INTERACTIVE BEHAVIORAL PROTOTYPING USING VIRTUAL REALITY TOOLKIT

F. Harrouet, P. Reignier, J. Tisseau

Laboratoire d'Informatique Industrielle, Ecole Nationale d'Ingénieurs de Brest
[harrouet, reignier, tisseau]@enib.fr

Abstract. Mobile robotic is a complex domain. Building a robot is not only a question of tuning a set of parameters on an algorithm. New algorithms have to be designed for each new problem and tools are needed to help in this process. This paper introduces a Virtual Reality platform based on a dynamic multi-agent programming language. It has been designed to show that simulating a Multi-Agent System in a virtual environment with dynamic properties can be used for Interactive Prototyping. After making out a list of requirements to achieve this goal, we present our tool. Then, an example shows, through a very simple application, what an Interactive Prototyping session looks like.

1 Introduction

Mobile robotic is a complex domain. Building a robot is not only a question of tuning a set parameters on a program. No "universal" solution exists by now and for each new application, a new algorithm has to be designed or an existing one has to be adapted. Tools are needed for easily create, manipulate and test those algorithms. And even if simulation will never replace real tests, it can accelerate the development process.

The aim of this paper is to point out the fact that a multi-agent virtual reality environment, as performed with our oRis/ARéVi platform, can be run as a tool for dynamic/interactive prototyping complex systems as robots' controllers.

The classical spiral, very often used to build a prototype of a system to be designed, consists in the following loop:

1. to generate the prototype,
2. to test the prototype,
3. to adjust the prototype according to the test's results,
4. to go to step 2 while needed.

This approach is quite efficient in many cases where the described system can be seen as a whole corresponding to one global model. However, when the system becomes complex (i.e.

http://rcin.org.pl

made up of multiple parts in interaction), this global model is much harder to design and to tune, due to the fact that every possible interaction cannot be listed in an exhaustive way. According to this remark, our main goal is to introduce a tool which allows to handle complex systems to dynamically and interactively model their behavior.

If the simulated system is complex, then its global behavior cannot be fully described. Therefore, it would be easier to try to give its components the best behavior when peculiar situations happen to them. This can be done only when the simulation of the system is running because we are not able to know in advance every possible configuration of the system. Thus, the prototyping loop can disappear and **the user and the designer of the system have to be present inside the simulated system**. The use of a tool allowing to perform this kind of prototyping should consist in:

- describing an initial state of the system to work on as a set of components with their own behaviors,
- letting the tool simulate the system's global behavior which results from interactions between components,
- allowing the user to modify and mend behaviors and structures **during run-time**.

In section 2, we will describe what we exactly define as Interactive Prototyping. The functionalities required by a tool to achieve this goal will be presented section 3. Then, a possible solution, illustrated by an example, will be shown section 4.

2 Interactive Prototyping and Virtual Reality

Nowadays, Virtual Reality is a term which is very commonly used, in many contexts and sometimes with different meanings. We will try here to find out the main meaning of this term that suits to Interactive Prototyping. This will be done considering first the animation meaning, then the simulation aspect, and finally the interactive point of view.

http://rcin.org.pl

2.1 3D animation

Virtual Reality is usually seen as a graphic animation in a 3D world. In this case, people only have to watch the animation but can neither do anything else not interact with it. Sometimes, the user can travel in the virtual world, choosing its own way to look at different parts of the system under many points of view. In this case, the animation is qualified as *"walk through animation"*.

The quality of such a tool can be resumed to the quality of the rendering (3D images on large screens or head mounted displays, 3D sounds, . . .). The user can only travel and look around. Therefore, this is not interactive enough to be run for *Interactive Prototyping*. Nevertheless, this kind of tool can be useful for architectural and artistic designing [1, 2].

2.2 Interactive simulation

Virtual Reality can also be seen as a tool for Interactive Simulation. It means that in the 3D world, the user will not only be able to move but also to interact with the components of the represented system according to pre-defined parameters. Thus, in this situation, the only way to act on the animation, is to change the values of these parameters and nothing else.

The designer of such an animation has to think about which parameters will be accessible to the user. Then, he must give the user a way to control them. A very common way to control these parameters is to use a classical graphic user interface (control panels made up of buttons, switches, sliders, . . .), but in 3D worlds we often use specialized devices to control geometric parameters (joysticks, 3D mice, data gloves, trackers, force feedback actuators, . . .). The more accurate and easy to use these devices are, the better the quality of this tool is. This kind of animation can be useful to install premises [3], to train people to work in a specific environment (SOFI project [1]), or to work in teleoperation [4]. However, the only changes to be allowed must be predefined by the designer before the user acts, and consequently, they are very often structural (as moving objects).

http://rcin.org.pl

2.3 Interactive prototyping

We try here to get rid of the main limitation of the above kind of tool: a restricted expression of changes, due to the fact that the designer should think about every possible change the user could try. Since prototyping represents the action of designing, building the model of the system, the previous tools cannot be run for this. They can be used to test the prototype in several predefined situations, but they do not allow to build such a prototype. The de-signer builds the system, the user tests it and the designer re-builds the system until the user is satisfied of his tests' results. This approach can be described as *"prototyping"* but cannot be qualified as *"interactive"* (it corresponds to the prototyping spiral seen section 1).

Then, if we want to have an *Interactive Prototyping* process, we must give the user the opportunity to substantially change the behavior of the system's components whenever and however he wants. This can be done in order to try many behaviors against each other, to simulate defaults or mistakes in a process, to watch how the system reacts to an unknown situation, to simulate the reconfiguration of an industrial process, and many operations which cannot be done on a pre-established simulation.

Therefore, if we want to give the user the full control of his experiments on the system, he has to be able to use the same expressions and ways to act than those used by the components in the system. This means that if **a language is used by the designer** to describe the system's components and their behavior, the user has to use **the same language during the experiment**. In this case, more than testing the component's behavior, he will be able to tune or to mend them in situation**, while running them**.

Thus, we can say that the user is in *"immersion through the language"* so that there is no longer any limit between him and the designer of the simulated system. This is the main meaning we give to *Interactive Prototyping*, and that's the way we consider *Virtual Reality* too (as much as being in sensitive immersion). According to our own opinion, the three main aspects of *Virtual Reality* can be summarized as follows:

http://rcin.org.pl

Animation	→	"what the model **is**"
Simulation	→	"what the model **does**"
Interactive Prototyping	→	"what the model **becomes**"

Further explanations on our systemic approach of Virtual Reality and Interactive Prototyping are presented in [5].

2.4 Multi-Agent systems

This kind of system is a set of components which are not only static objects but which also have a dynamic behavior. More than just looking like real world's objects, the virtual components must behave as real ones.

In this kind of virtual world, the designer cannot know in advance how the system will change during the simulation. The global evolution of the system is the consequence of many interactions between its components. These ones react according to local information, so that it is very difficult to deduce a global result from so many local data. In this case, the only way to determine the global behavior of the represented system is to let it work.

Thus, the simulated system is a Multi-Agent System in which each component is an agent endowed with its own goal and behavior. It implies that the behavioral/dynamic aspect of the virtual components must be taken under consideration as much as their structural/static aspect. The designer of such a system cannot model it as a whole but has to describe each agent's behavior using the following sequence:

- perception, in the virtual environment (other agents or objects),
- decision-making, according to its own perception, state and goal,
- action, in the environment and on itself.

In such a tool, the user can have a very important role. He can introduce changes and make the simulated system go in a state which may never have been reached without its intervention. A

http://rcin.org.pl

possible way, for the user, to act in the virtual world, is to use an *"avatar"* able to perform several actions, and seen as an agent in the system. But more than that, the user must be able to dynamically change every part of the system by rewriting their behavior when needed.

3 Requirements for interactive prototyping

In the previous section, we have shown that the same formalism should be used by the designer to describe the multi-agent simulation and by the user to experiment and tune the system. Then, we will try to find out which properties should be found in a development environment to achieve this goal.

3.1. 3D Worlds and sensitive immersion

The main visible aspect of *Virtual Reality* concerns the sensitive immersion in a 3D environment. High-level concepts like 3D geometry, kinematics, cameras, scenes, viewers or manipulators can be found in 3D libraries such as *Open Inventor* [6], *OpenGL Optimizer* [7], *Fahrenheit* [8], *Ilog Vision* [9] . . . These libraries are useful to build a *Virtual Reality* tool but they do not offer sufficient abstractions to be used by an end user.

However, there are Virtual Reality platforms that can be run by end users such as *Virtual Builder II* [10], *Minimal Reality Toolkit* [11] and its Environment Manager [12], *Clovis* [13], *Virtual Design* [14], *Dvise* [15] or *Dive* [16].With these platforms, the concept of 3D autonomous object exists so that the end user, building a virtual world, can concentrate its work on the application more than on the technical/graphical aspects. A large range of specific devices are managed by these tools. Different kinds of behavior can be predefined, from simple callbacks to high-level intentions.

3.2. A dynamic programming language

In subsection 2.3, we have reached the conclusion that the user should be in *"immersion through the language"* and should describe the behavior of many components. Thus, we must use, to design the simulation, a language which is both object-oriented and based on run-time

http://rcin.org.pl

code rewriting. Nevertheless, for many languages the qualifier *"dynamic"* or *"dynamically extensible"* does not represent the point we are interested in.

Most of these languages, and especially Java, could imply that it is possible to rewrite the code of an application at run-time but it is far from granted. These languages allow to load and use extension libraries that may be compiled after the application was built. Thus, the main application must be designed to explicitly load such libraries. Moreover, these ones can only extend what was designed to be extended in the main application.

For example, in Java, it is possible to use a ClassLoader to load a compiled class that **must** extend a more abstract class managed by the application. If it does not extend a predefined class, then the application will only know that it is an Object and will not be able to use it very efficiently. By using ugly hacks, we can try to load many times a given class with several changes in its code. Then, different classes (with the same name) are obtained in the application **but it is impossible to change the code** of an existing class.

The C/C++ languages that are not qualified as *"dynamic"* allow exactly the same operation in an even easier way (using calls to dlopen(), dlsym() and dlclose() functions | see the UNIX man pages). Actually, you can design for example, a C++ application managing an abstract class Vehicle and let many other designers create many specific classes extending Vehicle (Car, Lorry, Truck, . . .) in dynamic libraries. The only thing to know about these libraries is that each one provides its own prede,ned function (say for example: extern "C" Vehicle * newVehicle(...);) that the main application can use to obtain a new instance of a specific vehicle. If a class is no longer used, its library can be unloaded, and eventually, reloaded after code modifications and a new compilation. As with Java, different classes with the same name can be loaded.

The main limitation to these extension libraries lies in the fact that it is only possible to load classes which inherit from the predefined classes of the application. They are explicitly loaded and used in specific points of the program. To dynamically introduce a totally different class,

http://rcin.org.pl

the main application should be modified to manage it. Thus, this kind of application can only be extended in the way the designer allowed it.

Interpreted languages such as Scheme or Prolog use similar representations for code and data so that the code might be changed at run-time as easily as data are. Therefore, this kind of languages can be used to design self-rewriting programs but the syntax and the style used in this case have nothing in common with *"classical"* programming. More than the fact that the programming style may be difficult to handle for a non-specialist user (in comparison with imperative languages), object-oriented extensions of these languages must be studied with a particular attention. Actually, these extensions can be implemented using classes (every instance of the same class share the same code) or as prototype languages (each instance has its own code), but it is much harder to use a mix of these two approaches. This is a very important point of our objective because we want the user to be able to change entire classes or only several instances during *Interactive Prototyping*. We call this property *"instance granularity"*. For us, this expression means that the most specific behaviors are not only described in very specific classes but can also be given to particular instances (which then differ from their original class). The reason for this choice is a simple projection of what we do in reality; when someone makes changes on his own car, he does not change every car of the same model. But afterwards, if the changes on an object seem to be better than the original version, it can be useful to generalize this new behavior to the entire class too. Thus, the prototyping language must offer an easy and clear way to allow changes in both cases.

3.3. Active objects, concurrency and scheduling

In subsection 2.4, we have declared that more than designing a global control for all the components in the simulation, it is better to let these components *"live"* and inter-act to produce themselves the global behavior of the simulated system. This means that each component should be given its own autonomy in the execution of its behavior. Thus, more than being passive objects whose methods are called from an external control, the components should be active objects, *"living"* in concurrency, and deciding on their own to trigger actions according to their local perception.

http://rcin.org.pl

This kind of execution could imply the use of object-oriented and multi-threaded programming languages such as C++ with Posix threads or Java. But this must be used with a lot of care. Actually, we are not performing a simple animation but a **simulation** of a system made up of many concurrent objects. In these circumstances, the tool used to activate the objects **must not introduce a bias** in the simulation. The sharing of the execution time has to be as fair as possible. In a discussion during the *SMAGET'98* meeting (CEMAGREF, October 1998, Clermont-Ferrand, France), Jacques Ferber made an oral intervention concerning this problem. He pointed out the fact that the designer of the simulation of a distributed system must not only describe the behavior of each component of the system but also has to exactly describe the way the scheduling is done in the simulation. This care would allow the designer or someone else to find out in the scheduler an artifact that could drive the simulation in a state which would have never been reached without it.

An illustration of this problem stands in the fact that the priority of Java threads is not interpreted in the same way under UNIX and under Windows. In the worst case, we are sure that a thread with a low priority will not be active as long as other threads with higher priorities can be activated. But in the second case, the priority of a thread only represents a probability to be activated. More over, the scheduling of Java threads uses a cooperative policy. This means that each thread is responsible for letting the others become active (using the yield() function). Then, a thread can keep the entire activity for itself. By using another ugly hack, it is possible to simulate a round-robin policy (a high priority thread sleeping during a few milliseconds in an infinite loop), but this solution is not very stable. Actually, in a small Java program using this hack with a lot of threads, it occasionally happens that the activity is given many consecutive times to the same thread, during a few seconds or more, and is then given to the other threads without any visible reason. Such a scheduling can be sufficient for parallel computing but must not be used to simulate distributed systems.

The problem of the scheduling policy is the same with Posix threads which are scheduled in a cooperative way too. The round-robin policy is only accessible for processes with the super-user privileges (due to the fact that threads' activity is managed with processes' activity - see the UNIX man pages). Therefore, it must be handled with care. Even if it is possible to

http://rcin.org.pl

simulate a round-robin policy (with a hack similar to the previous one), there is another scheduling problem due to the fact that Posix threads are stored in queues (First In { First Out). This means that in a set of same priority threads, these ones will always be activated in the same sequence (except for those which are blocked). This could be very detrimental to the fairness of the simulation. For example, if a set of active objects are regularly in competition for the same resources, those which have been created first have an advantage over the next ones because they always react before them. Therefore, with this sequence, the scheduler introduces an artificial and unwanted priority between the active objects that could bias the simulation.

However, the queues used to manage the threads ensure a very interesting property: we are sure that every thread of the same priority will be activated the same number of time (except for those which are blocked). A good solution could consist in a round-robin keeping this property while breaking the unwanted sequence. This could be done with a scheduling policy like on figure 1. This solution is quite simple but requires the building of an entirely new scheduler.

repeat forever:

- while the set of same priority threads is not empty:
- pick and extract at random a thread from this set
- activate it during a time slice
- put it back in a second set

swap the set of same priority threads with the second set

Fig.1. A fair round robin

Many parallel programming environments exist, allowing to build active object applications. These environments can consist in libraries (Poggi & Rimassa's library [17], *Para++* [18]) or in extensions of existing object-oriented languages (*sC++* [19], *Parsec* [20], *Jade* [21], *ICC++* [22]). Most of these environments are based on an event driven execution relying on synchronization primitives such as the "rendez-vous". They can be formally described (this is the case for *sC++* [23]) so that a theoretical study allows to prove some properties about the

http://rcin.org.pl

execution or allows to detect inconsistencies such as *"deadlock"*. But the entire model only relies on the synchronization primitives and no detail is given about the scheduling policy. Then, these environments are very useful for parallel computing but do not fit very well the simulation of distributed systems.

3.4. Agents and communication

On the assumption that an active object environment with the scheduling policy described figure 1 is available, we must now study how the interactions between these active objects can be performed. As seen section 1, the simulated system is made up of many components whose interactions determine the global behavior. Therefore, in this kind of application, the communication means have to be studied with a particular attention.

The most simple communication mode which can be used consists in synchronous method invocations, as usual in imperative object-oriented languages. This kind of communication implies that the object receiving the method call **must execute it**; the entire decision-making has been made by the caller. The caller waits until the method is ended (in fact, the time needed to execute the method is spent on the caller's activity). The main care to be taken consists in synchronization with critical sections around shared resources to ensure that many active objects can-not perform some antagonist actions at the same time. This is usually done with mutual exclusion semaphores (hidden behind the synchronized keyword in Java).

However, in Martin Carroll's article [24], the author ex-plains that it is very difficult to detect all the possible critical sections and then, in practice, a lot of them are very often forgotten. Thus, he introduces a pattern based on a mix between a synchronous method invocation and an asynchronous processing to design active objects. The synchronous method invocation only consists in putting a message in the receiver's mailbox. Then, the communication between active objects becomes asynchronous. This implies that the execution of the service is no longer performed by many callers but, from now on, is always per-formed by the receiver when checking its mailbox. Then, the service does not need to be synchronized (only one execution at the same time), only the mailbox management requires a synchronization. Therefore, according to this point of view, the asynchronous communication seems to be easy

http://rcin.org.pl

to use. Nevertheless, this implies that the active objects' behaviors must be written in a very different way. Actually, the result of a service call may be returned a long time after the invocation was done, so that the caller may wait for this result or may do something else while keeping an eye on the result delivery. Furthermore, each active object must manage the actions it decides to do on its own as well as those asked by the others (through its mailbox).

Despite the fact that the programming style is quite different from synchronous method invocations, however it allows to consider active objects as agents. The main difference between an object and an agent can be represented figure 2, which is directly inspired from Jacques Ferber's figure [25, 26]. We can see here that method invocations are only used inside the agent itself, and that the communication between agents is performed with communication actions often called *"speech acts"*. This communication is based on messages and is compatible with the previous pattern for active objects. The KQML formalism [27, 28] inspired by the theory of speech acts can be used to design the different kinds of messages to be exchanged. The very particular point that makes an *agent* different from an *object* is that its own intentions and goals are used as a filter for communication actions. This means that the decision to perform a service is not only due to the caller but is also submitted to the receiver's objectives. Thus, very high-level intentions can be given to such components, and, why not, consciousness [29] or emotions [30].

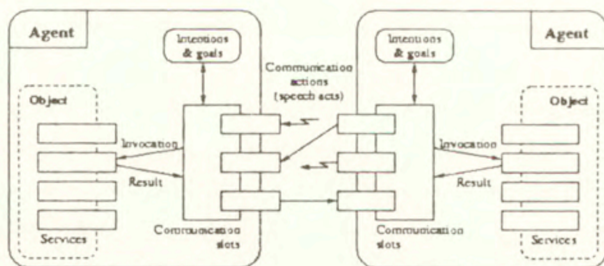

Fig. 2. Structure and communication of agents.

Moreover, concerning Interactive Prototyping, the most interesting thing relevant when describing the components of the simulation as agents relies on the modularity of this model.

http://rcin.org.pl

Actually, if the user is able to dynamically change a component's behavior, the code describing this one must not be deeply dependent from the code describing the other components' behavior. If communications between the components are restricted to message passing, then, internal changes will not affect the existing messages' semantics. However, the dynamic properties of the system allow to create new kinds of messages for the new semantics which may be introduced by these changes.

The fact that an agent is seen by the others through its communication slots is a very important point concerning the multi-agent approach for *Interactive Prototyping*. The way an agent is made inside does not matter to the others. Thus, if the user drives an *"avatar"* to perform actions in the virtual world, then, he can be seen has an agent similar to the others in the simulation.

Consequently, the above requirements imply the use of an object-oriented programming language, with dynamic properties, ensuring the full control of concurrency, using many communication means between objects/agents and offering high-level abstractions for 3D environments.

4 Our proposal : oRis/ARéVi

The preliminary study, previously summarized, drove us into realizing a tool to fulfill our needs. This is a *Virtual Reality* platform allowing to *perform Interactive Prototyping on Multi-Agent Systems*. It can be described as a dynamic, parallel and object-oriented programming language coupled with a high-level *Virtual Reality* toolkit.

4.1. The oRis language

The first step in our realization consists in designing the *oRis* programming language which fulfills many of the requirements described in subsections 3.2, 3.3 and 3.4.

oRis can first be seen as an interpreted object-oriented programming language whose syntax is very near from C++ or Java. Each time a concept used in oRis al-ready exists in one of these

http://rcin.org.pl

two languages, we try to use a similar way to express it. However, *oRis* supports much less low-level details than these languages (only one integer format, only one ,oat format, no character but a built-in string type, no pointer but object names, no array but a built-in multi-dimensional vector type, . . .).Our language allows multiple inheritance and is strongly typed. Instances are dynamically created (as with Java)and are referenced through an explicit object name which represents the metaphor of an object pointer. This object name is automatically given to the instance and is made up of the class name and an integer identifier so that the user can easily "talk" to an instance. Unreferenced instances can be automatically deleted, but we can dynamically choose which instances are submitted to this *"garbage collector"*, and we can always explicitly destroy an instance. The language can be extended with C++ dynamic libraries making C++ objects, methods or functions accessible from oRis code. Thus, specialized C++ classes (GUI, 3D, network management, . . .) can be embedded inside *oRis* classes and then extend the application fields.

One of the main characteristics of this language lies in the fact that all that can be expressed in the initial source files can also be expressed at run-time. The entire language is available on-line and can be very easily used to change the running program whenever we want. The new source code can be introduced to perform a direct action such as instance creation/deletion, method call, . . . An-other purpose can consist in the rewriting of some existing code:

- global function rewriting,
- method rewriting in classes : *void Car::brake(void) { /* new code */}*

In the same way, the new code can contain new declarations and definitions:

- new global functions,
- new classes : *class Car : Vehicle {/* ... */};*
- new attributes and methods in classes : *float Car::avgSpeed;*
- new attributes and methods in instances : *void Car.5::park(void) {/* ... */}*

There are many ways to introduce new source code in a running application:

- the user can write it in a dialog window,
- the program already running can build a string containing this new code and call the parse() function,

http://rcin.org.pl

- the program already running can load it from a new source file,
- some specialized instances can supply this code from a specific origin such as network,

To make these modifications easier, a graphical browser allows the user to inspect every function, class, instance, method, of the application. With this tool, we can, for example, select an instance, change the values of its attributes, change its methods and try them. This browser is entirely written in oRis using both an introspection/reflection package and a GUI package provided by this language.

Another main characteristic of oRis lies in the fact that active objects can easily be created and controlled. If an object is endowed with a **main()** method, then it is considered as an active object. This method represents the entry point of its behavior and is automatically called by the scheduler (and restarted when ended). Moreover, the user can launch other activities in parallel with the active objects, and each active object can consist in several concurrent activities (using the **start** primitive. Of course, basic synchronization tools are avail-able, such as mutual exclusion semaphores or low-level critical sections (which ensure no activity switching during code execution). Furthermore, three possible scheduling policies can be chosen:

- **Cooperative:** each activity explicitly gives up its execution,
- **Preemptive:** each activity is executed during a chosen time slice,
- **Deeply parallel:** an activity switch occurs between each oRis micro-instruction (slower but very useful to detect synchronization failures).

The way an activity is picked to be executed can either be a serial fashion (First In - First Out, should never be used) or a random one (for the fair round-robin (see figure 1). A work about the role of the scheduler in a simulation has been made in [31].

Communication between objects can be performed by direct method invocations as usual when working with objects. A more specific mechanism lies in the use of *"attribute callbacks"* which is a concept similar to *"fields"* and *"routes"* specified in VRML [32] or *"links"* used in the Lightning platform [33]. These are objects in charge of triggering an action

http://rcin.org.pl

just before the value of a specific attribute of a specified instance is changed, and another action just after. This allows, for example, to bound the possible values of an attribute or to propagate variations through a set of connected objects (loops in propagation are detected and broken). Thus, this can be seen as communication means which ensure that a set of objects keep consistent.

Concerning *Multi-Agent Systems*, the communication re-lies essentially on messages. A message contains at least the name of its emitter and can be extended in as many classes as needed by the communication protocol. Such a message, after being composed, is able to reach the receiver's mailbox and then, this latter is able to read and process it whenever it wants. This is an asynchronous *"peer to peer"* communication but these messages can be involved to perform a broadcast communication too. Each agent can dynamically declare itself to be sensible to some kinds of messages so that an appropriate callback is triggered when such a message is broadcasted. In this case, many receivers (not even known by the emitter) can re-act to a message broadcasting. These callbacks are triggered in a synchronous way, but they can simply consist in putting the broadcasted message in the receiver's own mailbox to be processed as an asynchronous message.

Then, from now on, our oRis language allows to build concurrent simulations made up of objects, active objects and agents. With this tool we can achieve applications mixing low-level and high-level concepts, we can dynamically browse and change the content of a running simulation, and we can control the way the scheduler works. Thus, the following stage consists in embedding this language in a Virtual Reality environment to perform *Interactive Prototyping* experiences

4.2. The ARéVi platform

ARéVi is first a Virtual Reality library made up of many C++ classes concerning 3D geometry, kinematics, visualization and interactions. These classes represent high-level concepts, which are not dependent on a specific 3D library, so that they can use different ones (Open Inventor, Ilog Vision, OpenGL, . . .). Some specific devices are managed to allow sensitive immersion (head-mounted displays, trackers, 3D glasses, data gloves, . . .).

http://rcin.org.pl

The oRis <-> C++ link has been used to make these classes available from our language. The main ARéVi classes that can be used in oRis are shown figure 3. They can be described as follows:

- **Object:** root class in oRis,
- **ArEntity:** object with a 3D shape and kinematics,
- **ArLight:** light (many kinds),
- **ArCamera:** camera to choose a 3D point of view,
- **ArViewer:** window to display a camera image,
- **ArScene:** set of 3D objects to be shown,
- **ArEvent:** object sensible to hardware events (key stroke, mouse click, ...)

A particular point lies in the fact that the visualization and interaction means are considered as any other instance of the simulation. Therefore, they can be designed as agents with their own behavior. For example, we can use a viewer which decides on its own to change the display mode (normal, no texture, wireframe, . . .) according to the measured performances (number of frames per second). We can also use a scene which automatically decides to integrate the entities near its geographic area and to reject the others.

Running a Virtual Reality application with ARéVi consists in extending these oRis classes to describe objects/agents and interaction means, in letting them be activated by the scheduler, and then, in making interactions and changes with the specific devices or in *"immersion trough the language"*.

http://rcin.org.pl

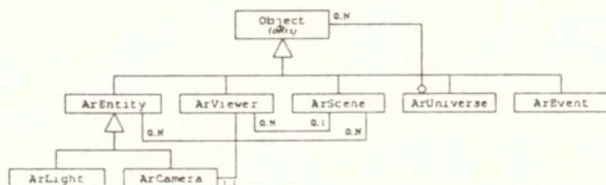

Fig. 3. UML Diagram of the main AReVi classes

4.3. An Example : multiple robot navigation

We are now going to present a simple example to illustrate the on-line modeling capacities of our system.

We consider 4 groups of 4 robots. Each group of robots have the same goal to reach. The problem is how to program the behavior of each robot.

The fist very simple approach consists in guiding each robot toward its goal without taking into account what the others are doing. The prototype of the robot's agent is given table 1. In particular, we can see that there is no obstacle avoidance method.

We can then observe the result for this first behavior. The robot is moving, acting according to its behavior and we have no way to change what it is doing. . This is what we called 3D animation. We can do of course more : change interactively the position of the goal to observe more precisely what the system is able to do. This is what we call interactive simulation. Moving the position of the goals will of course allows us to see the obvious limitations of our first approach . As there is no obstacle avoidance, the robots will collide.

```
class Human : ArEntity, ArEvent
{
Ball _goal;
```


http://rcin.org.pl

```
void new(void);
void delete(void);
void stop(void);
void main(void);
void walk(void);
void reachGoal(void);
};
```

Table 1. Prototype of the robot's agent

The next step is now to modifiy interactivley the model of the agent by adding a new method : the *avoidance* method. This is what we call virtual reality and dynamic prototyping.

The first avoidance method we add is the following : when a robot is seeing another one close, it is turning in the other direction. Once again, we go back to interactive simulation and test the whole system.

This very simple avoidance strategy works fine on most cases. But there are still problems and under certain circumstances, the robots might block each others (see figure 4).

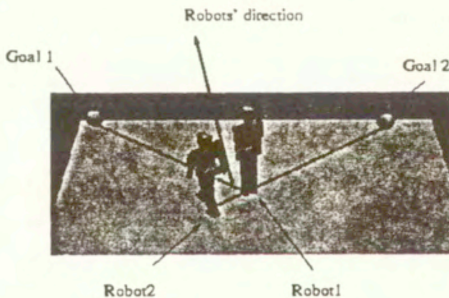

Fig. 4. Both robots are blocking each others and go in the wrong direction

http://rcin.org.pl

We can then refine the behavior : when two robots try to avoid each other, one might stop to let the other one go. This new behavior can be first tested on a group of robots (instance granularity), and if satisfactory, be used for the whole population.

5 Conclusion and future works.

Our objective was to show that a multi-agent virtual reality environment could be performed for Interactive Prototyping. The main point of Interactive Prototyping concerns the use of dynamic modeling capabilities in a virtual environment designed as a Multi-Agent System.

As far as we are concerned, the oRis / AReVi platform we provide seems to be a useable tool to interactively design and tune a distributed complex system as a robot or multiple robots and their environment. The complex behavior of such a system can be described by many agents whose interactions ensure a consistent representation of the global virtual world. The dynamic properties of our language allow the user to decide what becomes the model he is working on.

The next stage of our work relies on the use of the KQML formalism and the dynamic properties of our language to build behavior servers. This could be used by an agent to dynamically ask the environment it is entering, the behavioral primitives usually run in it. Of course, the use of this concept requires both a high-level dialog between agents, such as KQML, and the capability for an agent to rewrite its own code. Another subject under study concerns the capability to run a session on many workstations to allow Interactive and Cooperative Prototyping. The low-level layer consisting in message exchanges between workstations is already running but we are still working on the way to keep the simulation consistent. Actually, running a distributed simulation of a system is quite different from simulating a distributed system. The updates between the different workstations can be very easily performed by sending directly some oRis code, but the policy in charge of these updates (as "dead-reckoning" to ensure the graphical consistency) needs to be studied with a particular attention.

http://rcin.org.pl

References

[1] P. David. Three application cases of virtual reality to SNCF (in French). *In Sixième journée du Groupe de Travail Réalité Virtuelle*, CNET/France-Telecom, Issy-les-Moulineaux (France), Mars 1998.SMITH, I.: On path planning of mobile robot. *Journal of Autonomous Systems*, **11** (1986), pp. 123-230.

[2] A. Fagot. Application of a virtual reality solution to an industrial case (in French). In *Sixième journée du Groupe de Travail Réalité Virtuelle*, CNET/FranceTelecom, Issy-les-Moulineaux (France), Mars 1998.

[3] F. Coudret. Development of a VR platform adapted to the building sector (in French). In *Sixième journée du Groupe de Travail Réalité Virtuelle*, CNET/France-Telecom, Issy-les-Moulineaux (France), Mars 1998.

[4] [4] A. Thibout, P. Even, and R. Fournier. Virtual reality for teleoperated robot control. In *ORIA'94,From telepresence towards virtual reality*, Marseille (France), pages 131-139, December 1994.

[5] J. Tisseau, P. Reignier, and F. Harrouet. Exploitation of models and virtual reality (in French). In *Actes des sixièmes journées du groupe de travail "Animation, Simulation, Systèmes Dynamiques"*, Toulouse (France), pages 13-22, October 1998.

[6] J. Wernecke. *The Inventor Mentor*. Addison-Wesley,1994.

[7] Silicon Graphics. *OpenGL Optimizer*. http://www.sgi.com/software/optimizer.

[8] Silicon Graphics. *Fahrenheit*. http://www.sgi.com/fahrenheit/home.html.

[9] ILOG. *Ilog Vision 1.0 Reference Manual*. ILOG S.A.,1996.

[10] E. Gobbetti. *Virtual Builder II*. PhD thesis, EPFL DI-LIG, 1993.

[11] M. Green and L. White. *Minimal Reality Toolkit*, Reference Manual. University of Alberta, 1995.

[12] Q. Wang. *Networked Virtual Reality*. Master of Science, University of Alberta, Edmonton, 1994.

[13] Medialab. *Clovis - user's guide*. Medialab, 1995.

[14] P. Astheimer, W. Felger, and S. Muller. *Virtual Design: a generic VR system for industrial applications.Computer Graphics*, pages 671-677, 1993.

[15] Division. *Dvise*. http://www.division.com.

http://rcin.org.pl

[16] C. Carbsson and O. Hagsand. *Dive - a platform for multi-user virtual environment. Computer Graphics*,17:663-669, 1993.

[17] A. Poggi and G. Rimassa. An Efficient and Flexible C++ Library for Concurrent Programming. *Software -- Practice and Experience*, 28(13):1437-1463, November 1998.

[18] O. Coulaud and E. Dillon. Para++: A High Level C++ Interface for Message Passing. *Journal of Parallel and Distributed Computing*, 51:46-62, 1998.

[19] C. Petitpierre. Synchronous C++: A Language for Interactive Applications. *IEEE*, pages 65-72,September 1998.

[20] R.Bagrodia. Parallel Languages for Discrete-Event Simulation Models. *IEEE Computational Science & Engineering*, 5(2):27-38, April-June 1998.

[21] M. Rinard and M. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483-545, May 1998.

[22] A. Chien, J. Dolby, B. Gangul, V. Karamcheti, and X. Zhang. Evaluating High Level Parallel Programming Support for Irregular Applications in ICC++. *Software -- Practice and Experience,* 28(11):1213- 1243, September 1998.

[23] C. Petitpierre. *sC++ -- Programmation pseudo parallèle orientée objet.* Presses Polytechniques Universitaires Romandes, 1998.

[24] M. Carroll. Active Objects Made Easy. *Software -- Practice and Experience*, 28(1):1{22, January 1998.

[25] J. Ferber. *Les systèmes multi-agents - vers une intelligence collective.* InterEditions, 1995.

[26] J. Ferber. Multiagent systems: a general overview (in French). *Technique et science informatique*, 16(8):979-1012, 1997.

[27] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Third International Conference on Information and Knowledge Management, CIKM'94*, ACM Press,1994.

[28] *The DARPA Knowledge Sharing Initiative, External Interfaces Working Group. Specification of the KQML agent-communication language* - Draft. DARPA, http://www.cs.umbc.edu/kqml/papers/kqmlspec.ps, June 1993.

[29] M. Tokoro. An agent is an individual that has consciousness. In *ECAI'96 Workshop*, Budapest (Hungary), pages 45-46, August 1996.

http://rcin.org.pl

[30] H. Maldonado, A. Picard, P. Doyle, and B. Hayes Roth. Tigrito: A Multi-Mode Interactive Improvisational Agent. In *IUI'98*, pages 29-32, 1998.

[31] P. Chevaillier, F. Harrouet, and P. De Loor. Application of Petri nets to the validation of multi-agent systems for control (in French). *Journal Européen des Systèmes Automatisés*, (submitted), 1999.

[32] *The Virtual Reality Modeling Language*, ISO/IECDIS 14772-1. http://vrml.sgi.com/moving-worlds, April 1997.

[33] J. Landauer, R. Blach, M. Bues, A. R-osch, and A. Simon. Toward next generation virtual reality systems. *IEEE*, pages 581-588, 1997.

[34] P. Chevaillier, J. Tisseau, F. Harrouet, and R. Querrec. Prototyping Manufacturing Systems. Contribution of Virtual Reality, Agent Systems and Synchronous Interpreted Petri Nets. In *INCOM'98*, Nancy & Metz (France), June 1998.

[35] P. Chevaillier, F. Harrouet, P. Reignier, and J. Tisseau. Using Virtual Reality for Manufacturing Systems Prototyping. In *7ème journées du Groupe de Travail Réalité Virtuelle*, Laval (France), June 1999.

http://rcin.org.pl