Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

RubySimF: A Discrete Event Simulation Framework in Ruby

Prof. Eliezer Correa
Prof. Sergio Rivas

RT 2010-01

Centro de Investigación de Operaciones y Modelos Matemáticos (CIOMMA)
Caracas, Enero 2010.

RubySimF: A Discrete Event Simulation Framework in *Ruby*

Prof. Eliezer Correa (elcorrea@gmail.com)
Prof. Sergio Rivas (sergiorivas@gmail.com)
Universidad Central de Venezuela
Ciudad Universitaria, Los Chaguaramos
Caracas, Venezuela.

ABSTRACT

RubySimF is an open source simulation framework based on the *Ruby* language for customizing simulations using a process interaction approach. In this work we discuss the benefits of using a paradigm with dynamic languages such as our *Ruby* based framework, allowing for changes at execution time, block handling, and compatibility with various general purpose libraries. For handling simulation time between discrete events we propose implicit interoperable execution threads, furthermore we use classes for encapsulating behavior of the complex components in a system and also tools for input modeling and output analysis. *RubySimF* is suited for general-purpose simulation of complex systems and was designed with emphasis on usability, however it can be extended or customized with little coding effort. This work is in progress and we are working on new classes to support performance evaluation of computational systems.

General Terms

Simulation Frameworks

Keywords

Discrete Event Simulation, *Ruby*, Simulation Handling Time

1. INTRODUCTION

Simulation is a tool used to evaluate different scenarios of a situation and their possible outcome in a practical and economic manner. In the simulations, time management can be complex or inefficient. This work proposes a framework that allows writing simulations with little codification effort and easy usage. This framework *RubySimF* is based on the *Ruby* programming language, which is increasingly used for computational developments in different contexts.

The advantages of a framework based on the *Ruby* programming language, are to use the dynamic features of the language, allowing a legible and cleaner code. The proposed framework *RubySimF* is a tool for writing simulation code in short time as high-level simulation packages and also allows writing details of the simulation as low-level simulation libraries. *RubySimF* uses the best of *Ruby*'s features at each level of abstraction.

2. DISCRETE EVENT SIMULATION

Given a situation where there is the need to advance the time as a variable during simulation. There are different strategies for advancing the clock (to emulate the passage of time), and simulate the dynamics of the entities in the system along the advance of time. Discrete event simulation is an strategy to advance the simulation clock (time handling) of an efficient manner.

2.1 Time handling

A strategy based on reality is to advance the simulation clock according to the real-time clock. Although this strategy may be accelerated and run the simulation proportionally in time, it represents an expense in unnecessary resources and therefore it is a very inefficient strategy.

Another strategy is to advance the simulation clock in fixed time intervals and executing the actions pertaining to activation in a particular interval. This simple strategy can be inadequate, as by advancing the simulation clock there may not exist any action that changes the state of the system at that time and therefore the simulation would use many resources such as memory or processing units, that are not needed.

Besides the before mentioned strategies, a more efficient strategy is advancing the simulation clock at exactly those moments of time where the system is scheduled to change. The discrete event-based simulation can advance over time in steps, avoiding delays for unnecessary executions.

The discrete event simulation is based on modeling the system changes at separate points in time [1]. System state changes at certain points of time, and the logic of the simulation only concentrates on observing those points representing the situation to model.

Simulation theory proposes a general control algorithm to handle the course of the simulation clock. Events are defined as instants in time where the system changes; this algorithm processes the events in chronological order and manages a list of events waiting to be executed. Simulation algorithm advances the simulation clock in a coherent form with the system to simulate [5].

2.2 Process Interaction Approach

The simulation based on event scheduling requires the user to handle the concepts of events and time scheduling very well. In fact, the developer must be aware of each type of event that exists in the situation to be represented.

The programmer must also know which events generate other events; this level of detail can be very complex in a large system so high-level alternatives are needed for creating a simulation program [4].

Today, developers are more focused at the system and the interaction between entities, rather that at the events. The current programming languages and the object-oriented approach tend to solve computational problems in order to simulate a level of abstraction where the programmer defines a high-level entities and their interaction [7].

Process-based simulation or also the process interaction approach hides the low-level events in discrete event simulations to focus on the processes and their interaction [2].

Aspects to take into account with this approach are:

1. Simulation consists of the interaction of entities in the system.
2. In the simulation should identify the processes and activities that define the system behavior.
3. There are not explicitly events in the code.

A process represents a set of activities, however, in discrete event simulation a process is a sequential list of events and time delays that include demands on system's resources [3] (see Figure 1).

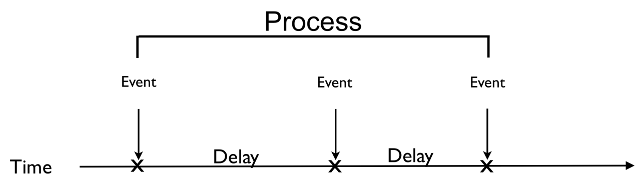


Figure 1: A Process example

Delays represent the time duration of an activity or to be waiting for a condition.

A language, library, or framework based on interaction of processes should allow a way to specify which are the processes, and also the activities that define it.

This approach hides the handling of events to internal layers of development, and under this approach it should be possible to specify processes and implicit events should be created automatically.

3. *RUBY* PROGRAMMING LANGUAGE

The *Ruby* programming language is a full object-oriented, general-purpose and open source licensed programming language focused on simplicity that was initially created in the 90s.

Ruby has lately gained much popularity by taking advantages of its dynamism, and many developers prefer *Ruby* language as it allows focusing their code on human behavior rather than machine's. This approach allows writing cleaner, more elegant and natural code.

In relation to the implementation and extensibility *Ruby* is multiplatform, and can run in interpreted or compiled; additionally, there are now versions of *Ruby* in other languages such as *JRuby* which is an implementation of *Ruby* written in the Java, therefore *Ruby* programs can run in a Java environment and use Java libraries from *Ruby* programs. Furthermore, *Ruby* has tools for integration and extension of other languages like *C* and *Python*.

Ruby is a dynamic language with several mechanisms for changing the behavior of an object at runtime.

Among the features of dynamism as an advantage for simulation we can highlight the following: It is reflexive, presence of the block concept allowing a method to change behavior, handling of messages instead of calls, the *method_missing* method, code evaluation, open classes, allows mixing of code, and has hooks methods. All these mechanisms allow changing: a method, the structure of a class, or the inheritance hierarchy at runtime, among others. The described features are helpful to simulate structural changes and represent evolution of systems.

Dynamism may seem dangerous but today is used as an advantage to generate code at runtime and write less code.

4. *RUBYSIMF*

RubySimF (*Ruby Simulation Framework*) is a framework written in the programming language *Ruby* that takes advantage of the dynamic features of *Ruby* allowing the coding of simulation in an elegant, clean and simple way. *RubySimF* use these features to help to developers to make simulations based on the interaction process approach and provides an *API* for extending behavior in a simulation, e.g. random values generation, input modeling, output analysis among others.

Subsequently some of the features of *RubySimF* are shown, highlighting some of the advantages of its use as a simulation framework.

4.1 Asynchronous calls & implicit processes

RubySimF handles advancing the simulation clock internally by a discrete event-based simulation. The level of abstraction in the framework is the process interaction approach.

In contrast to other tools, in *RubySimF* processes are not explicit, neither the specification of a process is intrusive in the structure of an application domain; some simulation languages force inheritance of certain classes to work with the process interaction approach.

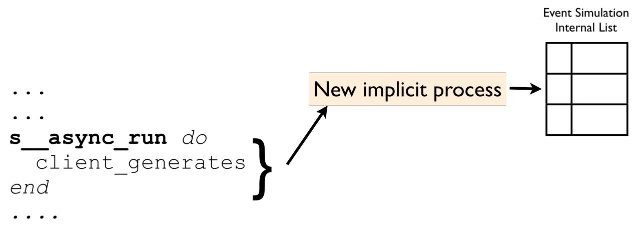
RubySimF bases simulated process activations on an instruction (*async_run*) that receives a block of code. When *async_run* is executed then a new process is created with the code specified within the block, and simultaneously a new event is scheduled in the list of events pending to activate this process later.

RubySimF takes advantage of *Ruby* to write asynchronous calls in a single block and at runtime to create and keep control of the blocks giving a simple and elegant form for invoking an implicit process.

This problem of multiple instances (*processes*) running simultaneously in a simulation is solved by asynchronous calls and the framework has control of the simulation clock.

For each asynchronous call the framework creates a new internal process that inherits from a thread *Ruby* class and manages the execution control of that thread, as well as their status changes.

For example the process is blocked or unblocked as appropriate, and it is all done internally. See Figure 2.



```
...
...
s__async_run do
  client_generates
end
....
```

Figure 2: Asynchrony Call

4.2 Entities Prebuilt

For less effort in the coding time, the framework provides pre-built classes that model the behavior of complex entities in a simulation domain.

Resources are entities that are used as servers with queues for a waiting line situation. Resources get a predefined capacity at the moment of initialization and can later be changed. The resource blocks the process that tries to use it when the capacity is full, inserts the process in a waiting queue; when the resource it is released the next process if any, of the queue is unblocked.

Resources are entities that model a ticket window, or simply a service with a waiting queue.

Currently we are researching which should be the general complex entities to be provided by the framework and how to add modules to extend the framework with a set of entities solving complex problems in a specific domain.

4.3 Random Generates

The *Ruby* language does not include an official library for generating values of random variables for known probability distributions. The framework also provides methods for the generation of random values with a *binomial, negative binomial, Bernoulli, geometric, triangular, discrete-uniform, continuous-uniform, hyper-geometric, normal, exponential, Poisson, log-normal* probability distributions among others.

To generate the values we used convolution, inversed cumulated density function, direct and indirect methods; all these methods are based on a pseudorandom [6] number generator provided by the framework because the random method of the *Ruby* language is not enough tested.

Using these functions the programmer also prevents redoing the necessary code needed to generate the random numbers involved in a simulation.

4.4 Statistic Collectors

For output analysis it is necessary to collect statistics. *RubySimF* provides the monitors, which are entities responsible for keeping observations at various points in time during an execution, at the discretion of the programmer.

At the end of an execution, after gathering or monitoring values it can be inquired about the observations statistical average, standard deviation, sum of observations, time-weighted average, minimum, maximum.

4.5 Templates & Patterns

In order to create simulations programs, *RubySimF* produces self-generated programs by using the "*generate*" command, which provide skeletons for simple models of simulation.

The generated templates are documented and can be modified, as the intention is to provide a base for larger programs and complex simulation models.

For the construction of this framework several design pattern were used such as *Singleton, Strategy, Template Method*, and *Decorator* among others. Design patterns allow for an elegant, known and reusable solution for creating frameworks.

4.6 Input & Output in Simulation

For the analysis of situations, the user usually requires a previous work consisting of structuring, classification and preparation of data to support the model, likewise, there is a subsequent work in a simulation in analyzing the results to interpret the behavior of the system under a given situation. For both cases, the framework provides useful tools.

4.6.1 Input Modeling

For input modeling *RubySimF* provides templates for input analyzers using generators. Furthermore, test algorithms can be applied to recognize known distributions in the input. This feature is in progress.

4.6.2 Output Analysis

Regarding output analysis, *RubySimF* integrates tools such as plotters with statistical collectors allowing create histograms, scattering plots, among others. There are also available methods for estimating confidence intervals as an analytical tool with the simulation results. This feature is in progress.

5. CONCLUSION

RubySimF is a simulation tool for writing simulation code with a simple to use and elegant form. It takes advantage of the dynamic features of *Ruby* allowing writing simulations with less code. Specifying implicitly the processes allows writing a non-intrusive code in the problem domain and enhances a clean design and codification. Use of threads to model a process gives control of block or unblocks an activity with events that activate and inactivate the process.

RubySimF is still under construction, and in its current state provides pre-built entities for extensions. These entities are under study seeking to provide generally used entities for simulations environments. The extensions should add new pre-build entities specified for a particular context. The framework also provides tools for model input and output analysis, with the goal of being an integral tool for simulation. In addition, *RubySimF* provides a library to perform the generation of random values with common probability distributions.

6. REFERENCES

[1] Law A. *Simulation Modeling & Analysis*. Mc Graw Hill. 2007.

[2] Banks J. *Discrete Event System Simulation*. Prentice Hall. 2005.

[3] Tyszer, J. *Object-oriented computer simulation of discrete-event systems*. Kluwer Academic Publishers. 1999.

[4] Pooch U, Wall J. *Discrete event simulation: a practical approach*. CRC Press Inc. 1993.

[5] Rivas S. *Estrategias del manejo del tiempo en simulaciones dinámicas.* Notas de Docencia ND 2010-01. Escuela de Computación U.C.V. 2009.

[6] Dagpunar J. *Principles of Random Variate Generation*. Oxford University Press 1988.

[7] Law A. *An introduction to Simulation using SIMSCRIPT II.5*. C.A.C.I. 1984.