# An Object Oriented C++ Approach for Discrete Event Simulation of Complex and Large Systems of Many Moving Objects

Mauricio Marín
Departamento de Computación
Universidad de Magallanes
Punta Arenas, Casilla 113-D, Chile
mmarin@ona.fi.umag.cl

Patricio Cordero
Departamento de Física
Universidad de Chile
Santiago, Casilla 487-3, Chile
pcordero@uchcecvm.cec.uchile.cl

**Abstract**

*Modelling and programming process requires a completely different approach in discrete event simulation of complex and large systems of many moving objects. In this paper we describe the public interface of a C++ class library that addresses this problem. Our class library has the virtue of being a flexible and application-independent object-oriented environment which presents to the user a* world view *that simplifies the simulation of these systems.*

## 1 Introduction

Discrete event simulation can be a powerful analytical tool in the study of complex systems based on many moving objects. Its application ranges through a variety of systems, including airplane or vehicular traffic, message passing on wide area networks or microscopic fluids.

The generic problem consists in simulating the behavior of each moving object in a space populated by many other objects. The main aspect of the simulation are the interactions between objects. For example, an airplane which enters a region covered by a radar [2] or elastic collisions between particles in a fluid [6]. Each interaction is an event that occurs at discrete instant of time during the simulation. In addition, events generated by each object may exist, as is the case of an airplane that changes its route according to a pre-established flight plan. Between events, nothing interesting happens, so the simulation advances through variable intervals of time driven by the chronological occurrence of the events.

Even for simple simulation models, it is no obvious how to reduce the high cost associated with the modelling and programming process using commercial simulation products such as SIMSCRIPT, GPSS or SLAM. Many moving objects simulation requires a completely different approach and it is necessary to develop new strategies to be able to reduce the effort involved in the analysis of complex and large systems. Steps in this direction have been given by [2].

In this paper we propose a C++ class library that addresses the modelling and programming process of complex and large systems of many moving objects. Our interest is centered in systems consisting of many thousands of objects, with several object classes, and binary interactions between them. Rather than implementation details, we describe the conceptual model or *world view* that our library provides to the user through its public interface. Furthermore, we present an example showing that our world view has the virtue of increasing the user productivity providing an environment flexible and independent of the applications which has a logic more descriptive than procedural. Thereof we show the ubicuity of an object-oriented languague as C++ in this kind of simulations.

### 1.1 Library Design

The class library is divided in two main components:

- Simulation system or *kernel*; that contains the data structures and algorithms used to make the efficient simulation of general systems of many moving objects. We have implemented the kernel using generalized versions [1] of our strategies originally deviced for microscopic fluids [5], although there are other strategies that may be used in this simulations [3, 4, 7, 8].

- Modelling system or *builder*; that enables the communication between the specific features of each simulation model and the kernel. These features are mainly related with the classes to which

each object belongs and the functions that determine the interactions between them. This module provides formalisms that define: the object classes; the objects interactions; the grouping of interactions in parallel processes; and the strategy used to make the automatic allocation of objects in the space of the model.

The protagonists of our conceptual model are instances of the set of classes stored in the library – called *descriptors, patterns, sheets, xlists* and *maps* – which are used to build a valid model that can be simulated by the kernel. The methodology used is a sort of client-server model where the user initializes an interface (the model) that is passed to the server (the kernel), which proceeds to make the simulation.

For the sake of simplicity the class descriptions are given for two dimensional systems. Each class is summarily described and it is assumed there may exist constructors, destructors and other additional methods that performs tasks such as, for example, the graphic display of objects on the screen. The greater part of the public names used in the library are auto-explicatives, so we do not describe them in detail.

### 1.2 Example (part 1)

We show how to use our class library by developing an illustrative aplication example which describes the modelling and programming process of a two dimensional microscopic fluid. Each part of the example is developed in turn in each section of the paper according to the new concepts there introduced.

The fluid is modeled as a set of hard disks uniformly distributed in a rectangular space, which has a circular obstacle in its center. The obstacle can have two states — denoted $A$ and $B$. The simulation objects are the hard disks and the obstacle, and their interactions are their ellastic collisions.

Some variables and functions in connection with the physical aspects of the model, whose description leaves the scope of this article, are the following:

```
long Collisions=0;
long MaxCollisions=1000000;
void TestEndCondition();
void Init();
void End();
// DD = Disk-Disk
// DO = Disk-Obstacle
void CollisionTimeDD();
void CollisionTimeDO_A();
void CollisionTimeDO_B();
void ChangePositionsDD();
void ChangeVelocitiesDD();
void ChangePositionD();
void ChangeVelocityDO_A();
void ChangeVelocityDO_B();
```

The rest of the paper is organized as follows. The classes used to define the simulation objects are described in section 2. Section 3 gives the classes to create the objects and allocate its positions in the space of the model. Section 4 presents the classes that define the interface between the simulation model and the kernel. Final comments, conclusions and future investigations in this area are given in section 5.

## 2 Objects

Let us distinguish between active and passive objects. Active objects perform some perceptible activity during the system operation. Conversely, passive objects represent inert system entities. The kernel only performs the simulation of active objects. Passive objects are considered as partners in the active object interactions.

### 2.1 Descriptors

A *descriptor* is a user defined class that contains the data structures and functions associated with a collection of objects, namely it defines their computer implementation. An object collection is created using the same instance of a specific descriptor that has been adequately initialized by the user. All descriptors must be defined as a derived class from one of the following two library classes,

```
class BDescriptor {
public:
  BDescriptor(int Type, int States);
  void FirstState(int State);
  void ChangeState(int NewState);
};

class GDescriptor: public BDescriptor {
public:
  void AddCircle(double X, double Y,
                 double Radius);
  void AddLine(double Xa, double Ya,
               double Xb, double Yb);
  ...
  //GC = Geometric Center
  void SetPositionGC(double X, double Y,
                     double Angle);
  void GetPositionGC(double &X, double &Y,
                     double &Angle);
};
```

where *Type* defines whether the objects are actives or passives, and *States* identifies the maximal number of discrete states that may attain each object. We indentify the behavior of each active object using a private integer discrete state variable, which is set with the *FirstSate()* method and subsequently modified with the *ChangeState()* method. The methods *Add*()* are used to build a list of geometric primitives which define the approximate shape of the objects. The coordinates $(X, Y)$ of each primitive are given according to an origen $(0, 0)$ that is considered the geometric center of the object.

### 2.2 Example (part 2)

For the microscopic fluid used as an example, we define the descriptors *CDisk* and *CObstacle*, and the initialized instances of these descriptors *Disk* and *Obstacle* such as is shown in the following program fragment,

```
class CDisk: public GDescriptor {
private:
  double VelocityX, VelocityY;
public:
  void SetVelocity(double Vx, double Vy)
       { VelocityX=Vx; VelocityY=Vy; }
  void GetVelocity(double &Vx, double &Vy)
       { Vx=VelocityX; Vy=VelocityY; }
};

class CObstacle: public GDescriptor {
private:
  double temperature;
public:
  void SetTemperature(double t){temperature=t;}
  double GetTemperature(){return temperature;}
};

CDisco Disk(ACTIVE,1);
CObstaculo Obstacle(PASSIVE,2);// 2 states

void Descriptors()
{
//Initialize geometric form
  Disk.AddCircle(0.0,0.0,0.25);
  Obstacle.AddCircle(0.0,0.0,0.5);
}
```

The instances *Disk* and *Obstacle* will be used to create the simulation objects and to define the interface with the kernel.

In our approach, it is not necessary that each descriptor represents a physical or concrete system entity. They may also represent to abstract entities such as weather and flux measurer. Furthermore, it is not a requirement that all the objects have geometric shapes; consequently we have defined the class *GDescriptor* as a derived class of *BDescriptor* (the Base Descriptor). The user may define other derived classes from *BDescriptor* as Queue Centers for example. Therefore, the term active object is not strictly a synonym of neither moving object nor concrete entity with geometric shape.

## 3 Space

In this section we present the methodology used in the automatic allocation of objects with geometric shape into the space of the model. We define *space* as the rectangular area where the geometric objects are placed. The origin $(0, 0)$ for the spatial coordinates is located in the lower left corner of the space. Basically, the objects come to the space from one or more *sheets* previously created. These sheets are created making the *fusion* of other sheets; duplicating *patterns* in their spread; or inserting objects individually into them.

### 3.1 Patterns

A *pattern* is a small rectangle that contains one or more geometric objects. All patterns defined for a simulation model are instances of the following library class,

```
class Pattern {
public:
  Pattern(double X, double Y);
  void AddObject(GDescriptor &D, double Xgc,
                 double Ygc, double Angle);
};
```

where $X$ and $Y$ set the rectangle size and the method *AddObject()* is used to place objects into the pattern. The geometric center of each object is placed in the position given by the parameters *Xgc* and *Ygc* with rotation angle given by *Angle*. In this case the origin $(0, 0)$ is the lower left corner of the rectangle defined by $X$ and $Y$.

### 3.2 Sheets

A *sheet* is a large rectangle that contains objects created duplicating a pattern in all its extent; adding objects individually; or making the fusion of two sheets. Each sheet is an instance of the following class library,

```
class Sheet {
public:
  Sheet(double X, double Y,
        double Xo, double Yo);
  void Duplicate(Pattern &P);
  void AddObject(GDescriptor &D, double Xgc,
                 double Ygc, double Angle);
  void Fusion(Sheet &S1, Sheet &S2,
              boolean V, boolean W);
};
```

where $X$ and $Y$ set the rectangle size, and $Xo$, $Yo$ specify the rectangle position (its lower left corner) within the space. The parameter $P$ identifies the pattern to be duplicated — using the method *Duplicate()* — into the area defined by $X$ and $Y$.

The fusion of two sheets is performed by the method *Fusion()* which makes the join of the sheets *S1* and *S2* — where $V$ and $W$ may take the values *TRUE* or *FALSE*. If $V$ is *FALSE* the objects in *S1* are not included in the final sheet. Inversely, when $V$ is *TRUE* all the objects in *S1* are included in the final sheet. When $W$ is *TRUE* are included in the final sheet all the objects of *S2* that do not intersect an object of *S1*. If $W$ is *FALSE* are included in the final sheet all the objects of *S2* that intersect at least one object of *S1*.

### 3.3 Space

The space of the model is an instance of the following library class,

```
class Space {
public:
  Space(double X, double Y);
  void Objects(int n, Sheet &S ...);
  int  InsertObject(GDescriptor &D, int V,
                    double Xgc, double Ygc,
                    double Angle);
  void DeleteObject(int Identifier);
};
```

where $X$ and $Y$ specify the extent of the rectangular space. The method *Objects()* has a variable argument list and it is used to insert within the space the objects stored in the sheets specified in the argument list. All the objects stored in the $n$ sheets $S$ are inserted into the space.

During simulation, it is possible to delete and create objects dynamically. Each object is identified by an integer number. The method *InsertObject()* creates a new object, inserts it into the space, and returns the identifier assigned to the new object. When $V$ is *TRUE* all the objects that intersect the recently created object are deleted. If $V$ is *FALSE* the new object is created and inserted without considering the intersections with other objects. When $V$ is *CONDITIONAL* the object is created and inserted only if it does not intersect other objects.

### 3.4 Example (part 3)

For the microscopic fluid used as an example, the objects and space are created as follows,

```
Space SimSpace(1000.0,1000.0);

void Objects()
{
   Pattern PDisk(1.0,1.0);
   PDisco.AddObject(Disk,0.5,0.5,0.0);

   Sheet SD(1000,1000,0,0);
   SD.Duplicate(PDisk);

   Sheet SO(1000,1000,0,0);
   SO.AddObject(Obstacle,500,500,0.0);

   Sheet SDO(1000,1000,0,0);
   SDO.Fusion(SO,SD,TRUE,TRUE);

   SimSpace.Objects(1,SDO);
}
```

## 4 Simulation

Let us define *xlist* as a list of pointers that is used by the kernel to execute the functions or routines pointed to by these pointers. Let us also define *map* as an input-output structure which contains one xlist in each output and has input parameters that allow the kernel to find these xlists within the map. Class descriptors, object states and interaction types are used as input parameters.

In all simulation model there are two maps:

- Predictions map; which contains xlists that point to prediction functions used to calculate the interaction times between the simulation objects, and
- Interactions map; which contains xlists that point to interaction functions used to make the change of states of the objects involved in an interaction.

The kernel performs the simulation of the system making accesses to the predictions and interactions maps in order to execute the functions pointed to by the xlists. These model-dependent functions pointed

to by the members of the xlists have the responsability of the correct evolution of the system being simulated. The kernel only makes calls to these functions at the precise instants.

Each xlist may contain other general purpose model-dependent functions (called control functions) which may be used in tasks such as, for example, system evolution measuring.

Given two objects $i$ and $j$ — with states $e_i$, $e_j$ and class descriptors $d_i$, $d_j$ — the input parameters for the predictions and interactions maps are $\{d_i, d_j, e_i, e_j\}$. In addition, if it has been defined the interactions $a$ and $b$ between $i$ and $j$ (interactions valid at the states $e_i$ and $e_j$) the input parameters for the interactions map may also include either $a$ or $b$ according to the interaction ($a$ or $b$) that is being processed just in the instant when the kernel accesses this map.

The simulation algorithm performed by the kernel consists in: searching and executing one or more xlists of the predictions map; getting the chronological next event that must take place; processing this next event searching and executing one xlist of the interactions map; returning to the predictions map and so forth until the end condition of the simulation is reached.

More in detail, when ocurrs an interaction $a$ between the objects $i$ and $j$, the kernel executes the xlist associated with the input parameters $\{d_i, e_i, d_j, e_j, a\}$ in the interactions map. Then, the kernel make predictions — executing xlists stored in the predictions map — between the object $i$ and all the objects $k$ for which it has been defined a xlist associated with the input parameters $d_i, e_i, d_k, e_k$ (the same is repeated if $j$ is an active object). Then, the kernel determines the next event and again accesses the interactions map and so on.

It is possible to form groups of xlists within the predictions and interactions maps. These groups are used to build parallel process — called *activities* — associated with each object class. Several activities can be defined for a given object class.

An activity may be seen as a set of predictions and interactions which are simulated by the kernel individually each other and for each object associated with. The grouping among predictions and interactions is set using an additional input parameter $p$.

When an interaction associated with an activity $p$ takes place, the predictions performed by the kernel during the process of this event are only realized between the objects whose class descriptors are members of $p$. In other words, if it has been defined that the class descriptors $d_i$ and $d_j$ belong to the activity $p$, then when an interaction in $p$ for an object $i$ ocurrs the kernel performs predictions between $i$ and all the objects whose class descriptors are $d_i$ or $d_j$.

The activities focuses the predictions to determined object classes making possible the existence of several independently simulated processes for the same object.

Conditional events can be used to sincronizate the activities since these events are executed when a given condition is reached. For example, a condition may test the end of one or more activities and the associated event (which is triggered when the condition is true) may begin the execution of a new activity.

The object interactions are strictly binary although the particular case of unary interactions can be implemented using a predefined void class called *VoidDescriptor*. This void class can be used to model activities that are uniquely associated with an specific class without having relation with other object classes.

## 4.1 XLists

A *xlist* is a list of function pointers and must be an instance of the following library class,

```
class XList {
public:
  void Insert(char *List ...);
  void Delete(char *List ...);
  void Execute();
};
```

where the methods *Insert()* and *Delete()* are used to add and remove function pointers in a xlist. These methods have variable argument lists. The parameter *List* is a string that specifies — by mean of "f" letters — the number of pointers placed in the argument list . For example, three pointers are specified by the value *List*= "fff". Furthermore, it is possible to assign a priority value to each function pointer by associating an integer number to each "f". For example, the value *List*= "f1f2f3" sets three pointers with priorities 1, 2 and 3. In this case, the kernel execute the functions in priority order (1 has greater priority than 2 and 2 greater than 3). The function pointers are placed after parameter *List* and it is assumed the type *void (*function)()* for these pointers.

## 4.2 Maps

A *map* sets relations among descriptors, states, interactions and xlists. Each map is an instance of the following library class,

```
class Map {
```

```
public:
  Map(int Type);
  void Connect(Descriptor &D1, Descriptor &D2,
               int StateD1, int StateD2,
               XList &XL, int Activity=0,
               int Interaction=0);
  void Connect(Descriptor &D1, Descriptor &D2,
               int StateD1, int StateD2,
               char *List ...);
  void Disconnect(Descriptor &D1, Descriptor &D2,
                  int StateD1, int StateD2,
                  int Activity=0,
                  int Interaction=0);
};
```

where *Type* specifies if the instance is a predictions or interactions map. The first method *Connect()* is used to associate one xlist *XL* with each combination of descriptors *(D1, D2)* and states of objects *(StateD1, StateD2)*. The parameter *Activity* is used for grouping combinations of descriptors and states into activities.

If the instance of the class map has been defined as a predictions map, the parameter *Interaction* is ignored. But, if this instance is an interactions map the parameter *Interaction* can be used to define several interactions between each pair of descriptors and states.

In a similar way as the parameter *List* is used in the class XList, the parameter *List* in the second method *Connect()* can be used to specify directly the xlist, and can be also used to define activities and interactions. For interactions the parameter *List* include the letter "i" followed by an integer, and for activities the letter "a" sufixed by an integer.

The method *Disconnect()* can make the dynamic disconnection of the relations established between descriptors, states, activities, interactions and xlists by the connect() methods.

The default activity (Activity=0) consists in making predictions with all the objects of the system, as these objects belong to classes that have been associated with one xlist. Also, as parameters for the connect and disconnect methods can be used the predefined constants *AllDescriptors* and *AllStates*.

### 4.3 Example (part 4)

For our example of the microscopic fluid we have defined the following maps and xlists,

```
// XLists y Mapas

XList XInit;//Initialize and
      XEnd; //Finalize the simulation.
      XPred;//calls TestEndCondition()

Map   MP(PREDICTIONS);  // predictions map
Map   MI(INTERACTIONS); // Interactions map

void XListsMaps()
{
   XInit.Insert("f",Init);
   XEnd.Insert("f",End);
   XPred.Insert("f",TestEndCondition);

   MP.connect(Disk,Disk,0,0,"f1",
              CollisionTimeDD);
   MP.connect(Disk,Obstacle,0,0,"f1",
              CollisionTimeDO_A);
   MP.connect(Disk,Obstacle,0,1,"f1",
              CollisionTimeDO_B);

   MI.connect(Disk,Disk,0,0,"f1f2",
              ChangePositionsDD,
              ChangeVelocitiesDD);
   MI.connect(Disk,Obstacle,0,0,"f1f2",
              ChangePositionD,
              ChangeVelocityDO_A);
   MI.connect(Disk,Obstacle,0,1,"f1f2",
              ChangePositionD,
              ChangeVelocityDO_B);
}
```

### 4.4 Event List

The *Event List* is the data structure where the event generated during the simulation are stored. The kernel has the responsability of performing an efficient administration of these events. The library provides the following methods related with the Event List,

```
class Event {
public:
  void SetEvent(double Time,
                int IdObj1, int IdObj2,
                int Activity=0,
                int Interaction=0 );
  void SetEvent(int (*F)(),
                int IdObj1, int IdObj2,
                int Activity=0,
                int Interaction=0 );
};

class Kernel {
public:
  void Schedule(Event &E);
  void Cancel(Event &E);
  void Cancel(int IdObj, int Activity=0);
  void Cancel(int IdObj,
              int n, Descriptor &D ...);
  void Predictions(int OnOff, int IdObj,
                   int Activity=0);
```

```
  void Predictions(int OnOff, int IdObj,
                   int n, Descriptor D ...);
      ...
};
```

where the *SetEvent()* methods are used to initialize the user events that will be stored in the Event List. Each user event is an instance of either the class *Event* or any other derived class from *Event*. These methods allow defining either a user-event that will take place at the instant given by the parameter *Time* or a conditional user-event that will take place when the function pointed to by the parameter $F$ returns the value 1. All the conditional user-events are tested to ocurr before scanning a non-conditional next event in the Event List.

The method *Schedule()* inserts an event $E$ in the Event List. The first *Cancel()* removes a specific event from the Event List. The second *Cancel()* removes all the events scheduled for the object *IdObj* and the specified *Activity*. The third *Cancel()* removes from the Event List all the events related with the object *IdObj* and the objects partners that belong to the $n$ classes $D$.

The methods *Predictions()* control the access to the predictions map during the process of an interaction. The effect of these methods "vanishes" after processing the interaction. The parameter *OnOff* enables or not the predictions for the object *IdObj*. Furthermore, these predictions can be restricted to either a specific *Activity* or the set of classes especified by the $n$ descriptors $D$.

In each step of the simulation, the kernel initializes a global object called *CurrentEvent* which contains data associated with the event that is being processed currently. *CurrentEvent* must be declared by the user as an object of class *Event* or some other user defined class derived from *Event*.

### 4.5 Kernel

The kernel contains the algorithms that perform the model initialization and its efficient simulation. The methods used for doing these tasks are given in the following library class,

```
class Kernel { // continued
public:
  ...
  void Space(Space &S);
  void SetXLists(XList &B, XList &P,
                 XList &I, XList &E);
  void Maps(Map &MP, Map &MI);
  void Start();
  void Stop();
  void Continue();
  void Finish();
  int  CreateObject(Descriptor &D1);
  void DeleteObject(int Identifier);
};
```

where the parameter $S$ indicates the space of the model; $B$ and $E$ indentify the XLists that will be ejecuted during initialization and finalization of each simulation period respectively; $P$ indicates a XList that is ejecuted each time that is initiated a secuence of predictions; and $I$ is a XList that is ejecuted before processing an interaction that takes place during simulation.

The parameters *MP* and *MI* specify the maps of predictions and interactions respectively.

The method *Start()* is used to initiate a new simulation period which is terminated by the methods *Stop()* or *Finish()*. *Stop()* temporally terminates the simulation period being possible to continue it by using the method *Continue()*. *Finish()* terminates completely the simulation.

The methods *CreateObject()* and *DeleteObject()* are used to create and eliminate objects without geometrical shape.

### 4.6 Example (final part)

Finally the initialization and simulation of the microscopic fluid is realized as follows,

```
// Initialization and simulation

Kernel Sim;
Event CurrentEvent;

void main()
{
   Descriptors();
   Objets();
   XListsMaps();
   Sim.Space(SimSpace);
   Sim.XLists(XInit,XPred,XVoid,XEnd);
   Sim.Maps(MP,MI);
   Sim.Start();
   Sim.Finish();
}
```

## 5 Conclusions

The application independence is ensured because the class library works with class descriptors and system-dependent functions defined by the user.

Class descriptors are used to create the simulation objects which represent the system entities and system-dependent functions are used for modelling the behavior of the entities and its interaction with each other.

Xlists provide the flexibility needed in this simulations because it is possible to insert and remove function pointers from the them during simulation. For example, on a first stage of the simulation it is necessary to hold in xlists to functions dedicated to validate the model, and after on next stages others functions are needed to study the system evolution. Xlists increase the efficiency of the simulation because only the strictly necessary pointers can be holded in each one.

Maps provide an intuitive mechanism for defining the relations between class descriptors, object states, interactions and xlists. These relations can be grouped in activities for representing parallel processes associated with the simulation objects. This concept can be used to define several independent processes associated with determined object classes which are simulated individually by the kernel for all the objects that belongs to these classes. Conditional events enable the activity sinchronization.

The kernel only realize predictions and interactions for the objects whose classes have been related through a xlist within the predictions and interactions map, so it is possible to define objects of very distinct significance in the same simulation model: some objects might "see" to objects that belongs only to determined classes, while others ones might "see" to all of the simulation objects; moreover during simulation these relations might change dinamically.

Finally, patterns, sheets and fusions provide to the user a general formalism that simplify the process of creating and allocating thousands of object with geometrical shape in the space of the model.

Currently our research work is oriented toward the problem of creating a graphical language that can be used to define simulation models of many moving objects systems. In this case, the class library here described can be used as a low-level language so that the outcome of the specifications given in the high-level graphical languague can be transformed to C++ declarations such as in the microscopic fluid used as an example in this paper.

Another research activity focuses on how to use the Rule-Based Expert System Technology for providing to the user a more descriptive mechanism than the prediction and interaction functions mechanism described in this paper. In this case, the partial or global state of the simulation can be considered as the fact data base and the production rules can be used for guiding the behavior of the active objects during simulation. In this case, within the xlists must exist pointers to functions that send messages to the inference machine of the expert system for obtaining some answer represented in the form of events that are triggered inmediatly or some steps ahead during simulation.

## Acknowledgements

Partially supported by *FONDECYT* grant 1931105 and University of Magallanes grant F1-01IC-94. Thanks to Dr. Baeza-Yates for encouraging the presentation of this work.

## References

[1] R.Baeza-Yates, M. Marín and P.Cordero, "The Analysis of an Improved Priority Queue for Discrete-Event Simulation of Many Moving Objects", Proceedings of the XIV International Conference of the Chilean Computer Science Society, Concepción, Nov. 1994, Chile.

[2] P.D.Corey y J.R.Clymer, "Discrete Event Simulation of Object Movement and Interactions", Simulation, **56** (1991) 167.

[3] G.Gonnet and R.Baeza-Yates, *Handbook of Algorithms and Data Structures*, (Addison-Wesley, NY, 1991).

[4] B.D.Lubachevsky, "How to simulate billiars and similars systems", Journal Computational Physics, **94** (1991) 255.

[5] M.Marín, D.Risso and P.Cordero, "Efficient Algorithms for Many-Body Hard Particle Molecular Dynamics", Journal of Computational Physics, **109** (1993) 306.

[6] D.C.Rapaport, "Time Dependent Patterns in Atomistically Simulated Convection", Phys. Rev. Lett., **43** (1991) 7046.

[7] K.Shida and Y.Anzai, "Reduction of the Event-List for Molecular Dynamic Simulation", Computer Physics Comunications, **69** (1992) 317.

[8] D.C.Rapaport, "The Event Scheduling Problem in Molecular Dynamics Simulation", Journal of Computational Physics, **34** (1980) 184.