \sum_{i=1}^{n} I_i(\varepsilon) L_i \tag{2}$$

In particular, the aim of IS is to find a $P*$ that minimizes the relative error, or equivalently, the variance of the output $I(\varepsilon)L$. Although in some cases a new distribution $P*$ that increases the probability ε might be a proper choice, usually there exist more constraints in selecting a good one. For more information about the characteristics of a good distribution function for IS, please see Juneja and Shahabuddin,[23] Obal,[32] and Nicola et al.[33]

5.2. Rare-event simulation in SimGine

To make SimGine capable of simulating rare events, the syntax of input models is extended to having an optional method *ISDelay* in *Event* definition, which represents the new distribution function (i.e. $P*$ in IS theory reviewed in the previous subsection). It is a more powerful and flexible method for the definition of distribution functions compared to the other tools in this regard (like UltraSAN[3] which uses governor definition).

As we mentioned in Section 5.1, IS uses a quotient called 'likelihood ratio' to achieve an unbiased estimator. Computing the ratio requires knowing not only the value of the event's delay (returned by the *ISDelay* method), but also the exact information about the distribution of the execution time, including its probability density function (PDF) and cumulative distribution function (CDF). Thus, in IS simulation, we restrict events' delay function to some standard distribution functions (i.e. distribution functions like exponential, normal, Weibull, and so on, but not deterministic) instead of unknown general ones. To make the SimGine parser capable of extracting the required information from *Delay* and *ISDelay* functions, their return type must be an explicit method call from the class *Distribution* (e.g. see the delay function defined in Figure 8). The events with a deterministic execution time, like immediate transitions in GSPNs, as well as those which return a float number (rather than using an explicit call from the class *Distribution*), cannot be modified by the *ISDelay* function.

The next subsection gives a technical description of the implementation of IS technique in SimGine.

```
<Event ID="fail_1">
    <Action type="void"> "{ type_1 --; failed_1 ++; }" </Action>
    <Delay type="float"> "{return Distribution.Exponential(.005*type_1);}" </Delay>
    <ISDelay type="float">
        "{if(failed_1+failed_2 >= 4)
          return Distribution.Exponential(25*type_1);
        else if(failed_1+failed_2 >= 3)
          return Distribution.Exponential (3*type_1);}"
    </ISDelay>
    <Precondition type="bool"> "{ return type_1 > 0; }" </Precondition>
    <Priority type="int"> "{return 1;}"</Priority>
    <Reactivation type="bool"> "{ return false;}" </Reactivation>
    <Weight type="float"> "{return 1;}" </Weight>
</Event>
```

Figure 8. An example of an *Event* for rare-event simulation.

5.3. Software implementation of IS in SimGine

The performance measure in a discrete-event simulation is usually determined as a function of a sample path. The term *sample path* refers to the sequence of events that occur during the simulation. Therefore, an event is rare if the sample paths in which it occurs are rare. The IS approach alters the probability measure of the model to lead the simulation to rare-event sample paths and then correct the estimation using the likelihood ratio. The modeler is responsible for choosing a new probability measure (determined by *ISDelay* functions) to be used in IS technique. A good probability measure is the one which decreases the variance of the simulator to the lowest possible value.

To estimate the likelihood ratio, an iterative approach has been presented in Nicola et al.[33] by maintaining an accurate description of the conditional density and distribution functions. We have adapted this approach for IS simulation in SimGine. Consider that the model state in the sample path is presented by $X_i = (\omega_i, e_i)$ where e_i is the ith event in the sample path (starting with e_0) and ω_i presents the internal state of the model before the execution of the mentioned event. Let's assume that t_i denotes the execution time of e_i and $T_i = t_{i+1} - t_i$ while $t_0 = 0$. The probability of the whole sample path is given by equation (3) where $X_{0,n} = (X_0, \ldots, X_n)$ is the sequence of states in the sample path:

$$P(X_{0,n}) = \prod_{i=0}^{n-1} P(X_{i+1}|X_i) \tag{3}$$

Let $f_i(t, e)$ and $F_i(t, e)$ be the PDF and CDF of the remaining execution time of the event e at the state X_i where e is enabled in that state. Also, let $F'(., e) = 1 - F(., e)$. Upon the next event, if the event e has not been executed yet and is still enabled, we will have

$$\begin{aligned} f_{i+1}(t, e) &= \frac{f_i(t + T_i, e)}{F'_i(T_i, e)} \\ F'_{i+1}(t, e) &= \frac{F'_i(t + T_i, e)}{F'_i(T_i, e)} \end{aligned} \tag{4}$$

And, for newly enabled events, we will have

$$\begin{aligned} f_{i+1}(t, e) &= f(t, e) \\ F'_{i+1}(t, e) &= F'(t, e) \end{aligned} \tag{5}$$

Equation (5) gives an iterative function for updating the conditional PDF and CDF in every sequence in the sample path. As in Nicola et al.,[33] $P(X_{i+1}|X_i)$ can be given by equation (6), assuming that E_i is the set of enabled events in the state X_i, as follows:

$$P(X_{i+1}|X_i) = f_i(T_i, e_i) \prod_{e \in E_i - \{e_i\}} F_i(T_i, e) \tag{6}$$

In the IS technique, the likelihood ratio is defined as the ratio of the probability of a sample path in the original model to the probability of the same path in the biased model. Thus, the likelihood ratio in the ith state is then given by

$$L(X_{i+1}|X_i) = \frac{f_i(T_i, e_i)}{f_i^*(T_i, e_i)} \prod_{e \in E_i - \{e_i\}} \frac{F_i(T_i, e)}{F'^*_i(T_i, e)} \tag{7}$$

where f^* and F'^* are associated with the new probability distributions that are specified by the *ISDelay* function in the model. Using the above equation, the likelihood ratio associated with the whole sample path $X_{0,n}$ is given by equation (8):

$$L(X_{0,n}) = \prod_{i=0}^{n-1} L(X_{i,i+1}) = \prod_{i=0}^{n-1} \frac{f_i(T_i, e_i)}{f_i^*(T_i, e_i)} \prod_{e \in E_i - \{e_i\}} \frac{F'_i(T_i, e)}{F'^*_i(T_i, e)} \tag{8}$$

Consider V as the interested reward variable, defined by the modeler, which includes *impulse rewards* and *rate reward*. In traditional simulation, the value that is earned by the

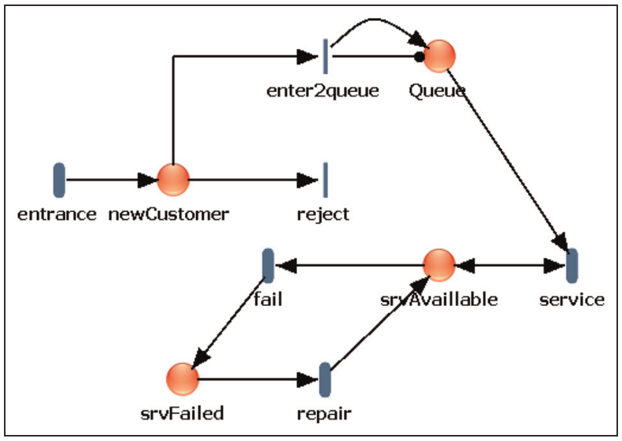


Figure 9. An example of SRN model.

reward variable at the point in time t can be defined by equation (9):

$$V_t = rate(\sigma_t) + \sum_{e \in SE} imp(e) \tag{9}$$

where t and SE denote the state of model and the set of events completed in time t, respectively. In IS, the reward structure is changed to equation (10):

$$V_t = rate(\sigma_t).L(X_{0,N}) + \sum_{e \in SE} imp(e).L(X_{0,N}) \tag{10}$$

Here, N is the number of events that cause the simulation clock to exceed t in the sample path generation and $L(X_{0,N})$ is the likelihood ratio corresponding to this sample path.

As mentioned earlier, the expected value of any performance or dependability measure can be expressed in terms of reward variables. They include the cumulative ones over a time interval, rewards computed in an instance of time, and steady-state rewards. This makes the engine capable of evaluating a wide range of problems with or without rare events.

6. Some examples executed by SimGine

As mentioned earlier, for the simulation of a model using SimGine, it must be translated into the input language of SimGine. In this section, we present two examples: a server modeled by SRNs and rare-event simulation of a SAN model. More complex examples of other formalisms, including SANs, CSANs, MOSEL, and PEPA, in addition to the complete script of the two examples, can be found at the SimGine homepage.[19]

6.1. The SRN model of a server system

In this subsection, the simulation results of a sample SRN model using SimGine are presented. The model shown in Figure 9 is a server system which delivers a service to its customers. The server in the system can deliver service to only one customer at a time and there is a first-in first-out (FIFO) queue in the system, modeled by the place *Queue*, in which the customers can wait. The size of the queue is limited to 50 customers. Two immediate transitions, *enter2queue* (with an inhibitor arc with multiplicity 50 to place *Queue*) and *reject* (with guard *Queue.Mark* > 50) check the queue capacity. The arrival of customers into the system is exponentially distributed with the rate 0.9. If the queue is not empty, the server selects the first customer to be serviced in a time that is exponentially distributed with the rate 1.0 (the execution of the transition *service*). The server may fail and be unable to deliver service to the customers during some periods of time. However, there is a repair person who can repair the server (the transition *repair*) when it has failed. The fail and repair of the server are also exponentially distributed events with the rates 0.1 and 1, respectively.

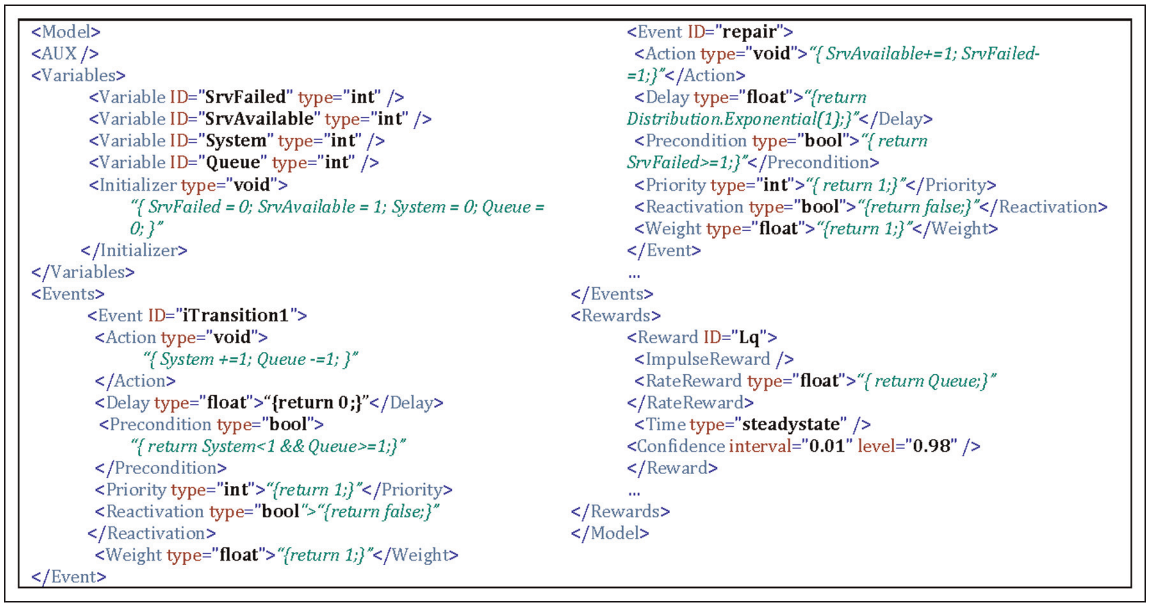
The interesting measures of the system are the availability of the server and the average queue length in steady-state. Thus, two reward variables *QLen* and *avlServer* are defined: the first evaluates the marking of the place *Queue* and the other measures the *srvAvailable*'s marking.

The SRN model can be translated into the SimGine model. Parts of the translated model are shown in Figure 10. As shown in the figure, all places are translated into integer variables where their values represent the marking of places. The transitions are also translated into events considering their guards and the connection and multiplicity of (ordinary and inhibitor) arcs. The results of simulations with the confidence level 0.98% within the confidence interval 0.01 show that the average queue length is 25.54 and the server is available 90.9% of the time.

6.2. Rare-event simulation of a SAN model

SANs[17, 18] are a stochastic generalization of Petri nets. For this case study, consider the SAN model of the machine–repairman system presented in Obal,[32] as depicted in Figure 11.

The system consists of two types of components with different failure rates. The number of each working component in the system is modeled by the marking of the place as *type_1* and *type_2*. Timed activities *fail_1* and *fail_2* model the failure of a working component. The time to failures is exponentially distributed and its rate depends on the number of currently working components (i.e. the firing time is marking dependent to present load balancing), as shown in Table 2. One repairman exists in the system, which waits for the failure of at least two components of the same type, and then begins to repair the whole group. After finishing the repair process, all components of that type will be as good as new ones. This repair policy is



```
<Model>
<AUX />
<Variables>
        <Variable ID="SrvFailed" type="int" />
        <Variable ID="SrvAvailable" type="int" />
        <Variable ID="System" type="int" />
        <Variable ID="Queue" type="int" />
        <Initializer type="void">
              "{ SrvFailed = 0; SrvAvailable = 1; System = 0; Queue =
              0; }"
       </Initializer>
</Variables>
<Events>
        <Event ID="iTransition1">
         <Action type="void">
                "{ System +=1; Queue -=1; }"
         </Action>
         <Delay type="float">"{return 0;}"</Delay>
          <Precondition type="bool">
              "{ return System<1 && Queue>=1;}"
         </Precondition>
         <Priority type="int">"{return 1;}"</Priority>
         <Reactivation type="bool">"{return false;}"
        </Reactivation>
         <Weight type="float">"{return 1;}"</Weight>
</Event>
        <Event ID="repair">
         <Action type="void">"{ SrvAvailable+=1; SrvFailed-
        =1;}"</Action>
         <Delay type="float">"{return
        Distribution.Exponential(1);}"</Delay>
         <Precondition type="bool">"{ return
        SrvFailed>=1;}"</Precondition>
         <Priority type="int">"{ return 1;}"</Priority>
         <Reactivation type="bool">"{return false;}"</Reactivation>
         <Weight type="float">"{return 1;}"</Weight>
        </Event>
        ...
</Events>
<Rewards>
        <Reward ID="Lq">
         <ImpulseReward />
         <RateReward type="float">"{ return Queue;}"
        </RateReward>
         <Time type="steadystate" />
        <Confidence interval="0.01" level="0.98" />
        </Reward>
        ...
</Rewards>
</Model>
```

Figure 10. A part of an SRN model of the server system expressed in SimGine.

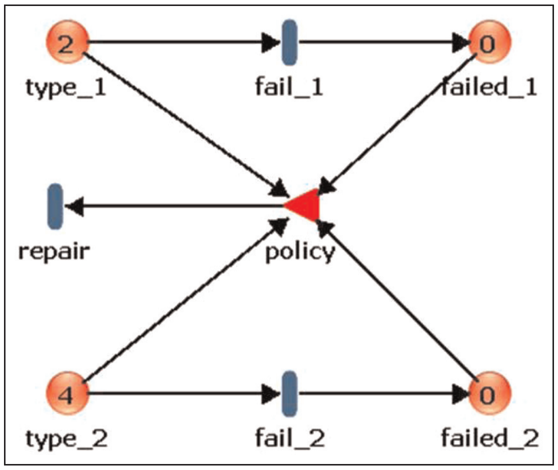


Figure 11. The SAN model of the delayed repair system.

Table 2. Distribution of timed activities.

Activity	Rate (*exponential*)
fail_1	0.005 * *type_1. Mark*
fail_2	0.01 * *type_2. Mark*
Repair	1.0

implemented in the input gate *policy* whose configuration is depicted in Table 3. The *repair* activity is enabled if two or more components of either type have failed, and *type_1* component repair is given preemptive priority over the repair of *type_2* components. Upon completion, the action taken depends on the marking of the place *failed_1*. If two components of *type_1* have failed before the repair completes, then the *type_1* components are repaired; otherwise, *type_2* components are repaired (as implemented in the function of the input gate *policy*). If all components of both types fail, the system fails, and the repair activity halts, causing the failed state to be an absorbing state.

Since we are interested in the unreliability of the modeled system in the interval *[0,100]* and failed state is an absorbing state, we can define the instance of time reward variable *unreliability* with no impulse reward and a rate reward, which determines the failed state in time *100* (i.e. *if (type_1.Mark==0 && type_2.Mark==0) return 1;*). For using the IS technique, the IS delay function should be specified for fail activities. Here, the *ISDelay* is defined based on the one in Obal[32] and is given in Table 4. The results of the IS simulation, as well as the standard simulation, compared with the analytical results are shown in Table 5. Simulations run within the confidence interval 0.1 and 95% of the confidence level on a 2.0GHz Intel Core2Duo laptop with 1.5GB RAM and Microsoft Windows XP.

7. Comparison

In this section, we briefly compare SimGine with other multi-formalism simulation tools. Among the available tools, we have selected those which are more related to SimGine: Möbius, MOSEL, SMART, SHARPE, and OsMoSys. For comparison purposes, we consider the capability of extending the tool, supporting step-by-step

Table 3. Configuration of the input gate *policy*.

Predicate	Function
return ((type_1.Mark > 0 || type_2.Mark > 0) && (failed_1.Mark == 2 || failed_2.Mark == 1));	If(failed_1.Mark==2){ failed_1.Mark=0; type_1.Mark=2;} else{ failed_2.Mark=0; type_2.Mark=4; }

Table 4. *ISDelay* function for *fail* activity in the model of Figure 11.

Activity	Function
fail_1	if((failed_1.Mark >= 2 && failed_2.Mark >= 2)||(failed_1.Mark >= 1 && failed_2.Mark >= 3)) return Distribution.Exponential(0.005*type_1.Mark/((.005*type_1.Mark + 0.01*type_2.Mark))); else if((failed_1.Mark >= 2 && failed_2.Mark >= 1)|| (failed_1.Mark >= 1 && failed_2.Mark >= 2)) return Distribution. Exponential (0.09*0.005*type_1.Mark/ ((.005*type_1.Mark + 0.01*type_2.Mark)*(1-0.09)));
fail_2	if((failed_1.Mark >= 2 && failed_2.Mark >= 2)||(failed_1.Mark >= 1 && failed_2.Mark >= 3)) return Distribution.Exponential(0.01*type_2.Mark/((.005*type_1.Mark + 0.01*type_2.Mark))); else if((failed_1.Mark >= 2 && failed_2.Mark >= 1)|| (failed_1.Mark >= 1 && failed_2.Mark >= 2)) return Distribution.Exponential(0.09*0.01*type_2.Mark/ ((.005*type_1.Mark + 0.01*type_2.Mark)*(1-0.09)));

Table 5. Simulation results for the model of Figure 11.

	Analytical	Traditional	IS
Result	$1.91 * 10^{-6}$	$1.71 * 10^{-6}$	$1.94 * 10^{-6}$
Relative error	–	10.4%	1.5%
Time	–	2956	16
Samples number	–	224,632,000	110,000

simulation, the capability of analytic solution of models, and supporting models' interactions. By 'extensibility' we mean that the tool can be extended to evaluate new kinds of models instead of having a predefined set of supported models. Step-by-step simulation is a feature which enables the tool to support graphical representation of a simulation during the simulation progress (e.g. token play in Petri nets). Here we just consider simulation tools, but some of these tools are also able to evaluate models analytically.

Table 6 presents the comparison of the multi-formalism simulation tools mentioned above. The mentioned tools were introduced in Section 2. Below, we briefly compare some of their features with SimGine:

- Möbius is an extensible multi-formalism modeling tool which provides this extensibility through the use of an abstract functional interface (AFI) that uses abstract classes to provide a formalism-independent interface. The complexity of the framework and poor documentation make the implementation of a new formalism without a direct Möbius developers team's assistance impossible. Instead, SimGine has an XML-based language which makes it easier to evaluate new formalisms. Möbius does not support step-by-step simulation and rare-event simulation, and cannot be used as a library. However, it can analytically solve (Markovian) models and is able to evaluate composed models within different formalisms.
- MOSEL is a textual language and environment. For evaluation, the model is specified in the language, and then the environment calls an external tool after having translated the MOSEL description into the

Table 6. Comparison of the selected multi-formalism simulation tools (for stars, see the text).

	Möbius	MOSEL	SMART	SHARPE	OsMoSys	SimGine
Extensible	✓	–	*	–	✓	✓
Step-by-Step Sim.	–	–	–	–	–	✓
Rare-Event Sim.	–	–	–	–	–	✓
Analytical Solutions	✓	*	✓	✓	*	–
Library	–	–	–	–	–	✓
Models' Interaction	✓	–	✓	✓	*	–

respective tool's description format. Therefore, it does not have any engine for model evaluation. The language also does not support rare-event simulation in its language constructs.

- SMART permits extension of the tool by integrating new solution algorithms. It uses a declarative language to specify stochastic Petri nets and queuing networks. Models can interact by exchanging their results (which is a weak kind of model interaction compared to supporting the composed/hierarchical models).
- Although SHARPE is not a simulation tool, it was an important and early effort in multi-formalism modeling tools in which models can be expressed in various formalisms. It can analytically evaluate models for performance and dependability analysis. SHARPE is not an extensible tool.
- The OsMoSys approach to multi-formalism modeling is meta-modeling, to define and integrate different formalisms. The approach is different from SimGine and the aforementioned tools, and basically is an architecture for dealing with multi-solution when analyzing multi-formalism models. The methodology is based on the orchestration of different solvers which enables the interoperability among solution methods and solvers.

SimGine is just a simulation engine and cannot solve models analytically. There is another ongoing project in PDELab, named SolvGine, with the aim of solving SDES models (with the same input language as SimGine). It is also notable that, unlike some of the mentioned tools, SimGine is not a modeling tool but a simulation engine, and so, it does not provide a GUI to build (high-level) models. As stated earlier, one can use SimGine in one's own modeling tool, or use PDETool (which aims to provide an interface for constructing some high-level models and translating them into the SimGine input language).

Regarding hierarchical modeling, currently SimGine does not support hierarchical models directly and the modeler has to use a flat model. But the modeler can still create a composed high-level model and translate the flattened model into a SimGine model.

8. Conclusions

In this paper, we introduced SimGine, a simulation engine based on a unified description, named SDES, for the simulation of stochastic discrete-event systems. Using SimGine, one can simulate models of various formalisms by translating them into the input language of the engine. For each kind of model, a translator should be developed. One of the important features of SimGine is its capability for rare-event simulation using the IS technique. As far as we know, this is the first multi-formalism simulation tool capable of simulating rare events.

Currently, we are working on the engine to support model composition. This capability will enable the engine to have hierarchical models in which each part of the model can be constructed using different formalisms. Another feature under development is an ongoing project called SolvGine for the analytic solution of Markovian models. Another future work is enabling the engine to perform parallel simulation. This is important because of emerging new multi-core architectures that will speed up the simulation progress.

References

1. Zimmermann A. *Stochastic discrete event systems: Modeling, evaluation, applications*. Berlin: Springer-Verlag, 2007.
2. Zimmermann A, Knoke M, Huck A, et al. Towards version 4.0 of TimeNET. In: *Proceedings of the 13th GI/ITG conference on measuring, modelling and evaluation of computer and communication systems (MMB'06)*, 2006, pp. 1–4.
3. Sanders WH, Obal WD, Qureshi MA, et al. The UltraSAN modeling environment. *Perform Eval* 1995; 24(1–2): 89–115.
4. Vittorini V, Iacono M, Mazzocca N, et al. The OsMoSys approach to multi-formalism modeling of systems. *Softw Syst Model* 2004; 3(1): 68–81.
5. Hirel C, Sahner R, Zang X, et al. Reliability and performability modeling using SHARPE 2000. In: *Proceedings of the 11th international conference on computer performance evaluation: Modelling techniques and tools*, 2000, pp. 345–349.
6. Ciardo G and Miner AS. SMART: Simulation and Markovian analyzer for reliability and timing. In: *Proceedings of the 1996 IEEE international computer performance and dependability symposium*, 1996, p. 60.
7. Deavours DD, et al. *Formal specification of the Möbius modeling framework.* PhD Thesis, University of Illinois at Urbana-Champaign, IL, 2001.
8. Williamson AL. *Discrete event simulation in the Möbius modeling framework*. Master's Thesis, University of Arizona, AZ, 1998.
9. Bolch G and Herold H. MOSEL – MOdeling Specification and Evaluation Language. In: *Proceedings of the 2001 Aachen international multiconference on measurement, modelling and evaluation of computer and communication systems*, Aachen, Germany, 1995.
10. Wiichner P, De Meer H, Barner J, et al. MOSEL-2 – A compact but versatile model description language and its evaluation environment. In: *Proceedings of the MMBnet workshop*, 2005, pp. 51–59.
11. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. San Diego, CA: Academic Press, 2000.

12. Sarjoughian HS and Elamvazhuthi V. CoSMoS: A visual environment for component-based modeling, experimental design, and simulation. In: *Proceedings of the 2nd international conference on simulation tools and techniques*, 2009, p. 59.
13. Kim S, Sarjoughian HS and Elamvazhuthi V. DEVS-Suite: A component-based simulation tool for rapid experimentation and evaluation. In: *Spring simulation multi-conference*, San Diego, CA, 2009.
14. Castro R, Kofman E and Wainer G. A formal framework for stochastic discrete event system specification modeling and simulation. *Simul* 2010; 86(10): 587–611.
15. Molloy MK. Performance analysis using stochastic Petri nets. *IEEE T Comput* 1982; C-31(9): 913–917.
16. Marsan MA, Balbo G and Conte G. *Performance models of multiprocessor systems*. Cambridge, MA: The MIT Press, 1986.
17. Movaghar A and Meyer JF. Performability modeling with stochastic activity networks. In: *Proceedings of the real-time systems symposium*, Austin, TX, 1985, pp. 215–224.
18. Meyer JF, Movaghar A and Sanders WH. Stochastic activity networks: Structure, behavior, and application. In: *Proceedings of the international workshop on timed petri nets*, 1985, pp. 106–115.
19. PDELab. SimGine homepage, http://pdel.iust.ac.ir/projects/SimGine.html.
20. Sanders WH and Meyer JF. A unified approach for specifying measures of performance, dependability, and performability. *Depend Comput Fault-Tolerant Syst* 1991; 4: 215–237.
21. Khalili A, Jalaly Bidgoly A and Abdollahi Azgomi M. PDETool: A multi-formalism modeling tool for discrete-event systems based on SDES description. In: *Proceedings of the 30th international conference on application and theory of petri nets and other models of concurrency (ICATPN'09)*, Paris, France, 22–26 June 2009, pp. 343–352.
22. Khalili A and Abdollahi Azgomi M. RayLang: A modeling language for performance evaluation of stochastic discrete-event systems. In: *Proceedings of the 2009 international middle eastern simulation multiconference (MESM'09)*, 2009, pp. 119–130.
23. Juneja S and Shahabuddin P. Rare event simulation techniques: An introduction and recent advances. In: Henderson SG and Nelson BL (eds) *Simulation* (*Handbooks in Operations Research and Management Science*, 13). Amsterdam: Elsevier, 2006, pp. 291–350.
24. Shahabuddin P. Importance sampling for the simulation of highly reliable Markovian systems. *Manage Sci* 1994; 40(3): 333–352.
25. Glynn PW and Iglehart DL. Importance sampling for stochastic simulations. *Manage Sci* 1989; 35(11): 1367–1392.
26. Glasserman P, Heidelberger P, Shahabuddin P, et al. Splitting for rare event simulation: Analysis of simple cases. In: *Proceedings of the 28th conference on winter simulation*, 1996, pp. 302–308.
27. L'Ecuyer P, Demers V and Tuffin B. Splitting for rare-event simulation. In: *Proceedings of the 2006 winter simulation conference (WSC'06)*, 2006, pp. 137–148.
28. Kahn H and Harris TE. Estimation of particle transmission by random sampling. *Natl Bureau Stand Appl Math Ser* 1951; 12: 27–30.
29. Villén-Altamirano M, Villén-Altamirano J and de Madrid UP. RESTART: A method for accelerating rare-event simulations. In: *Proceedings of the 13th international teletraffic congress*, Copenhagen, Denmark, 19–26 June 1991, pp. 71–76.
30. Villén-Altamirano M and Villén-Altamirano J. RESTART: A straightforward method for fast simulation of rare events. In: *Proceedings of the 1994 winter simulation conference*, 1994, pp. 282–289.
31. Villén-Altamirano M and Villén-Altamirano J. On the efficiency of RESTART for multidimensional state systems. *ACM Trans Model Comput Simul (TOMACS)* 2006; 16(3): 251–279.
32. Obal WD II. *Importance sampling simulation of SPN-based reward models*. Master's Thesis, University of Arizona, AZ, 1993.
33. Nicola VF, Nakayama MK, Heidelberger P, et al. Fast simulation of highly dependable systems with general failure and repair processes. *IEEE T Comput* 1993; 42(12): 1440–1452.

Author biographies

Ali Khalili received his BSc degree in software engineering from Ferdowsi University of Mashhad, Iran, 2005. In 2009, he received an MSc degree in computer engineering (software) from Iran University of Science and Technology (IUST). Currently, he is a PhD student at the Italian Institute of Technology (IIT), Genova, Italy. His research topics are modelling and simulation, formal verification, and model-based testing.

Mohammad Abdollahi Azgomi received his BS, MS, and PhD in computer engineering (software) in 1991, 1996, and 2005, respectively, from Sharif University of Technology, Tehran, Iran. His research interests include dependable systems and software, performance and dependability modelling, and network security. He has published several papers in international journals and conferences. Dr. Abdollahi Azgomi is currently a faculty member at the School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

Amir Jalaly Bidgoly received his BSc degree in computer engineering (software) from the Department of Computer Engineering, Kashan University, in 2005. He received an MSc degree in computer engineering (software) from Iran University of Science and Technology (IUST) in 2009. He is currently a PhD student at the Department of Computer Engineering, Isfahan University, Isfahan, Iran. His research interests include modelling and simulation, rare-event simulation, and fault-tolerant computing. He has published several papers at national and international conferences.

Appendix: SDES description

A discrete-event system is a system that is in a state during some time interval, after which an atomic event might happen that changes the state of the system immediately.[1] Several stochastic discrete-event models have been proposed, all of which share some common characteristics, and many of the algorithms and methods that have been developed for one model are applicable for

many of them. SDES[1] is a unified description for stochastic discrete-event systems. Popular model classes like automata, queuing networks, and Petri nets of different kinds with stochastic extensions are subclasses of stochastic discrete-event systems and can be translated into the introduced SDES description. In this section, we review the definition of SDES. For more information, please see Zimmermann.[1]
A stochastic discrete-event system SDES is a tuple

$$SDES = (SV^*, A^*, S^*, RV^*)$$

which describes the finite set of state-variables SV^* and actions A^* together with the sort function S^* and the reward variables RV^* correspond to the quantitative evaluation of the model. SV^* is the finite set of n state-variables, $SV^* = sv_1, \ldots, sv_n$, which is used to capture the states of the model. S^* is a function that associates an individual sort to each state-variable of SV^* and action variable of $Vars^*$ (which will be defined later) in a model; in other words,

$$S^* : (SV^* \bigcup Vars^*) \rightarrow \delta^*$$

where δ^* denotes the set of all possible sorts. Each state-variable $sv \in SV^*$ has the attribute $(Cond^*, Val_0^*)$. Sometimes there are cases in which not all values that belong to a particular sort of a state-variable are actually allowed. The state condition $Cond^*$ is a boolean function which returns for a state-variable in a specific model state whether or not it is allowed such that

$$Cond^* : SV^* \times \Sigma \rightarrow B$$

where Σ is the set of all theoretically possible states of a certain SDES and is defined as

$$\sum = \prod_{sv \in SV^*} S^*(sv)$$

Val_0^* is a function that specifies the initial value of each state-variable, which is necessary as a starting point for an evaluation of the model behavior that satisfies the following requirement:

$$\forall sv \in SV^* : Val_0^*(sv) \in S^*(sv)$$

A^*, which denotes the set of actions of an SDES model, describes possible state changes of the modeled system. An action $a \in A^*$ may be composed of several internal actions with different attributes in some SDES model classes, or may contain individual variants or modes in other examples. To capture this, the action variables $Vars^*$ define a model-dependent set of variables $Vars^*(a)$ of an action a with individual sorts. One setting of values for these variables corresponds to an action mode *mode*. This is, for example, equivalent to a binding in a colored Petri net. Many attributes depend on an action a together with one of its corresponding modes, $mode \in Modes^*$. Any possible pair of an action and an action mode is called an action variant, written as v. The set of all possible action variants AV is defined as

$$AV = \{(a, mode) | a \in A^*, mode \in Modes^*(a)\}$$

Each action a, $a \in A^*$, is

$$a = (Pri^*, Deg^*, Var^*, Ena^*, Delay^*, Weight^*, Exec^*)$$

where:

- Pri^* associates a global priority to every action. The priority is used to decide which action is executed first if there are several activities that are scheduled to finish at the same point in time.

$$Pri^* : A \rightarrow N$$

- The enabling degree Deg^* of an action specifies the number of activities of it that are permitted to run concurrently in any state. This is for instance used to capture the difference between infinite-server and single-server semantics.

$$Deg^* : A \rightarrow \{N \bigcup \infty\}$$

- Action variants may only start and proceed over their delay under certain conditions until execution. If these conditions hold in a state, we say the action variant is enabled in it. The value of the boolean enabling function Ena^* of an action variant returns for a model state whether or not it is enabled.

$$Ena^* : AV \times \Sigma \rightarrow B$$

- $Delay^*$ describes the time that must elapse while an action variant is enabled in an activity until it finishes. This time is in most cases not a fixed number, but a random variable with positive real values.

$$Delay^* : AV \rightarrow F^+$$

- The weight $Weight^*$ of an action variant is a positive real number that defines the probability of selecting it for execution in relation to other weights. This applies only to cases in which activities with equal priorities are scheduled for execution at the same instant of time (e.g. firing weights of immediate transitions in Petri nets).

$$Weight^* : AV \rightarrow R^+$$

- *Exec** defines the state change that happens as a result of an action variant execution (i.e. the finishing and execution of the activity) and is called the execution function. As actions change the state, *Exec** is a function that associates a destination state to a source state for each action variant. This function does not need to be defined or have a useful value for pairs containing a variant that is not enabled in the respective state.

$$Exec^{*} : AV \times \Sigma \rightarrow \Sigma$$