ObjectDtree: A simulation tool for business-like scenarios

Ing. Daniel Blank and Lic. Pablo Duboué

Universidad Empresarial Siglo 21
Rondeau 165 (5000) Córdoba – Argentina
{blankd, pablod}@uesiglo21.edu.ar

Abstract. The modeling of business-like simulations involves the need to deal with actors with a complex, usually non-deterministic, interaction-logic. This behavior is frequently modeled by the use of decision trees.

In this paper we introduce ObjectDtree, a discrete event simulator based on objects whose behavior is modeled by non-deterministic decision trees. The goal of our work is to develop a methodological approach and tools for the modeling and simulation of business situations, characterized by players interacting in a game-like, competitive environment.

Keywords. Business modeling, object oriented simulation, decision trees, functional language application

Introduction

The modeling of business-like simulations involves the need to deal with actors with a complex, usually non-deterministic, interaction-logic. This behavior is frequently modeled by the use of decision trees, a concept that comes from Game Theory (von Neumann and Morgenstern, 1953). This technique has widespread use (take for instance, Clemen 1991 and Schiffman and Kanuk 1997).

In this paper we introduce ObjectDtree, a discrete event simulator based on objects whose behavior is modeled by non-deterministic decision trees. The goal of our work is to develop tools for the modeling and simulating of business situations, characterized by actors interacting in a game-like, competitive environment.

Therefore, ObjectDtree is an implementation of a simulator for Discrete Event Simulation systems (DEVS) following the Object Oriented Programing (OOP) paradigm, toghether with the ability of capturing objects behavior by means of a mixture of tailored trees and imperative coding.

Following Nance, R. E. (1981), we consider a DEVS as a modeling technique that assumes that the system being simulated changes its states only at a set of finite (discrete) points in its simulation life-time. The simulation model jumps from one state to

another upon the occurrence of an **event**. There are so many similarities between DEVS and OOP that the later is usually used to implement the former.

Methodological Approach

Our proposed methodology is, basically, an OOP enriched with a non-traditional method to express the behavior of the distinct objects upon extern interaction. The previously stated intermixing of OOP and DEVS is captured in ObjectDtree by the use of data-enriched events. These kinds of events are of an expressive power perfectly compared to a usual message in OOP. The innovative aspect of our contribution is given by the use of decision trees in order to implement the response to each of these events/messages.

Decision trees

A decision tree is a decision making modeling tool. It is supposed to capture the sequence of steps taken by an individual in order to choose among a set of possibilities. The outcome of any choice fires a specific branch of more detailed questions until an decision is finally reached. Although in our approach the decisions are binary (yes/no) that implies no loss of generality. Table 1, taken from Schiffman and Kanuk (1997), shows there is broad use for yes/no questions.

Category	Alternative A	Alternative B
Consume decisions	To buy or to consume a product (or service)	Not to but or consume a product (or service)
Brand decisions	To buy or to consume an specific brand	To buy or to consume a different trademark
	To buy or to consume the client's own brand	To buy or consume an established brand
[...]	[...]	[...]
Payment decisions	To cash the purchasing	To pay the purchasing with a credit card
	To pay everything upon reception of the goods	To pay in quotas

Table 1. Different types of buy-related decisions

Formally, we define a d-tree as a 'n'-ary tree with queries as inner nodes and actions as leaves. Each query should have a discrete response set that should be possible to constrain with an upper bound beforehand. According to these bounds, a d-tree should

be provided for each possible outcome of the query. The leaves contain the actions or decisions of the individual whose behavior is being modeled. In order to keep this definition as broad as possible we impose no restrictions upon the actions that could be executed.

The simulator

In its current state, ObjectDtree is a batch simulation tool that receives as inputs a set of files describing the simulation to generate, and producing as output, the simulation log (Fig. 1).

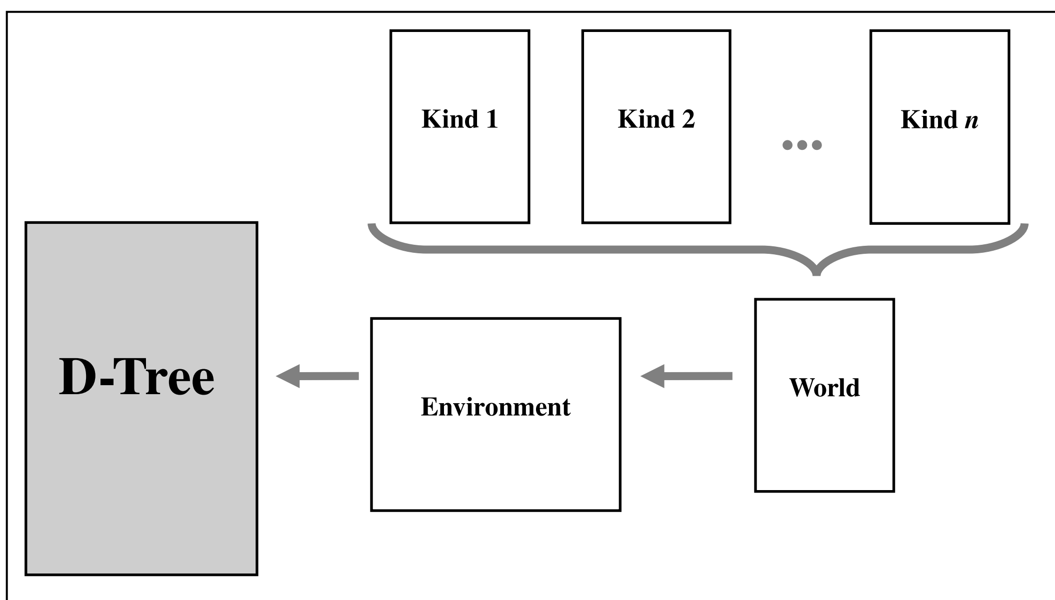


Fig. 1. Simulator diagram

Input files

As shown in Fig. 1 there are basically three types of input files:

- World files
- Environment files
- Kind definition files

In order to run a simulation, one Environment description file, one World description file and one Kind[1] file for each kind used are required.

---

[1] We use the term *kind* as a synonym to the term *class*.

The design rational used is the following: since we have different classes that we intend to use in different simulations, therefore it was natural to kept them in separate files for reuse (in a Java-like fashion). Then the World files contain the information about which classes happen to exist in each particular simulation and how many instances need to be created of each kind, together with instructions to initialize the attributes of each of those instances. An example of World file is shown in Fig. 2.

```
World {
  "client.kind": 200 instances with
    { patience = RandomNorm( 0.5, 0.001 )
    ; money = RandomNorm( 55.0, 10.0 )
    ; internetuse = RandomNorm( 15, 1.0 )
    ; buy = RandomNorm( 0.05, 0.001 )
    ; visit = RandomNorm( 0.5, 0.001 )
    ; netspeed = RandomNorm( 3.0, 0.1 )
    ; requestTime = 0
    ; books = 0
    }
  "bookstore.kind": 1 instances with
    { booksSold = 0
    ; bookInStore = RandomNorm( 0.75, 0.001 )
    ; visitsNum = 0
    ; timeOutNum = 0
    ; requestNum = 0
    ; cancelNum = 0
    ; found = 0
    ; price = 0.0
    ; prices = Random(10.0, 50.0)
    ; netspeed = RandomNorm( 10.0, 0.1 )
    }
}
```

Fig. 2. World file example

In addition to the World we use the concept of Environment in order to distinguish different simulation runs. The results obtained from a simulation can be quite sensible to initial values, termination criteria etc. Therefore, several runs of each simulation are required to obtain reliable data (Gogg and Mott 1995). We storage this kind of per-run data in a file, which encloses the **environment** where this especific run is being executed (not shown). Our primary concern here is the seed used by the pseudo-random number generator. However other information could be included such as termination criterion (maximum time reached or steady state) and, in case of

modeling system without perfect information, an extra *temperature* parameter could be used.

Kind files

The definition of each class is contained in the Kind files, being them the core of ObjectDtree. They have the general structure depicted in Fig. 3

```
Kind client {

  Attributes
    { attribute1: real          // comment
    ; attribute2: real
    ; otherAttribute: integer
    }

  Events {
    event SOMEEVENT
      { eventData1: integer
      ; otherEventData: real
      }

    event ANOTHEREVENT
      { moreData: integer
      }
  }

  Behavior {
    on SOMEEVENT {
      [ addEvent(
           ANOTHEREVENT { moreData = 5
        , event.otherEventData)
      ]
    }

    on ANOTHEREVENT {
      ask (event.moreData > attribute1 + Random()) {
        yes: { [ attribute2 = 5.0 ] }
        no:  { [ attribute2 = 1.0] }
    }
  }
}
```

Fig. 3. Kind file

In the previous figure we see that the kind files contain three parts:

- Attributes definition
- Events definition
- Behavior definition

The attributes are typed variables containing the state information of each instance. Currently only two types are supported: real and integer but extensions in this direction are straightforward. This attributes are read-only for the system and completely hidden for any other instance. Following usual OOP terminology they are defined as **private** information for each instance. However, any class can *sense* them by means of global, system-wide queries. This approach to attribute hiding is identical to the one taken in (Blank et al. 1996). It has the advantage of conforming to common distributed programming techniques (Chandy and Misra 1979), and facilitates the extension of ObjectDtree to a distributed environment.

The Events section following the Attributes section is needed by the one-kind-one-file approach used. All these events are broadcast, which means they are received by all the instances of every defined kind, however the sender ID is included in each event data so some granularity can be achieved.

The Behavior section contains a programmatically defined d-tree for each supported event. The inner nodes of the tree are the **`ask`** terms. They are yes/no questions whose answers are d-trees contained following the **`yes:/no:`** keywords. The actions are given by the sequence of instructions enclosed by [ ] brackets.

The query language includes Boolean algebra of logic terms, comparison of arithmetic terms containing system queries, event data and instance attributes. Two system queries are defined: **`HowMany`** and **`TheOne`**, with the following signatures:

`HowMany` (*kind name*, *condition*) *returns* **`integer`** ()

`TheOne` (*kind name*, *condition*) *returns* **`instance`** ()

The later picks in a non-deterministic fashion some instance of a given kind that verifies a given query (a runtime error is risen if none is found) and allowing to inspect its attributes. Regarding the discussion about the privacy of the attributes this is completely legal, as it is the system that is accessing the attributes of the instance. The former gives back the quantity of instances of a given kind who satisfies a condition.

The action language includes instructions to assign attributes, add events, and calculate statistics, among others. The interpreter internals make the extension of this language fairly simple.

Simulator internals

The simulator was written in Concurrent Clean, a lazy evaluation functional language developed at the **Research Group on Functional Programming Languages** of the **University of Nijmegen**, Holland (Brus et al. 1987). The use of a lazy functional language allowed us to speed–up the development cycle, obtaining an elegant solution that could be mathematically formalized. Moreover we were able to develop, using the available Clean tools, a Linux and Windows version of the interpreter without changing a single line of code.

An example: Internet Bookstore

The following situation has been developed as an application example:

There are 200 potential buyers of books (customers) over Internet interacting with a bookstore, which in turn is supplied by 5 editorials.

Each individual customer connects to the web with a frequency modeled by the attribute `internetuse`. For each connection the customer can hit the bookstore's site with a probability `visit`, with the probability of intention of buying a book modeled by the variable `buy`. When the customer is interested in buying a book she/he sends a request to the bookstore, which in turns should send back price and availability information. With this information the customer decides whether she/he finally closes the deal or not. However if the response takes too long the customer would eventually leave the site (time-out).

The seller (bookstore) is always looking for requests. When a request arrives the seller search the book in the stock. The variable BookinStore models the probability of the book being in stock. If the book is not available the seller asks the editorials (suppliers). The customer receives the event `ANSWER_TO_CUSTOMER` containing price and availability information.

The editorial (supplier) is modeled by means of an object, which reacts to the BOOKSTORE`_REQUESTS` event with an `ANSWER_TO_BOOKSTORE` message indicating the availability of the book (modeled by its attribute `BookinSupplier`).

Other attributes are used to represent the characteristics of the problem such as book prices, customer monetary capacity, maximum wait time, and so on.

The behavior of the bookstore, the editorials and the customers is modeled by means of decision trees. Features describing the incidence of customer satisfaction and spread of rumors on behavior are also included.

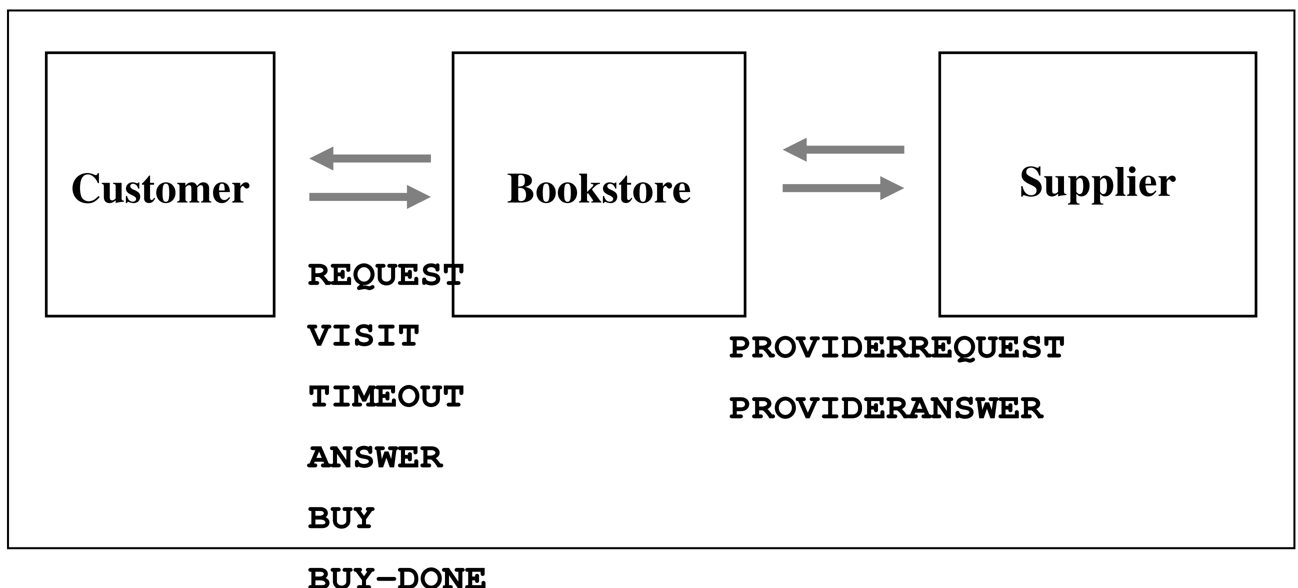
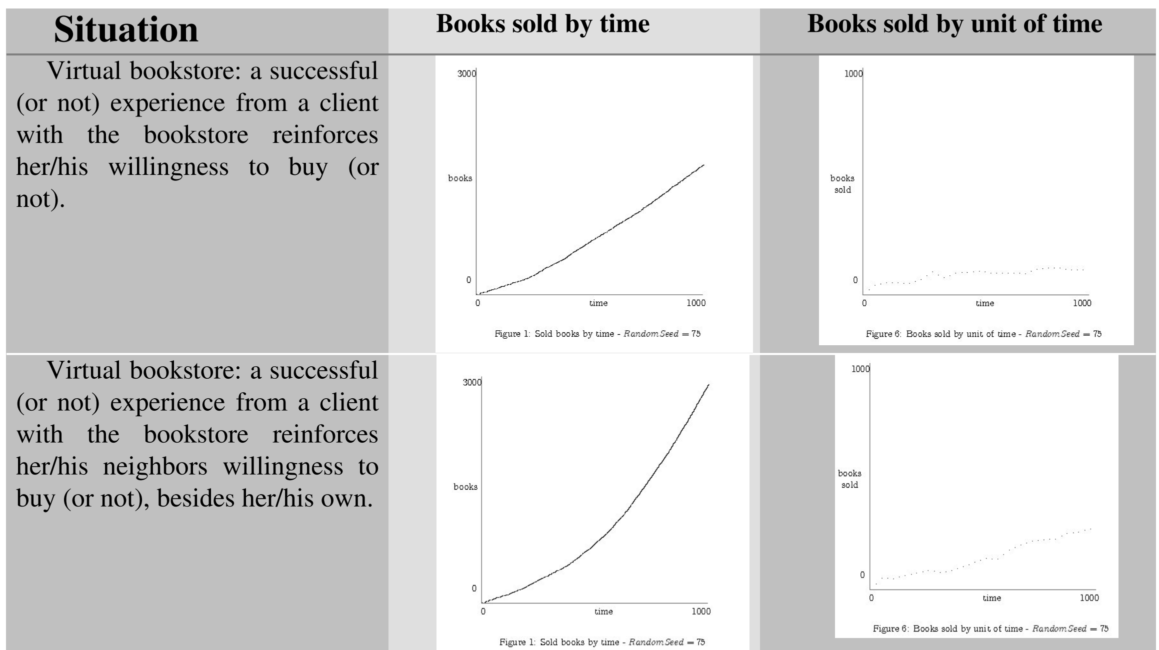


Fig. 4. Example diagram



Situation	Books sold by time	Books sold by unit of time
Virtual bookstore: a successful (or not) experience from a client with the bookstore reinforces her/his willingness to buy (or not).	3000 books 0 0 time 1000 Figure 1: Sold books by time - *RandomSeed* = 75	1000 books sold 0 0 time 1000 Figure 6: Books sold by unit of time - *RandomSeed* = 75
Virtual bookstore: a successful (or not) experience from a client with the bookstore reinforces her/his neighbors willingness to buy (or not), besides her/his own.	3000 books 0 0 time 1000 Figure 1: Sold books by time - *RandomSeed* = 75	1000 books sold 0 0 time 1000 Figure 6: Books sold by unit of time - *RandomSeed* = 75

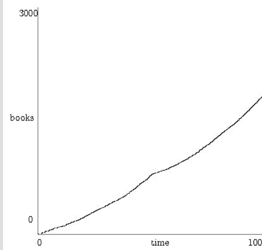
<table>
<tr>
<td>Virtual bookstore: as in the previous experience but a strong negative rumor is injected into the system at t=500.</td>
<td>

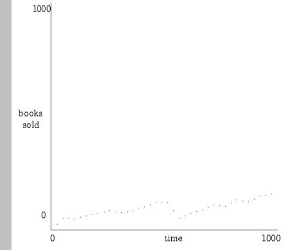
Figure 1: Sold books by time - $RandomSeed = 75$</td>
<td>

Figure 6: Books sold by unit of time - $RandomSeed = 75$</td>
</tr>
</table>

Table 2. Results table

Conclusions

We find that our simulator fulfills the motivations that ended up in its building. The resulting tool is robust and quite usuable. Moreover, its actual state seems to give us an starting point for further development in different ways, as we will see in the next section.

It could be argued that our d-trees have no difference with programming cascade `ifs` in a usual OOP language. From the expressive power of the language this is possible. However, by taking the d-tree out from the shadows we are able to **manipulate it** as any other piece of software.

Besides, it suggest an interesting methodological approach for modeling managment, marketing and the similar situations. In order to develop it completely it is necesary to extend this work multidisciplinarily. It also stands to be compared with other approaches in order to see its benefits and limits.

Another point we were interested in was benchmarking lazy functional language against CPU-intensive task such as DEVS. With this objective in mind, the choosing of Clean as our programming platform was a right decision. The lazy features of the language allowed us to capture some characteristics of DEVS simulation, such as state duplication, etc. in a more economical fashion compared against usual (eager) non-functional languages.

Further work

ObjectDtree is currently in active development with new features being added at fast pace. Most of them fall into extensions to the language used to describe the objects and user friendly interfaces to help the modeler's tasks. Moreover, there are some extensions we plan to include in the system that will greatly expand the expressive power of the tool, namely: adaptive decision trees (including genetic d-trees) and middle-road actions.

The idea behind adaptive decision trees is to incorporate means to specify dynamic changes to the strategy used by each kind. This can be taken to its limit by allowing two different instances to merge their adapted d-tree into a new, child-like, one.

Middle-road actions, on the other hand, change the definition of the trees by allowing the execution of actions in the inner nodes. Our main interest here is to simulate queries with side effects.

Another promising area of extension is the migration of the simulator to a distributed environment, following Chang and Jones (1994). The most difficult part of the task is getting rid of the global clock and global event queues. In spite of it, the way we protect each instance variable as explained above, together with our programming platform (lazy functional languages are well know for its facility to migrate to parallel environments) makes it quite promising.

References

Barcio B., Ramaswamy, S., Macfadzean, R., Barber, K.: Object Oriented Analysis, modeling and simulation of a National Air Defense. Simulation magazine vol 65 num 1 (1996)

Blank, D., Bruno, N., Duboué, P. A.: Desarrollo de un generador de simulaciones basado en especificaciones visuales. In Proc. IX Simposio Internacional en Aplicaciones de Informática (INFONOR'96), Antofagasta, Chile (1996)

Brus, T., van Eekelen, M.C.J.D., van Leer, M., Plasmeijer, M.J., Barendregt, H.P.: CLEAN – A Language for Functional Graph Rewriting. In Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, Kahn Ed., Springer-Verlag, LNCS 274, pp. 364-384 (1987)

Chang, W. and Jones, L. R.: Message-Oriented Discrete Event Simulation. Simulation magazine vol 63 num 2 (1994)

Chandy, K. M. and Misra, J.: Distributed simulation: A case study in design and verification of distributed programs. IEEE SE 5:440-452 (1979)

Clemen, R. T.: Making hard decisions. PWS – KENT Publishing Co., Boston, MA (1991)

Gogg, T. J. and Mott J. R.: Improve quality & productivity with simulation. JMI Consulting Group 2nd ed. (1995)

Nance, R. E.: The time and state relationships in simulation modelling. ACM. 24(4):173-179 (1981)

Neelamkavil, F.: Computer simulation and modelling. John Wiley & Sons, New York, N. Y. (1987)

Schiffman, L. G. and Kanuk, L. L.: Comportamiento del Consumidor. Addison Wesley Latinoamericana 5th ed. (1997)

Von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior. Princeton University Press, 3rd ed., (1953)