Open modeling and simulation framework for evolutive analysis

Eng. De Giusti Marisa Raquel[1], Lic. Lira Ariel Jorge[2], Lic. Villarreal, Gonzalo Luján[3].
[1]Comisión de Investigaciones Científicas de la Pcia. De Buenos Aires (CICPBA)
[1,2,3] Proyecto de Enlace de Bibliotecas (PrEBi)- Universidad Nacional de La Plata, Argentina
3 Consejo Nacional de Investigaciones Técnicas y Científicas (CONICET)

Index Terms: Discrete event simulation, storage of runs, statistical and evolutive analysis, simulation teaching.

Simulation is the process of executing a model, that is a representation of a system with enough detail to describe it but not too excessive. This model has a set of entities**,** an internal state, a set of input variable**s** that can be controlled and others that cannot, a list of processes that bind these input variables with the entities and one or more output values, which result from the execution of events.

Running a model is totally useless if it can not be analyzed, which means to study all interactions among input variables, model entities and their weight in the values of the output variables. In this work we consider Discrete Event Simulation, which means that the status of the system variables being simulated change in a countable set of instants, finite or countable infinite.

Existing GPSS implementations and IDE's provide a wide range of tools for analysis of the results, for generation and execution of experiments and to perform complex analysis (such as Analysis of Variance, screening and so on). This is usually enough for most common analysis, but more detailed information and much more specific analysis are often required.

In addition, teaching this kind of simulation languages is always a challenge, since the way it executes the models and the abstraction level that their entities achieve is totally different compared to general purpose programming languages, well known by most students of the area. And this is usually hard for students to understand how everything works underground.

We have developed an open source simulation framework that implements a subset of entities of GPSS language, which can be used for students to improve the understanding of these entities. This tool has also the ability to store all entities of simulations in every single simulation time, which is very useful for debugging simulations, but also for having a detailed history of all entities (permanents and temporary) in the simulations, knowing exactly how they have behaved in every simulation time.

In this paper we provide an overview of this development, making special stress on the simulation model design and on the persistence of simulation entities, which are the basis that students and researchers of the area need in order to extend the model, adapt it to their needs or add all analysis tools to the framework.

[1] marisa.degiusti@sedici.unlp.edu.ar
[2] alira@sedici.unlp.edu.ar
[3] gonetil@sedici.unlp.edu.ar

1. Introduction.

The purpose of any simulation is not to run (execute) a system in a computer but to gather as much information as possible for making all the analysis we can imagine and then get to know deeply how the real system behaves, detect problems and impove it, making it more efficient, faster and reliable. In order to achieve this goals, the analyst must at least:

- study the real system to simulate
- extract both system variables (controllable and un-controllable) and internal process from the system
- dump this information into a simulation program, reflecting data and behavior in a way the computer understands, creating an abstract – mathematical - model of the real system
- run somehow the model
- extract as much information as possible from the run, adjust the model and keep running it

Block programming languages offer an important abstraction level that allows the programmer to map objects from the real system to entities of the simulation in an almost transparent way, providing with each object a set of functions and facilities, which lets the programmer focus on designing the model and forget about programing of implementation details. This is particularly useful when looking for quick solutions, since time savings achieved can be very significant and this the required cost for modeling, simulating and experimenting with systems goes considerably down.

Simulation languages usually collect lots of data along simulation runs, and most of them offer tools for displaying this information, either using simple text reports and probably statistics graphs, tables and functions. GPSS programming language implementations, which this work are based on, are not an exception.

GPSS common model is thought to store and calculate statistics while the model are being run, and allow the analyst to access them when the simulation has finished. This way, it is very easy to know how most permanent entities (facilities, storages, logics switch, and so on) have behaved according to the model definition. This is very useful to detect bottlenecks in the system, useless objects or probable failure points, among many other uses.

In addition to all the information given once the simulation has ended, it is very important to know how our mathematical model has indeed reached this final state and why it has happened this way. This kind of analysis is far too complicated in discrete models executed in GPSS-like languages, specially average or large models with many entities, transactions and processes inside. Thousands of transactions might been created, moved and destroyed, and it is really hard to track them back individually; the main problem here is that the analyst only gets the information once everything has finished. Few GPSS implementations permits to use some sort of breakpoints (STOP/STEP commands), but it is not really handy to run large models step by step.

In this work we present an incipient open simulation framework that allows to store in a database all simulation objects in order to know how they have evolved along the simulation time and what has happened in each *clock* instant. This allows the analyst, for example, to pick any entity and know all changes it has suffered in all simulation times, which entities it has interacted with and what was its time life inside the model. In addition, they may select a range of time and see what entities existed at that moment. Or we can go even further, and consider running and persisting many simulations with different parameters and then compare information from different simulations runs.

As we have mentioned, it is an open and incipient framework. It is open because it has been created according to GPL license, and incipient because it implements - up to now - all basic GPSS model and entities which permits to run simple simulations but is being extended in order to incorporate other entities, completing the whole model or even enhancing it with new entities.

2. The model.

For the development of this framework a subset of GPSS entities have been selected, which allow us to execute simple simulations but that require the execution model to be complete enough. Among these entities we have Transaction, Facility and, clearly, the Simulation entity. The framework is written in Java language, using all Object Oriented power and encouraging programers to improve it since the language is well kown, powerful, portable and very easy to learn.

Before going on with the description of the model, we should make clear the following concept: A discrete simulation starts in a *virtual* zero time, which controlled by an internal system clock called the *Simulation Clock*. Once the simulation has started, it moves forward this internal clock in discrete, ordered and finite *time instants*, until the ending condition is achieved or something wrong has happened; whether the simulation has ended for any unexpected situation or everything went fine, we have a final clock, set to the value the clock had when the simulation had ended.

The importance of the concept lies in the fact that everything that happens in any simulation, happens with the clock set to a determined value. And here is the essence of this framework: given a time instant or *moment* in a simulation, we want to know what has happened, what was the simulation like before and how has it moved forward.

In any simulation, many entities interact among each other in very diferent ways, depending the class of entity (permanent, temporary), the moment they are creating (at the begining of the simulation, when it is first refered to, when the system requires it) and the amount of entities of the same class exist (only one, like Simulation or Clock entities, a fixed amount, like Facilities, or a dynamic amount, such as transactions). All entities form an horizontal composition cyclic graph, being the entity Simulation as the root of this graph, from which all other nodes can be visited (Fig. 1), making it a fully connected graph.

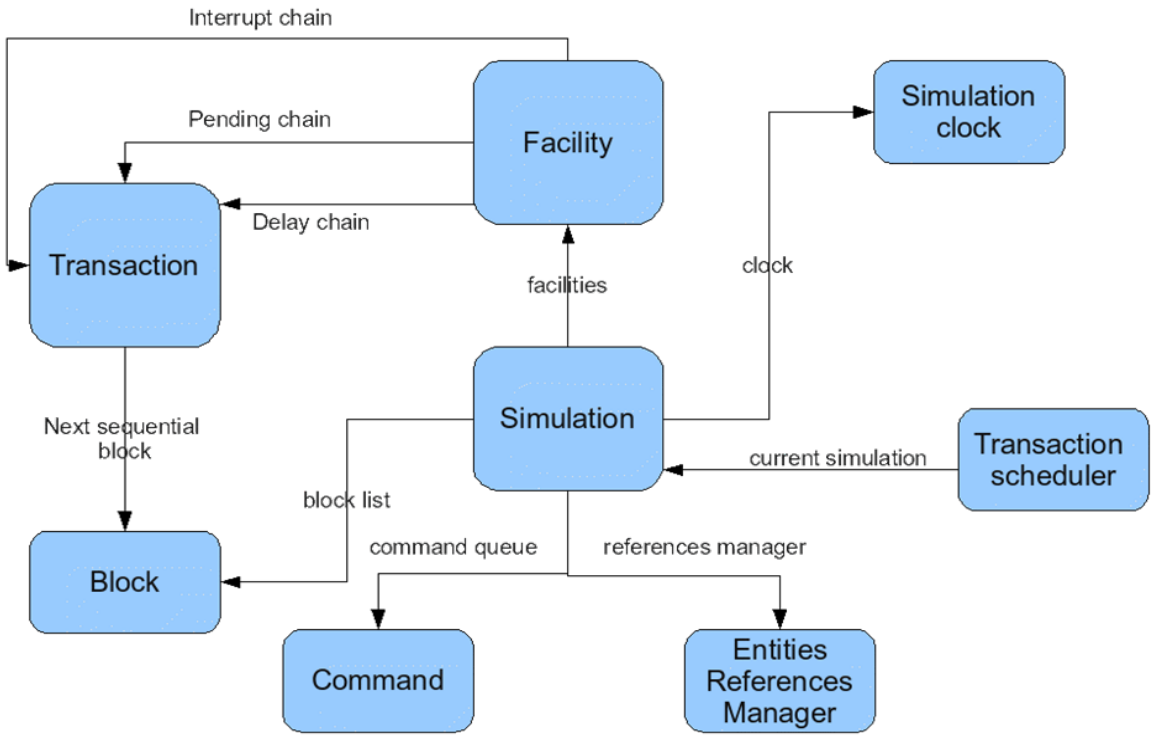

Figure 1: Simulation graph

Simulation entity is in charge of the execution of commands and the generation of transactions; it is also task of this entity to invoke the transaction scheduler which must decide which transaction have to be the next active transaction, and to manage the simulation clock, detecting when it must be updated and making all required tasks in each clock change; this implied the addition of Transaction Scheduler and System Clock entities inside the model.

Entities are dynamically being creating while the simulation runs, making them harder to locate them unless an extra system catalogue is used, adding more memory usage. To avoid this, it has been added an spare entity which deals with references to entities resolution, either by name or identifier; is entity is able to efficiently locate any entity at any time anywhere in the simulation, named Entity References Manager (ERM).

The entity Transaction Scheduler is extremely complex since it must interact, at first, with the two main chains of a simulation: current events chain and future events chain. But this entity must also be able to access transactions held by other chains in each existing entity every time is needed; for example, if a transaction is trying to release a facility, the transaction scheduler must select which transaction will be the following owner of the facility being released, meaning that it must analyze the current events chain, the delay chain of the facility, the preempt chain and the interrupt chain. Additionally, the scheduler must check if this transaction must be activated, queue in the CEC or stay where it this until other event ocurr.

The movement among chains is also a difficult task, since in implies not only the exchange of hundreds or thousands transactions from chain to chain, but it must also select transactions to exchange and choose the right moment to make this operation (change in the system clock, delay condition testing, and others). This taks is performed by both the Scheduler and Simulation entities, working together in order to keep the model integrity.

As mentioned before, Facility entity has been included in this first version of the framework. The decition of implementing this entity was not taken randomly; this entity involves a wide set of functions and entities, and has an intricate method for selecting transactions according to the internal state of each one of its chains and the way it is being accessed. With this entity, analyst may represent many diferent real world objects making model more useful; besides, this entity has set the basis for further implementations or other entities with intrincate chains orders such as storages or user chains. The hierarchical organization of all entities in the java model permits programmers to add new entities and reuse all functionality already implemented. For example, if a new entity has a special behavior and a FIFO, priority or BDT queue

3. Simulation persistence.

In order to store simulation in the disk, a relational database engine (MySQL) together with a java ORM (JPOX) has been used. As mentioned above, we are dealing with discrete simulations, hence we have a countable set of *simulation instants* in which many events have occurred and entities have been altered. So, we have consider to take a picture of the whole simulation, including all entities with their internal state, and to store this picture exactly as it is. The storage process happens every time the simulation clock is updated, just before this events actually occurs, in order to persist the simulation with the state before the clock changes.

Store a simulation in a determined time t implies to store all entities that take part in this simulation at this time t. In order to collect all entities, the composition graph is walked along, starting in the node representing the simulation entity and moving forward all simulation chains, entities list and chains of every entity. Since blocks are also entities, they must be persisted too.

Every time the graph is walked along, all persistible information is collected and, once it is finished, the simulation clock is updated and the execution goes on. Then, ideally, all this data should be persisted to disk and the system clock updated in order to go on with the run. The problem is that writing to the disk is too slow compared to main memory and processor speed, and we can not let the simulation stop until objects are stored.

Our solution to this problem consist of having a ready-to-store queue, and a low priority thread in charge of the persistence (Fig. 2). All data ready to persist is queued and remains in there until it is actually persisted, and the simulation continues running without waiting until data is written to disk. To persist objects, we have include a single thread, which runs together with the rest of the simulation. This thread picks up simulations ready from the persist queue and deals with DAO object and all database stuff (connections, queries, transactions and so on). It is important to remark that the extra thread has a lower priority than the simulation run; we have designed it this way because we do not want the simulation to be delayed for anything.

This solutions is good for having results immediately when the simulation ends while the persistence to the database is being made in background. The user may start analyzing data or adapting the model while the simulation keeps being stored in background, improving user experience and efficiency.

3.1 Data collection.

The whole application is written in Java, which means it uses all well known Object Oriented concepts (inheritance, hierarchy, OID, etc). When saving this objects to the database, we must make sure that they are not updated since we would loss the state of the objects before this update. One solution would be to have a version scheme where only new or changed data are stored; the main problem of using versions is that it would be really difficult to retrieve a simulation in any system clock. It would be efficient in disk, slow in data retrieve though.

Since hard disks are really huge these days, and computers have lots of memory, it is not expensive to duplicate data even if it has not changed. With this idea in mind, our solution has been both simple and

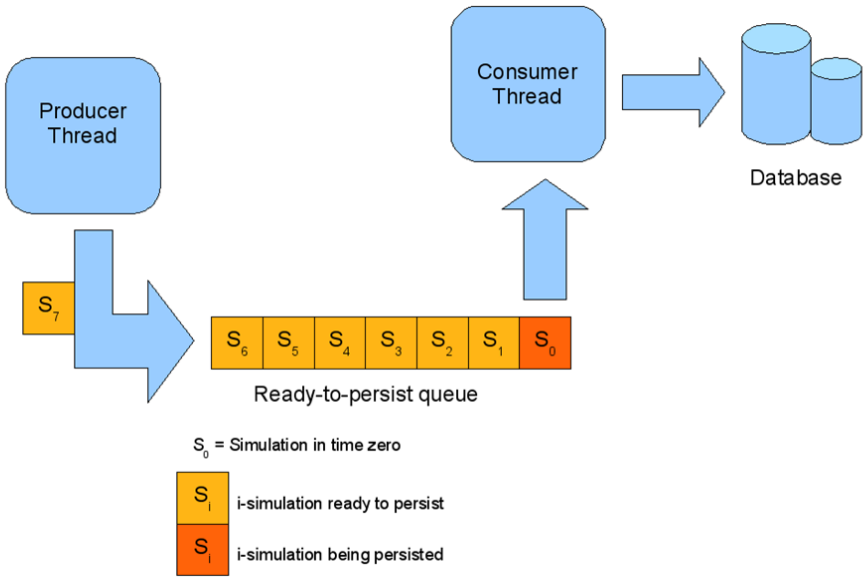

Figure 2: Producer/Consumer scheme

efficient: the whole simulation graph is cloned in every change of the system clock, copying every node that takes parts of the simulation (transactions, facilities, blocks, chains, and everything else). Once the cloning process has ended, the cloned simulation is queued to store and, as said above, the simulation can move on.

4. Data recovery.

The way data are stored permits to retrieve simulations very easily; in the database there are many copies of every simulation run, where each copy will differs in, at least, the state of the simulation clock. Hence, to recover a simulation in a specific time t is as simple as retrieve the Simulation object and again, start a chain effect retrieving all objects reachable from it.

But being able to retrieve a simulation in a time t is no the only advantage of this tool. All entities, both permanent and temporary, have an internal identifier, which never changes along the simulation. This is very useful to write new analysis tools that pick an specific entity and all copies generated before it; with all this information, this tool could show exactly how the entity changed, or which transaction were accessing it or were in its chains, or any other data that determine its internal state. This tool could even go beyond this ideas: we could have stored many simulation executions for the same model but with different parameters, and then we could retrieve any particular entity with all its states for all simulations. This way, we can not only see how this entity has changed, but to compare this changes with all instances of the same entity in different simulations.

6. Conclusion.

The framework is still being developed, incorporating new entities and blocks and improving the already developed ones. Besides the completion of GPSS model, this system is a great opportunity to every programmer who wants to add his own analysis tool, since all data of every simulation is ready to access and use; the fact that the framework is open source permits programmers to understand how data is organized, so they can efficiently retrieve it and show it the way they want.

Another goal of this implementation is that it can be used in any personal computer, since hardware requirements are the same for running any modern computer program. It is remarkably important that the whole framework can be installed in computers running any operating system that supports that Java Virtual Machine (Linux, Mac OS/X, MS Windows, OpenSolaris, among others).

There is also another important advantage that comes up from the use of an external database system. The framework data source is very flexible, permitting to distribute its location in other (or even computers) computers; this means that it can be used one computer for simulation entities processing and another for the database engine.

Besides the benefit of having a tool of this characteristics, there is another advantage not so obvious but very important: its use in academic environments to ease learning. Teaching simulation languages is a difficult task, since it requires to incorporate from one side a new way of executing sequential code where objects enter a blocks, execute a special routine and leave this block, and on the other side it simplifies the understanding of the operation of many entities very different and, in some cases, extremely complex. Here there is a development in a widely spread programming language and open source, which allows students from model and simulation areas to access and study *in an idiom they already understand and are used to* how these entities work, what actually mean some operations or routines, what happens then some functions are executed, and they can even change this behavior and learn by testing and customizing the framework.

References.

[AND99] Foundations of multi threaded, parallel and distributed programming. Gregory R. Andrews. Addison Wesley. 1999

[DUN06] Simulación y Análisis de sistemas con ProModel. Eduardo García Dunna. Heriberto García Reyes. Leopoldo E. Cárdenas Barrón. Pearson. 2006.

[JOR03] Java Data Objects. David Jordan, Craig Russell. O'Reilly Media, Inc. 2003

[JPO08] Java Persistent Objects.

http://www.jpox.org/

[LAN85] Teoría de los sistemas de información. Langefors, Börje. 2a. ed. (1985)

[KIM88]Schema versions and DAG rearrangement views in object-oriented databases (Technical report. University of Texas at Austin. Dept. of Computer Sciences). Hyoung Joo Kim. University of Texas at Austin, Dept. of Computer Sciences. 1988

[KIM90] Introduction to Object-Oriented Databases. Won Kim. The MIT Press. 1990.

[MIN05] Minuteman Software http://www.minutemansoftware.com

[SIM08] Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/

[VIT86] Design and Analysis of Coalesced Hashing. Jeffrey Scott Vitter, Wen-chin Chen. Oxford University Press, USA. 1986.

[WIL66] Computer simulation techniques. Wiley; 1st corr. printing edition (1966).