Introduction to a warehouse Visual simulator

J. M. Feliz Teixeira
António E. S. Carvalho Brito

LASSIP – Simulation Laboratory for Production Systems
GEIN – Section of Management and Industrial Engineering
Department of Mechanic Engineering and Industrial Management.
Faculty of Engineering of University of Porto
PORTUGAL

November 1997

ABSTRACT

Here we present an introduction to the work we have developed with the mainly intent of conceive a *Visual and Interactive Simulator Modeller* for warehouse simulation.

In the approach presented here, the different elements in a warehouse, as well as the processes they are related with, were based on warehouse systems and control systems the Portuguese company EFACEC S.A. uses in its process of building warehouse systems.

In this paper an initial reference is made to the simulation paradigm chosen *(discrete simulation)* and also to the method used for representing the flow of time (*next event)*.

The chosen operating system was the *Windows95/NT* and *Microsoft Visual C++* the programming language. So, the conception of the modeller has been developed on an *Objected Oriented Programming* basis (OOP), where there have been included a User Graphic Interface to permit an easy way of drawing and configuring the layouts.
This work also establishes the basis of a more general conceptual modelling using *Object Oriented Simulation* with the intent to be used as a good simulation start point for other future modellers. In fact, from this work resulted a simulation structure that can be easy readapted to a large number of different simulation cases.

1. Introduction

Warehouse automated systems are nowadays fundamental equipment to ensure the modernisation of production and distribution centres[i]. That is a rule for improving their flexibility and also to rise their processing capabilities in order to answer with more efficiency to the actual market demand.

The importance of simulation modelling in this field of industry is from all recognised, anyhow, as the commercial software modellers seem to be or too simple to permit a reasonable modelling of the real system, or too complex to consent to build it in a proper time, for these two main reasons only few Portuguese enterprises stand making some efforts in the field of simulation.

Nevertheless, with the strong evolution of programming languages in the direction of *Object Oriented Paradigm*, given its decrease of price and its improving flexibility, and also due to the excellent tools to build the user interface, simulation starts to become practicable even in those cases which imply the development of a particular solution.

In this work a warehouse is seen from a hierarchic point of view, which permits a more realistic separation of the responsibilities of its behaviour. It is a modular approach, where each entity is responsible for its own integrity and functionality, and where the decision rules are separated on three fundamental levels: *element level*, *control level* and *Management level*. In this perspective the other components of the simulation are just reduced to methods of material handling between *elements* and to some decision rules that can be implemented in some more or less complex algorithms.
With this work we intended to restructure certain concepts applied to warehouse simulation, and that those concepts could lead us to finally develop a warehouse object oriented simulator software.

Event simulation

As an brief introduction to the simulation methods used on this work it is important to refer we have chosen the *discrete simulation*, what means we are interested on representing the state of the system only in certain instants of the time. This method, when can be applied, usually conducts to a less complex formulation of the problem compared with, for example, continuous simulation techniques. However, there exist two different methods in discrete simulation to represent the time flow on the system (or the flow of the independent variable) (Pidd[ii]):

- Time Slicing, where the states of the system are sampled with a certain fixed frequency.

- Next Event, where the time advances to the next instant where there will be a changing in the state of the system.

Usually the Next Event technique is more efficient, as the sample frequency naturally adapts the way the states of the system change, what means one only have to analyse it in some relevant instants. The amount of calculations tends also to reduce in this kind of approach, implying fewer sources for accumulating errors.

The choice of this technique was due to the observation of the behaviour of warehouse systems, where processes develop in a discrete manner. By the other hand, this technique was early proposed also due to the deep tradition on *next event simulation* of this laboratory[iii], where it have been used for one decade to simulate various practical cases.

1.1 Concepts of entity and event.

The overall state time flow of any real system can be considered as the summation of some partial states of the elements of that system, elements which here we call *entities*. In that sense, *entities* can be, for example, the workers and the machines in a fabric, the clients and the staff personal in a post office, and, in general every elements that are able to show any state changes.

On the other hand, it is considered to happen an *event* in the instant of time an entity of the system changes it state. An *event* must then be responsible for starting a certain sequence of actions in the system, conducting or not to new events scheduled for the future. To this sequence of actions started by the event we call the *method* of that particular event.

In next event simulation literature one can easily find some didactic examples, as the well known philosophers case or the case of the haircutter. These cases are used to introduce the reader the basic concepts of simulation and also how to manage waiting lists. However, in this paper we have adopted another simple example for didactic purposes, which we have called "*the bus case*".

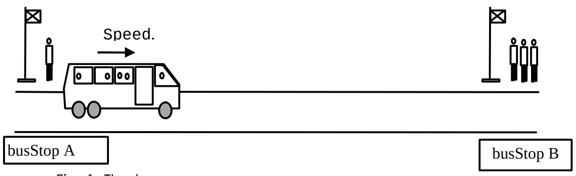

Fig. 1 The bus case

For now, lets suppose a bus the unique *entity* in the system. If at this moment we don't worry about the number of persons inside and waiting for the bus, lets consider the bus objective to go from bus-stop **A** to the bus-stop **B**. In this very simple example the bus can be found in one of the following two states: in march (we will call this state `MARCH`) while going from one stop to the other, and stopped (state `STOPPED`) while it is stopped near one of the bus-stops waiting for the clients to get out and to get in. The choice of these two states imply we must associate to the bus *entity* two different *events*: one to represent the instant of time the bus changes from `STOPPED` to `MARCH` (lets name it `EVENT_START`), and another to represent the change from `MARCH` to `STOPPED` (lets name it `EVENT_END`).

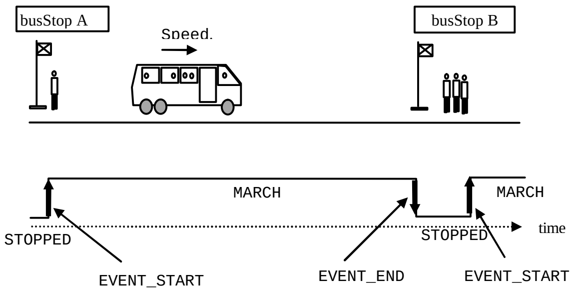

Fig. 2 The bus case with states and events

Each time an *event* happens it starts a group of actions, which defines the activity that will stand in the system till the time of the next *event*. This activity is defined in the *method* related with that *event*. This is the reason for associating to each *event* a *method* where it will stand a group of actions related with the required activity in the system.

Therefore, in the previous "bus case" the simulation will be resumed to the implementation of the following two methods:

- *Method* for the `EVENT_START` of the Bus *entity*:

```
Bus::Start()
{
    ◊ Set the state of the bus in MARCH.
    ◊ Knowing the speed of the bus and the
      distance to the next BusStop, calculate the
      time to reach the next BusStop.
    ◊ Schedule to that time the arrival of the
      EVENT_END.
}
```

- *Method* for the `EVENT_END` of the Bus *entity*:

```
Bus::End()
{
    ◊ Set the state of the bus in STOPPED.
    ◊ Calculate the time the bus will be stopped in
      the BusStop (dt).
    ◊ If it is to continue, schedule another event
      EVENT_START for the actualTime+dt.
}
```

In this simple case the activity of the system is reduced to the activity of a single entity: the bus. Anyhow, in practical systems one usually has to deal with many related entities, therefore the complexity of the model rises.

In general, before being executed, a event is first lined into the simulation *Event List* by means of a programming code method that receives the event and puts it in the proper place in that list ordered by increasing time. Here such method was named *Schedule()*. As we decided to identify an event with three parameters (the time when it is to occur, the identity of the event, and the entity responsible for executing the event), the *Event List* had to be considered as a list of objects where this information could be maintained. That was the reason for the creation of an elemental object that was named *SimTimeCell:*

```
class SimTimeCell : public CObject
{
 public:
    float        m_time;//event time
    UINT         m_event;//event number
    SimEntity*   m_pEntity;//entity to execute the event
};
```

The schedule of an event in the *Event List* is the responsibility of the *Schedule()* method, which will also order this list by increasing time and, in the case of events scheduled for the same time, by priority of execution. This priority was defined by the event identification number, a positive integer, where *zero* (0) represents the maximum possible priority. Any entity can use the *Schedule()* method to schedule events. The following example,

```
Schedule(t = 10, EVENT_START, pBus);
```

will result in the schedule of the event `EVENT_START` for time 10 associated with the entity `pBus`.

Once the event is scheduled in the *Event List*, its execution will be started by another method of the simulation named *Clock()*. This name was chosen for this method due to its responsibility of controlling the simulation time. This method removes the first event from the *Event List* and "sends" it to the associated entity to be executed. *Clock()* is then a loop on the simulator program code that will run while the *Event List* is not empty.

```
BOOL Simulator::Clock()
{
 SimClockCell* simClock;
 UINT evento;

 BOOL ret=FALSE;

 if(!m_eventList->IsEmpty())
   {
        ret=TRUE;

        //Remove the TOP event from the event list
        simClock = m_eventList->RemoveHead();

        //set the simulation time the same as the event execution time
        m_time = simClock->m_time;

        //"send"  the event to the related entity (to execute it)
        evento = simClock->m_event;

        simClock->m_pEntity->m_pSim = this;
        simClock->m_pEntity->Executa(evento);

        //Free memory
        delete simClock;
   }

 return ret;
}
```

2. Items in an automated warehouse

The components of a warehouse are in this approach grouped in *physical items*, that is the items occupying physical space in the warehouse, and *conceptual items*, those that represent conceptual forms of organisation, as *Jobs*, *Out Orders*, etc.

The *physical items* are grouped on:

- *Static items*
- *Dynamic items*

In the *Static items* one can include, for example, the reception and despatch zones of the warehouse, the paths for vehicle moving, the raking zones, the racking cells, etc. Belonging to the *dynamic items* group one considers the conveyors, the transfer tables, the workers, and all the kind of vehicles responsible for the movement of the material inside the warehouse. In general, the *static items* act as elements where the material stands, while the *dynamic items* are responsible for the transfer of the material between those *static* elements. In this approach, the warehouse can be thought as a black box receiving input and output tasks which starts an internal process of reorganisation involving the *physical* as well as the *conceptual items*.

3. Simulator structure

The *Simulator* object is the responsible for all the simulation process and was designed in accordance with the control and management informatic systems used by the company EFACEC on its warehouse systems. This idea was leading to the parallelism of concepts between those informatic solutions and the way the *Simulator* works, letting the user manage the simulation almost as he would manage the warehouse system.

Concerning the decision logic, we considered the following three distinct levels in the warehouse (fig.3): *element logic*, *control logic* and *management logic.*

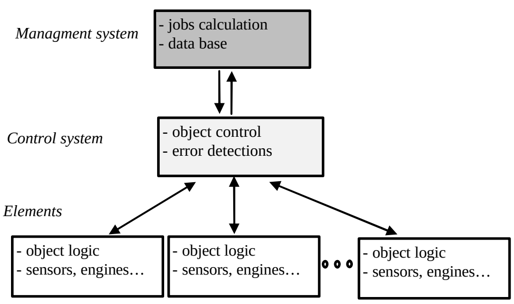

Fig. 3 Logic levels in a warehouse.

At the *elements* level the logic corresponds to the individual mechanism of the elements in the warehouse, that is, the way they work when apart from the system. The middle level, named *control system*, is related with the interaction with those elements, and deals with the criteria for exchange information between them. At the top level, named *Management system*, there is the logic related with the communication with the outside world, as well as the criteria to start jobs and also to access the warehouse model database.

Based on this idea the *Simulator* have been developed with a similar program code structure with the following correspondent levels (fig.4):

• ***SimWindow***, represents the physical space of the warehouse, where one can find the information about the conveyors, racking, aisles, vehicles, etc.,

• ***SimControl***, which represents the logic of the *Control system*,

• ***SimMaster***, which corresponds to the *Management system*. This block is responsible for the maintenance of the products information, output orders, input orders, and for the scheduling of the jobs of *dynamic elements* of the warehouse.

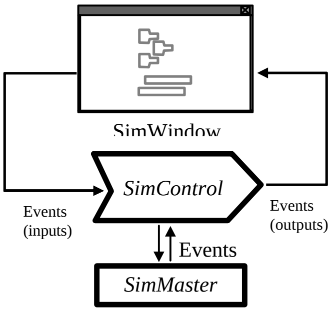

Fig. 4 Simulator structure.

3.1 Within the object *Simulator*

The *Simulator* is the object that controls the simulation, maintains the coherence of the time and "sends" to the entities the events to be executed. It is the *Simulator* the object that owns the reference for all the object lists on the system, those created by the operator before the simulation start as well as those created during run-time. It is also the owner of the simulation *Event list* and the methods concerned with its usage: the *Schedule()* to schedule the events, and the *Clock()* as the main simulation cycle.

One can say the *Simulator* has a "view" over the warehouse previously created by the operator, on which it executes the actions related with the events, then inducing state changes in the entities of the system, what leads to new orders and jobs appearing in the system, new spatial organisation for the material, new movements of the conveyors and vehicles…

The *Simulator* also owns the *SimControl()* and *SimMaster()* methods, which are deeply related with the *control system* and the *management system* of the warehouse, respectively.

3.1.1 The *SimControl()*

This method is used by certain objects as the decision maker in certain specific circumstances, for example, by a conveyor in the instant the palette reaches one of its endpoints. One can say the *SimControl()* is a control block where there can be included the criteria usually implemented in the "Programming Logic Controllers" (PLCs) on a real installation. This means the logic related with control operations can be concentrated in one particular block of the simulator, making easy its access to any future criteria changes.

In figure 5 there is an example of the interaction between a conveyor and the *SimControl()* when a palette reaches the endpoint of the conveyor `conv1`. This conveyor sends the *SimControl()* the event END and the *SimControl()* then decides what to do.

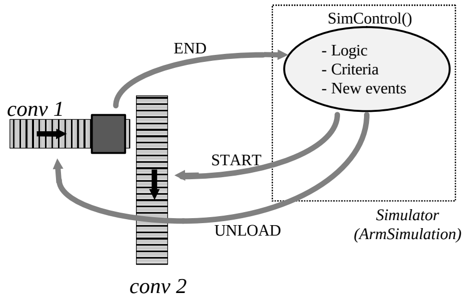

Fig. 5 Interaction of events with the *SimControl()*.

In the present case the *SimControl()* gives the conveyor `conv2` the order to start by sending it the event START, and then orders `conv1` to unload the palette by sending it the event UNLOAD.

This mechanism of control leads to a better organisation of the simulator, separating the program code by different levels of decisions, what will allow the change of control criteria without the need to change the conveyor's (or other element's) model code.

The next C++ code represents the basic structure of *SimControl()* where have been implemented the decision rule for when a palette reaches the end of a conveyor.

```
void Simulator::SimControl(time,event,pEnt)
float time;
UINT event;
SimEntity* pEnt;
{
  //if the entity is a conveyor:

  if(pEnt->m_tipo == CONVEYOR)
    {
     Conveyor* pconv = (Conveyor*)pEnt;
     switch(event)
     {
      case END:
      /////////////////////////////////////////////
      //  Rule to follow when the palette reaches the end:
      //  If exists an entity for the conveyor to unload and if
      //  that entity is ready to receive the palette, then send the
      //  conveyor the order to Unload(), else, stop it and put it in the
      //  state WAITING.
      //  Note: the unload can be to another conveyor or to a table.
      //  If the unload is to a conveyor it is only necessary that this
      //  conveyor is stopped and with no other palettes on it.
      //  The unload to a table it is considered at anytime possible.
      //  If the conveyor cannot unload, it will be stopped and putted
      //  on the object waiting for unload list…
      /////////////////////////////////////////////
      float dt = pconv->m_tUnload;
      SimEntity* pUnload = pconv->m_pEntUnload;

      if(pUnload!= NULL)
      {
       //Unload to a conveyor:
       if(pUnload->m_tipo== CONVEYOR)
       {
        if(pUnload->m_state==STOPPED) //unload
        m_pSim->Schedule(time+dt,UNLOAD,pconv);

         else //Wait
           {
            pconv->m_state = WAITING;
            m_objWaitList.AddTail(pconv);
```

```
            POSITION pos;
            pos = m_objMovingList.Find(pconv);
            if(pos!=NULL)
                m_objMovingList.RemoveAt(pos);
          }
      }
      //Unload to a table:
      else
       m_pSim->Schedule(time+dt,UNLOAD,pconv);
     }
    //there is no unload entity. Let fall down the palette…
   else
      m_pSim->Schedule(time+dt,UNLOAD,pconv);

    break;

    default:break;
   }
  }//the entity was a conveyor…
}
```

3.1.2 The *SimMaster()*

This method includes the *management system* decision logic of the warehouse. It is responsible for maintaining the product information data, for ensuring the job distribution in the warehouse model and the for the good communication with the "surrounding world". It is this method the responsible for the interpretation of the input and output orders, for the choose of the cells involved in the flow of the material inside the warehouse, and for the scheduling of the jobs of the dynamic elements who will move the material from one place to another. Similarly to the previous case of *SimControl()*, this block of code *SimMaster()* plays the rule of a main decision management centre which objective is to handle the high level operations on the model.

As the main "manager" of the warehouse, the *SimMaster()* is then the owner of all the object lists taking part on the simulation process, by which it keeps a "view" from the overall model.

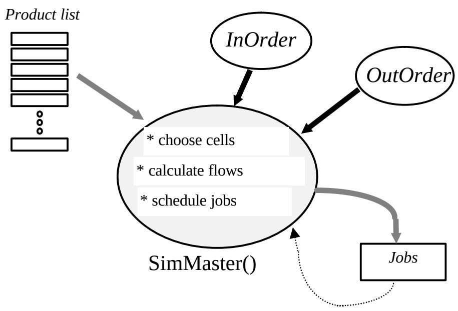

Fig. 6 Basic diagram of the *SimMaster() activity*.

Anyway, one of the most important tasks of the *SimMaster()* is to process the *input orders* as well as the *output orders* by means of analysing the list of products, the list of cells and its states, and the possible ways to take the material to its destiny. At this point it creates a new *job* to be given to the first *dynamic element* free (fig. 4.4).

As the decision criteria are implemented in this block of the simulator, certain objects send events to *SimMaster()*, then transferring the responsibility of the decisions to this high level logic processing "centre". That is the case of the transference of a *job* between different *dynamic elements*: each element is responsible for the execution of its part of the *job* and then it return the *job* to *SimMaster()* for that it can be given to the next *dynamic element*. The next figure represents one of these processes, in the case of moving a palette from an origin cell to a destiny cell involving vehicles and conveyors.

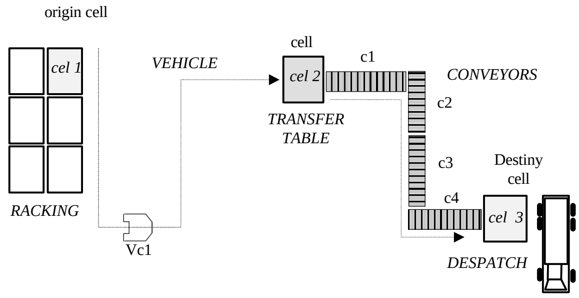

Fig. 7 Example of a simple output process.

In this example, first *SimMaster()* creates the following *job* that will be given to the vehicle `vc1`, as this is the *dynamic element* with access to the initial cell `cel1`.

Type of SubJob	*Number of unites*	*Type of unite*	*Product of the unite*	*Cell*	
LOAD	1	PALETTE	P1	**Cel1**	
UNLOAD	1	PALETTE	P1	**Cel2**	**END**
LOAD	1	PALETTE	P1	**Cel2**	
UNLOAD	1	PALETTE	P1	**Cel3**	

Once having the *job*, this vehicle starts to move in order to access the point of cell `cel1` where it will then load (LOAD) a palette containing the product P1. Finished the first line of the *job*, it will start to execute the next line related with the unload: it moves to the cell `cel2` where the palette will be unloaded (UNLOAD). Once executed the unloaded, the vehicle deletes its lines in the *job* and sends it back to the *SimMaster()* in the following form:

Type of SubJob	*Number of unites*	*Type of unite*	*Product of the unite*	*Cell*	
UNLOAD	1	PALETE	P1	**Cel2**	**UNLOAD (Done)**
LOAD	1	PALETE	P1	**Cel2**	**UNLOAD (end)**
UNLOAD	1	PALETE	P1	**Cel3**	

When the *SimMaster()* receives back this *job* it will know the palette have already been unloaded in the cell `cel2`. As the *job* is not yet complete, *SimMaster()* will try then to find the next entity to whom the job can be given, that is, the entity who will be able to load the palette from cell `cel2` and take it to and unload it in cell `cel3`. In this process, *SimMaster()* have to analyse the possible paths of transport defined in the warehouse and then decide which path is to be chosen based on the decision criteria implemented. In this particular case *SimMaster()* will decide to give the remaining *job* to the conveyor `c1` after deleting the UNLOAD line on the head of the *job* subJob list. The remaining *job* will then be:

Type of subJob	*Number of unites*	*Type of unite*	*Product of the unite*	*Cell*	
LOAD	1	PALETE	P1	**Cel2**	**UNLOAD (end)**
UNLOAD	1	PALETE	P1	**Cel3**	

So, this conveyor loads the palette from cell `cel2` and moves it till reaching the conveyor `c2`. Then, the palette and the *job* are transferred to the next conveyor, who will be now responsible to move the palette till reaching the conveyor `c3`. Again the process repeats, till the palette and the *job* is transferred to the last conveyor `c4`. Finally, conveyor `c4` will move the palette till reaching cell `cel3` where it will execute the unload of the material. As there are no more lines to execute in the *job*, the *job* will be considered finished and the *SimMaster()* then notified by the conveyor `c4`.

The next C++ code represents the main structure of the *SimMaster()*, where one can easily understand the way the events are passed to this method.

```
void Simulator::SimMaster(time,event,pEnt)
float time;
UINT event;
SimEntity* pEnt;
{
/***************************************************
/*   If the entity who sends the event is a VEHICLE:
***************************************************/
 if(pEnt->m_tipo == VEHICLE)
   {
    Vehicle* pveic = (Vehicle*)pEnt;
    switch(event)
    {
     case JOB_END:
     //////////////////////////////////////////////
     // Here it would appear the code to run when a vehicle gets FREE
     //////////////////////////////////////////////
                 . . .
     break;

     default:break;
     }
   }

/***************************************************
/*   If the entity who sends the event is a conveyor:
***************************************************/
 else if(pEnt->m_tipo == CONVEYOR)
   {
    Conveyor* pconv = (Conveyor*)pEnt;
    switch(event)
    {
     case JOB_END:

     //////////////////////////////////////////////
     /
     // Here it would appear the code to run when a conveyor ends the
     JOB
     //////////////////////////////////////////////
     /
                 . . .

     break;

     default:break;
    }
   }

/***************************************************
/*   If the entity who sends the event is another kind of entity….
***************************************************/
                 ...
}
```

3.1.3 The *SimMovie* (movement effects on simulation)

Certain entities on the simulation use the effect of movement to give the user a more realistic idea of the warehouse dynamics. That is the case of palette movement on the conveyors and the movement of the vehicles during the simulation run. All these "realistic" effects are sustained by the action of a new object in the simulator named *SimMovie*. Anyhow, the action of this object can be turned on or off, depending if the user wants the simulator to reproduce the visual movement of the dynamic entities on the computer display or not. So, this facility can always be turned off in order to speed up the simulation run. The object *SimMovie* is then an object reserved to display those movement effects that turn the visual simulation more closed to the behaviour of the real system. In order to make possible the action of *SimMovie* another event was defined in the system, named SIM_SHOW. If the operator decided to create the *SimMovie* object in the simulation, then this mechanism is activated, otherwise it stays disabled and no movement will be observed in the display. The structure of this object is the following:

```
class SimMovie : public SimEntity
{
 public:
   //Constructor:
   SimMovie(Simulator* pSim);

   //Event router:
   virtual BOOL Executa(UINT event);

   //Handler to the event SIM_SHOW:
   BOOL ShowON();
};
```

This object inherits from the base class *SimEntity* that will be presented soon on this paper. As already have been explained, the class *SimMovie* is responsible to handle the event SIM_SHOW routed to its method *ShowON()*. During creation, a pointer to the *Simulator* object is passed to *SimMovie*, through which it will have directly access to the list of objects moving in the simulation.

In the next figure it is represented a diagram of the mechanism which ensures the visual movement of the *dynamic elements* during the simulation run. This mechanism is enabled or disabled in the moment of starting the simulation process by creating or not an object *SimMovie*.

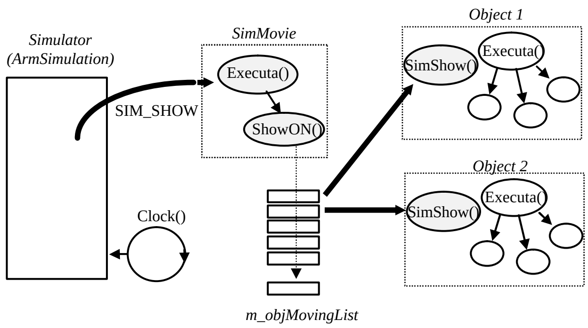

Fig. 8 Simulator mechanism for object moving in the display.

In this mechanism, when in the simulation *Clock() appears* the event SIM_SHOW, the *Simulador* sends it to the *SimMovie* object by calling its member function *Executa()*. This member function then calls its method responsible for handling the event SIM_SHOW, that is, the method named *ShowON()*. The activity of this method resumes to the call of the function *SimShow()* for each object in the simulator *object moving list* (`m_objMovingList`), which is the function responsible for the actualisation of the object on the display.

After, *ShowON()* calculates the next instant of time when is due the next display and schedules a new event SIM_SHOW to be executed in that moment of the future. The method *SimShow()* is presented in the following lines of code.

```
BOOL SimMovie::ShowON()
{
 SimEntity* pObj;

 //////////////////////////////////////////////////
 // Forces the representation on the display the moving objects…
 //////////////////////////////////////////////////
 POSITION pos;
 pos = m_pSim->m_objMovingList.GetHeadPosition();
 while(pos != NULL)
      {
       pObj = m_pSim->m_objMovingList.GetNext(pos);
       pObj->SimShow();
      }

 //////////////////////////////
 // schedule the next event SIM_SHOW
 //////////////////////////////
 float DT = m_pSim->m_DTime;
 m_pSim->Schedule(m_pSim->m_time+DT,SIM_SHOW,this);

 return TRUE;
}
```

4. Passive entities and active entities

During the simulation run one can consider two kind of entities running in the system: passive entities and active entities. Passive entities are considered those entities that are not responsible for the generation of any events, even if they can change their state, while active entities are those who plays an active role on event generation. A table, for instance, have been considered a passive entity, as it is viewed as a static element where the palettes can stay, while a conveyor or a vehicle have been thought as active entities, as they are responsible for the flow of the material along the warehouse, and then for the changing of the state of other related entities.

This section describes the mechanism by which the active entities on the system process their own events by inheriting from the base class *SimEntity*.

4.1 *Event* execution and class *SimEntity*

In this paper it was already described how to schedule events into the *event list* and how the simulator uses the method *Clock()* to remove them from that list, turning them executable. Nevertheless, nothing yet have been said about the mechanism subjacent to the event execution.

In such mechanism, when a certain event is to be processed by an object, the *Simulator* starts to call the method *Executa()* for that object, defined as virtual in the *SimEntity* class. In fact, this is the reason every active element on the simulation has to inherit from the base class *SimEntity*. Then, the execution of the event is started by the following statement:

```
m_pEntity->Executa(m_event);
```

which can be understood as “sending” the event `m_event` to the entity `m_pEntity`. Once the entity “receives” the event, the entity will search in its handle methods list the one designed to answer to that particular event, then calling it.

As an example, lets again think on the “Bus” case and in particular on “sending” the BUS entity the event `EVENT_END`. Let also assume there would be three events associated with the bus, named `EVENT_START`, `EVENT_WAIT` and `EVENT_END`, and then three handler methods named respectively *Start()*, *Wait()* and *End()*. When the *Simulator* retrieves from the *event list* the following *time cell* related with the event `EVENT_END`:

t = 10	EVENT_END	pBus

Then it simply sends the bus the event by executing the statement:

```
pBus->Executa(EVENT_END);
```

At this point the bus already “knows” which event is to process, and then it can choose the correct handling method defined and implemented on its object class. In this particular case the *End()* method would be chosen and the event finally processed. For all what was said here the virtual method *Executa()* defined in *SimEntity* class is considered as an event router (fig.9).

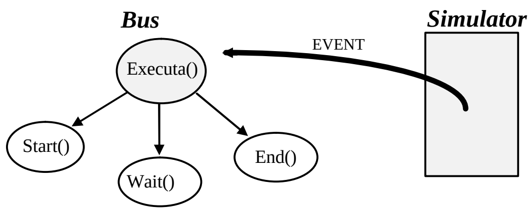

Fig. 9 Mechanism for executing events .

This mechanism was early proposed by Dirk Bolier [iv] in a publication concerning simulation libraries developed in C++, and it was adopted here due to its simplicity and practical efficiency. The method *Executa()* must have been declared public in the class definition of the object who will use it, in order to give the *Simulator* the possibility to access it. The remaining handler methods can most of the times be declared as private to that object.

As it is clear now the *SimEntity* class has the main objective of defining the basic functionality of any active entity in the simulation. That is why *Conveyor* objects as well *Vehicle* objects are made to inherit from *SimEntity*.

Nevertheless, in the class *SimEntity* have been included other kind of attributes useful to be used by any active elements on the simulation, as well as the definition of the method *SimShow()* related with the dynamic display of the object during the simulation run.

It is the following the *SimEntity* class structure:

```
class SimEntity : public CObject
{
 public:
   BOOL          m_livre; //generic use
   CString       m_nome;  //entity's name
   float         m_time;  //entity's time on the simulation
   UINT          m_estado;//entity's state
   Simulator*    m_pSim;  //pointer to the Simulator
   UINT          m_tipo;  //entity's type
   float         m_x;     //entity's x position
   float         m_y;     //entity's y position


   SimEntity(CString nome); //public constructor
   virtual void Serialize(CArchive& ar); //Serializer

 protected:
   SimEntity(); //private constructor

 public: //SIMULATION RELATED METHODS:
   virtual BOOL Executa(UINT event);//event router
   virtual void SimShow(); //movement display

  DECLARE_SERIAL(SimEntity)
};
```

5. Conclusions

This work leaded to the implementation of a visual warehouse simulator with which some results have been achieved mainly on simulating practical layouts developed by the firm EFACEC, what have shown the versatility and the easy use of the concepts presented here. There have been developed several layouts based on conveyor transportation and distribution of the material and also the action of vehicles with predefined paths interacting with palette racking systems.

The idea of separating the responsibility of decision criteria in the three referred levels (elements, control and "manager") have succeed and let us look at the warehouse as a modular and easy to modify system. The design of the entities was then thought as independent of the system as a whole, and so maintaining their functionality even isolated from the warehouse. This made possible to achieve a good degree of modularity as well as comfortable code portability.

In a certain point of view this approach lets us feel the *Simulator* more representative of the reality, as the events become property of the objects instead of considering the *Simulator* their owner. Also the hierarchy on the decision tasks lets us thought as managing a real warehouse system.

In particular concerning the *physical elements*, that is, conveyors, racking blocks, vehicles, transfer tables, etc., this work played an important role to start the development of future elements, in order to give the user the ability to access and choose objects from a list of standard warehouse equipment.

What concerns to the processing speed of this *Simulator* it was observed a practical value of ten (10) times faster than the real time flow, even if the implementation of the "three levels decision logic" early described could seem less efficient than if there would be no re-routing of the events between the objects and the *SimControl()* and *SimMaster()*. Anyhow, once nowadays computer systems performances are high and tend to increase, this question doesn't have to be considered relevant for typical problems of warehouse modelling.

References

[i] António E.S. Carvalho Brito,"*The Use of CAD Techniques in Configuring Visual Interactive Simulation Models: A New Approach for Warehouse Design*", Ph.D. Thesis, Cranfield Institute of Technology, U.K., 1992.

[ii] Michael Pidd, "*Computer Simulation in Management Science*", John Wiley & Sons Ltd, 1992

[iii] António E.S. Carvalho Brito,"*The Use of Computer Aided Design Techniques in Configuring Visual Interactive Simulation Models for Warehouse Design*", Journal of Decision Systems, Volume 1 - nº 2-3, Hermès, Paris, 1992.

[iv] Dirk Bolier, Anton Eliens, "*SIM : a C++ library for Discrete Event Simulation*", Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam. October 1995.

Bibliography

- Michael Pidd, "*Computer Modelling for Discrete Simulation*", Michael Pidd, John Wiley & Sons, 1989.
- Osman Balci, et al., "*Developing a Library of Reusable Model Components by Using the Visual Simulation Environment*", Arca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., July 1997.
- Osman Balci, et al., "*The Visual Simulation Environment*", Arca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., June 1997.
- Robert E. Shannon, "*Systems Simulation, the art and the science*", Prentice-Hall, New Jersey, 1975.
- "*Visual C++ Books Online, C++ Language Reference*", Microsoft Corporation, 1992-1995.