A Reactive Behavior framework for dynamic virtual worlds *

Frédéric Boussinot and Jean-Ferdinand Susini
INRIA EMP-CMA/MIMOSA
2004 route des Lucioles BP 93 06902
F-Sophia Antipolis FRANCE
{frederic.boussinot, jean-ferdinand.susini}@sophia.inria.fr

Frédéric Dang Tran and Laurent Hazard
France Telecom R&D DTL/ASR
38-40 rue du Géréral Leclerc
92794 Issy Moulineaux Cedex 9 FRANCE
{frederic.dangtran, laurent.hazard}@francetelecom.fr

Abstract

This paper presents a Java-based reactive programming framework well adapted to the construction of complex behaviors for CG objects within virtual environments. This reactive approach is based on an instantaneously broadcast event model and a semantically-sound synchronous/reactive formalism. The reactive framework degree of expressiveness is illustrated through several examples of behaviors which range from low-level animation of virtual creatures to high-level control of autonomous creatures' actions.

Keywords: virtual world, behavior, reactive systems, animation

1 Introduction

Building rich and entertaining 3D virtual environments accessible over the Internet involves not only the definition of a "modelling language" defining how objects look (or sound) and how they are organized spatially but also how they behave either as the result of events occuring in their environment (say a collision) or spontaneously ("bots"). In this regard, the VRML2.0 standard, with its behavior and scripting capabilities, has gone a long way towards providing means to describe dynamic 3D worlds. The VRML2.0 standard, proper, does not enforce a particular scripting system or programming language but just describes an execution model and an access protocol for external scripting languages. So far simple scripting languages (e.g. Javascript) or general-purpose object-oriented programming languages (e.g. Java) have been used to define VRML worlds' behaviors. Programming complex behaviors in VRML2.0 in this context beyond simple animation control of 3D objects is not necessarily easy.

The present paper proposes the use of a Java-based reactive approach for associating complex behavior to graphical entities within 3D virtual worlds. More precisely a reactive programming and execution model is proposed that fulfills the following requirements:

- It is semantically sound: there exists a formal semantics of proposed programming primitives yielding deterministic and reproducible execution.
- It has efficient implementations capable of coping with a large amount of concurrent behaviors and events.
- It is expressive enough in order to allow fine control over behaviors and the definition of complex synchronization constraints, for example:
  - the ability to preempt the execution of a behavior by presence of an event ("move toward the target until you receive an order to abort your mission")
  - the ability to react to an arbitrary combination of event occurrences or non-occurences ("move forward if the door is opened and no abort mission signal is received")
- it allows the incremental construction of complex behaviors by the combination (and reuse) of more elementary behaviors.
- it allows highly dynamic systems in which new behaviors and events can be added at run-time without restrictions.
- it is not tied to a modelling langage or rendering system and can be used for non-graphical simulations (e.g. for multi-user virtual world systems in which the computation of object behaviors is performed by non-graphical servers).

The structure of the paper is as follows: section 2 gives an overview of the reactive approach and of Junior, a Java based formalism for reactive programming. The application of this reactive framework for designing behaviors of objects in virtual worlds is considered in section 3. Section 4 describes the use of broadcast events for coding physical laws. Section 5 explains how the reactive framework has been integrated or can coexist with the VRML model. Related work is considered in section 6. Future work is covered in the following section. Finally we conclude.

2 Reactive Approach

The reactive approach proposes a flexible paradigm for programming reactive systems [4], especially those which are dynamic (that is, the number of components and their connections can change during execution). Reactive programming provides programmers with concurrency, broadcast events, and several primitives for gaining fine control over reactive programs executions. At the basis of reactive programming is the notion of a reaction: reactive programs are reacting to activations issued from the external world. Program reactions are often called *instants*. The two main notions are reactive instructions whose semantics refer to instants, and reactive machines whose purpose is to execute reactive instructions in an environment made of instantaneously broadcast events.

Junior [6] is a Java-based language for programming reactive behaviors. Basically, programming with Junior means:

- Writing a *reactive instruction*, which describes an application program.

*The work presented in this paper has been carried out within the scope of the IST Project IST-1999-11488 PING (Platform for Interactive Networked Games)[10]

- Declaring a *reactive machine*, to run the program.
- Adding the program into the machine.
- Running the machine; this is usually performed using a non terminating loop, which cyclically makes the machine and the program *react*.

Programming in Junior has a dynamic aspect: machine programs can be augmented by new reactive instructions added during machine execution. New instructions added to a machine do not have to wait for the termination of the actual program, but are run *concurrently* with it.

Junior concurrent reactive instructions can communicate using *broadcast events* that are processed by reactive machines. Broadcasting is a powerful and fully modular means for communication and synchronization of concurrent components. Broadcasting in Junior has a special coherency property: during a machine reaction, the same event cannot be tested both present and absent, even by two distinct concurrent instructions.

Junior defines primitive constructs allowing for code (reactive instruction) migration over the network. This aspect will not be considered here.

Junior is pure Java. It is provided with an API named Jr[7]. Using Jr, programmers can define reactive instructions and reactive machines, and have possibility to run them. Junior is a programming language, defining constructs for reactive programming. It can also be seen as a Java programming framework. From this last point of view, Junior provides Java programmers with an alternative to the standard threading mechanism. The benefit is that Junior gives solutions to some well-known problems of Java threads (see [8] for a description of these problems, and [2] for a comparison of Java threads with the related SugarCubes formalism).

2.1 Reactive Instructions

Reactive instructions are state-based statements, run (one also says, *activated*) by reactive machines. Some cyclic instructions are never ending across instants, while others are reaching a final state after several activations; in this case, one says that they are *completely terminated*. Because states are embbeded in them, reactive instructions are not reentrant: they must be copied, in order to get new execution instances.

Reactive instructions are composed from a small set of basic constructors. For example, the constructor `Seq` puts two reactive instructions A and B in sequence: A is executed up to complete termination (remember that it may take several machine reactions), and then B is executed. The associated state of `Seq` encodes termination of the first component: the state changes when the first component completely terminates; then, following executions directly go to the second component, without considering the first one.

Reactive instructions are Java objects implementing the `Program` interface. They are built using static methods of class `Jr`[1]. For example, to define the sequence of two reactive instruction A, B, one writes:

```
Program p1 = Jr.Seq(A,B);
```

Syntax for constructors is very basic; for example, to put three instructions A, B, and C in sequence is not directly possible with one unique call of the `Seq` constructor, which must be called twice:

```
Program p2 = Jr.Seq(A,Jr.Seq(B,C));
```

[1] To simplify, one also calls "constructors" the static methods of Jr which call constructors of Junior classes.

Among the reactive instructions are the ones, called *atoms*, used to interface with Java. Basically, an atom executes an action which possibly performs some interaction with the Java environment. The action is executed once and the atom immediately terminates after first reaction. Execution of an atom is *atomic*: once started, execution of an atom always terminates without any interference with other atoms.

2.2 Reactive Machines

Reactive machines implement interface `Machine`. A reactive instruction can be given at construction; when it is the case, it becomes the initial program of the machine. Reactions of reactive machines (often simply called "machine", for short) are obtained using the `react` method of interface `Machine`. One can now gives a minimalist example of Junior code:

```
public class HelloWorld
{
    public static void main(String[] args){
       Machine machine = Jr.SyncMachine();
       machine.add(Jr.Atom(
                    new Print("hello, world!")));
       for (int i = 0; i < 3; i++){
           System.out.print("instant "+i+": ");
           machine.react();
           System.out.println("");
       }
    }
 }
```

The `main` method defines a reactive machine (instance of class `SyncMachine`), and adds in it a program to print a message (using action `Print`); finally, the reactive machine is activated 3 times (a trace shows the sequence of machine reactions). One obtains output:

```
instant 0: hello, world!
instant 1:
instant 2:
```

The message is printed at first instant, that is during the first machine reaction. The two next reactions are empty, as the machine program is completely terminated at the end of first reaction.

The `Stop` instruction is the basic way to delay execution for the next instant. Executing a `Stop` terminates execution for current instant; however, execution is not completely terminated at this stage, and the `Stop` instruction becomes the new starting point for next instant. The state associated to `Stop` encodes end of the current instant: it changes at the end of the first instant of execution, indicating that instruction is completely terminated. Replacing in `HelloWorld` the added program by:

```
Jr.Seq(Jr.Atom(new Print("hello, ")),
       Jr.Seq(Jr.Stop(),
       Jr.Atom(new Print("world!"))))
```

would produce:

```
instant 0: hello,
instant 1: world!
instant 2:
```

Now, the `Stop` instruction splits execution in two distinct instants: "`hello,`" is printed during the first one, while "`world!`" is printed during the second one.

2.3 Events

Event are non persistent data with a binary status *present* or *absent*, possibly changing at each instant. An event becomes present during one instant as soon as it is generated by a program component during this instant. A strong coherency property holds: *during one instant, the same event cannot be tested as present by one component and as absent by another component.* In other words: *events are broadcast*.

A way to implement the coherency property of Junior machines is as follows:

- A new *unknown* event status is introduced; the machine assigns it to all events at the beginning of each instant.
- The status of an event is changed to *present* as soon as it is generated.
- When the machine detects that no new event can be generated, it changes to *absent* the status of all unknown events and decides the end of current instant.

Note that end of instant and absence of events are decided together, in the same step; this has important consequence, which are not discussed here (see [7] for details).

Values can be associated to events, during generations. Values generated during the same instant for one event are collected during the instant and stored in a table associated to the event. All collected values are available at next instant.

The Jr API gives several ways to deal with events. In the simplest one, events are identified by strings. For example, to generate an event named `e`, one writes: `Jr.Generate("e")` and to wait for it: `Jr.Await("e")`. In this last instruction, control is stopped while event `e` is not generated, and the instruction is completely terminated when `e` becomes present. The `Await` instruction has an associated state which codes for termination, that is, for the awaited event generation.

2.4 Concurrency

Junior owns a concurrency constructor, named `Par` (for parallelism) which puts two reactive instructions A and B in parallel: A and B are executed at each instant, and the parallel construct is completely terminated when both A and B are. The state of `Par` is the union of the state of A and of the state of B. The order in which, at each instant, A and B are executed is left unspecified.

Reactive instructions added to a machine are put in parallel with the machine program. However, to simplify programming and reasoning about reactive programs, an instruction added to a machine during the course of a reaction is not immediately run by the machine; actual adding of the instruction to the machine program is delayed to the *beginning of the next instant*. Actually, this is quite a general attitude in Junior: to avoid interferences, program changes issued by the external world are systematically delayed to next instant.

2.5 Implicit Java Object

Interfacing reactive instructions with Java is made easier using *implicit* Java objects set by *link* instructions. The implicit Java object set by a link instruction can be directly accessed and transformed by atoms executed by the link body.

The `Jr.Link(object,body)` reactive instruction defines `object` as being the implicit Java object associated to the reactive instruction `body`.

Actions executed by atoms (see 2.1) must implement the `execute` method with signature `void execute(Environment env)` and the implicit Java object (if defined) is returned by method `linkedObject` of `env`.

2.6 Preemption and Control

Junior defines two operators to get fine control over reactive instructions; one is the `Until` preemption operator which forces a reactive instruction to terminate when an event is present; the other one, called `Control`, allows a reactive instruction to execute according to presence of an event.

Instruction `Until` has the form:

```
Jr.Until("event",body,handler)
```

where `body` and `handler` are two reactive instructions. Execution of `body` is abandoned (one says, it is *preempted*) as soon as `event` becomes present; in this case, control directly goes to `handler` which is then executed.

The `Control` instruction gives a way to execute a reactive instruction only at instants where a given event is present. At others instants, the reactive instruction just stays in the same state, without executing anything. Actually, the `Control` instruction can be seen as "filtering" instants for its body: the body can proceed only when `Control` let instants reach it.

3 Reactive Behaviors for VWs

Objects in VWs combine a graphical aspects (usually 3D) and a behavior. Behaviors are often composite, combining standard behaviors (for example, ability to process collisions) with specific ones (for example, a pursuit behavior). In this section, one considers the two basic inertia and collision behaviors, and the way to combine behaviors to get more complex ones.

3.1 Defining Reactive Behaviors

There are three levels when defining reactive behaviors for VW objects:

- Pure data processing; it is implemented by object methods accessing object data.
- Interface between data processing and reactive behavior; it is implemented by atoms calling object methods.
- Reactive behavior; it is a reactive instruction executing previous atoms.

A `Link` instruction (see 2.5) is used to link the reactive behavior to a particular VW object. Then, the linked reactive behavior can be added to a reactive machine to be run by it. Existence of one or more reactive machine in the system depends on the VW considered (typically there is one reactive machine per process).

All reactive behaviors added in a machine share the same instants and are run at each instant; this has important consequences:

- When modeling physics, it is natural to identify instants with the basic time step dt appearing in equations.
- One gets a "fair" execution strategy, in which all object have globally same possibility to execute. Note that this property is not directly given by threading mechanisms, even preemptive ones.

3.2 Inertia

An object with an inertia behavior tends to maintain its speed. More precisely, inertial objects have data for position at current instant (`x` and `y`), and for speed (`speedx` and `speedy`). Method `inertiaAction` translates the object according to its speed; it is the basic data processing method of inertial objects:

```
  public void inertiaAction(){
     x += speedx; y += speedy;
  }
```

An action is defined for interfacing data processing with reactive behavior:

```
public class InertiaAction implements Action
{
  public void execute(Environment env){
   ((InertialIcobj)env.linkedObject()).
        inertiaAction();
  }
}
```

Inertial behavior consists in executing action `InertiaAction` at each instant; thus, to give it to an object `O`, one just has to add the following reactive instruction to the reactive machine:

```
Jr.Link(O,Jr.Loop(
            Jr.Seq(
               Jr.Atom(new InertiaAction()),
               Jr.Stop())))
```

Then, method `inertiaAction` of object `O` is called at each instant, which gives `O` an inertial behavior.

3.3 Collision

To process collisions is more complicated than to simply get an inertial behavior, because an object must know what are the others objects it collides. A solution is to have a global workspace in which objects are registered. The list of registered objects is returned by method `elements` of `workspace`. Here is method for determining which objects are involved in collisions:

```
public void collideAction(){
   Enumeration list = workspace.elements();
   while(list.hasMoreElements()){
      Icobj other = (Icobj)list.nextElement();
      if (other == this) continue;
      if (collisionCondition(other)){
            collide((InertialIcobj)other);
      }
   }
}
```

Methods `collisionCondition` which detects actual collisions, and method `collide` which performs collision are not given here.

3.4 Combinations of Behaviors

Parallelism is the basic operator for combining behaviors. For example, to get both inertial and collision behaviors, one simply put in parallel the two previous behaviors[2]:

```
Jr.Par(
   Jr.Loop(Jr.Seq(
            Jr.Atom(new InertiaAction()),
            Jr.Stop()))),
   Jr.Loop(Jr.Seq(
            Jr.Atom(new collideAction()),
            Jr.Stop())))
)
```

[2]Class of collision objects extends class `InertialIcobj` of inertial objects.

More complex behaviors can be obtained using reactive primitives presented in section 2. For example, consider the following behavior:

```
Jr.Par(
      inertia(),
      Jr.Seq(
         Jr.Until("revenge",
             runAway(),
             Jr.Repeat(50,Jr.Stop())),
         pursuitAndDestroy()))
```

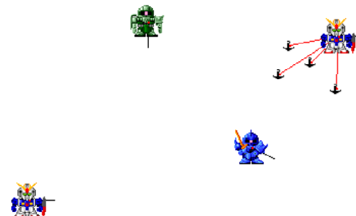
It corresponds to an object with an inertial behavior (returned by method `inertia`), which runs away until event `revenge` becomes present. When it is the case, object ends to run away, and, after 50 instants, it starts a new pursuit behavior (returned by method `pursuitAndDestroy`).

3.5 Robots Demo

This 2D demo program makes use of the behaviors described previously to construct a virtual arena in which autonomous robots are pitted against one another. Four robots are placed in a rectangle arena. These 4 robots have in common an inertia+collision behavior; moreover:

- Robot Zaku can launch bombs that explodes after 50 instants.
- Robot Gouf can launch webs that, after 50 instants, can capture other robots.
- Two similar robots Gundam have the run away/pursuit behavior described in section 3.4. Moreover, just before being killed by a bomb, they generate event `revenge`.

Strategy for Zaku consisting to kill one Gundam robot, then the other, is very dangerous; indeed, there is risk for Zaku to be destroyed by the remaining Gundam, as it has received event `revenge`. A safe strategy for killing both Gundam robots is first to capture one of them using Gouf, then to kill the other using Zaku, and finally to kill the captured robot. The following figure shows a situation where one Gundam robot has been captured by Gouf:



4 Use of broadcast events

In this section we show that instantaneously broadcast events are well suited for coding physical laws, such as gravity or attraction, and also superpositions of such laws.

Actually, we consider event generations with associated values (see 2.3), and we use a particular mechanism, called “event listener” to process generated values (event listeners are presently not part of Junior, but are implemented in the new version of SugarCubes). Basically, an event listener for an event `E` reacts during current instant to all values generated for `E`; listener reaction to a

generation consists in executing an atom, giving it the associated value as argument. Note that, in this way, the atom is executed during an instant as many time as `E` is generated during this very instant.

4.1 Gravity

Gravity can be seen as an interaction initiated by a `Ground` object which applies the same attraction vector to all objects present in the VW. One can thus imagine that, at each instant, `Ground` generates a `gravity` event representing the interaction, with an associated value which is the attraction vector. To be attracted, an object runs a behavior with a parallel component which is a loop that listen to `gravity` and calls an atom to fall on the ground.

Use of a broadcast event ensures that `gravity` can be seen by each object in the system, with same attraction vector, and at same instants. For example, one object will never receive `gravity` several time while an other one will not receive it. Therefore, it becomes unnecessary to introduce any particular time stamping techniques or synchronization protocol to ensure that every objects in the system have the same coherent vision of the world.

In this approach, each object in the system is responsible of its own state, and modifies it according to its own behavior or in response to other objects requests sent through broadcast events. For example, one can introduce in the system "phantom" objects which do not obey to gravity law; introduction of these new objects does not imply any change to `Ground` or to something else in the system.

4.2 Attraction

Now, in a little bit more complex scenario, a planet `Earth` is generating an event called `attraction` (different from `gravity` of the previous example); `attraction` is generated at each instant by `Earth` behavior, and position and mass of the planet are the values associated to it.

Let us consider a `meteor` object with an inertial behavior (described in section 3.2) and suppose that it also responds to `attraction` by applying the model of planet attraction (it calculates the attraction vector according to its own position and mass, and to position and mass of the attracting object; then it updates its speed accordingly). The result of the parallel combination of the inertial behavior and of the response to `attraction` automatically fits the physical principle of superposition of physics: the meteor object is attracted by the earth object, and, with appropriate values, it can be put in orbit around it.

This example shows the benefit of using instantaneous broadcast events:

- The behavior of a simple inertial object can be simply enhanced to respond to planet attraction law just by putting the corresponding behavior in parallel with the inertial one.
- A planet doesn't have to worry about which objects are really attracted. So, it can attract meteors, but also other planets, spatial ships, etc.
- A meteor doesn't have to take care about the object which is attracting it; it can be a planet, a star, another meteor, or anything else which generates the `attraction` event.
- If an other planet is introduced in the system, the event model automatically ensures that objects listening `attraction` will see two occurrences of it, at each instant; thus, they will behave accordingly to presence of the two planets, without anything to be changed in them.

4.3 Jurassic Park Demo

This 3D demo illustrates the use broadcast events to coordinate interactions between 4 dinosaurs moving in a scene with 100 trees. Dinosaurs are under control of the user. Here is a snapshot of the demo:

The demo has the following characteristics :

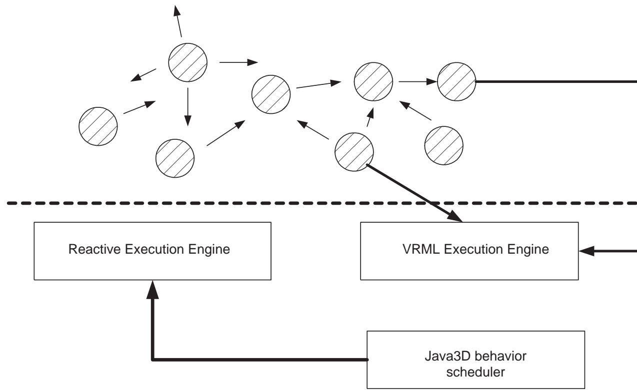
- Each object (dinosaur and tree) in the VW has its own specific behavior.
- There is no central control component (as the `workspace` of 3.3) knowing all the objects present of the simulation.
- Instantaneous broadcast events is the only means for communication between objects.
- Collisions are handled by events : each object in the system generates at each instant a dedicated event carrying its coordinates, its speed, and its volume (which, in a first attempt, is a sphere centered around the object).
- Collection of objects for 3D rendering uses instantaneous broadcast events : each camera in the world generates event `display here` to collect objects it will render at the end of the instant; then, objects can decide, according to their needs, their positions, or whatever else, if they register for rendering or not.

This techniques allow us to easily build several perfectly synchronized views of the same world.

The instantaneously broadcast event model provided by Junior is a very powerful mean in order to express objects behaviors in VWs, especially in the modeling of physical interactions between objects. In addition, it increases modularity in the design of objects by increasing independence between the implementation of an object and of its behavior, and the rest of the VW. Object are more reusable and expendable, communication between objects being clearly concentrated in the event model.

5 Integration with VRML and Java3D

The reactive programming framework presented in this paper is in the process of being integrated with VRML and Java3D. The current implementation relies on the VRML loader on top of the Java3D API. The following figure shows the software architecture of the platform.



The *Reactive Execution Engine* is responsible for the execution of the reactive program which is the aggregation of all reactive programs attached to virtual world entities (shown as grey circles). It embeds an instance of a reactive machine as described in section 2.2 . This engine is implemented as a Java3D behavior with an appropriate wake-up criterion (typically one activiation per frame or one activation per period of time). The wake-up of the engine entails a machine reaction.

The two execution models, the reactive one proposed here and the VRML one, can be considered as complementary.

The VRML event model [12] relies on an explicit routing of event between event producers and consumers. One-to-one, one-to-many (fan-out) and many-to-one (fan-in) communication patterns are possible. One logical instant in the VRML execution model corresponds to one cascade of events along these routes. The reactive approach proposed here instead relies on a broadcast event model. This communication model can be compared to radio communication. In broadcast communication emiters and listeners do not have to know about each other. The only requirement is that they communicate on the same frequency, using the same protocol.

The execution model of VRML is under-specified when it comes to script execution. In particular it allows asynchronous scripts which can create their own activities (threads typically). On the contrary, the proposed architecture allows a tight coupling between the (reactive) execution engine and the behavior programs that it supports. Concurrency and premption are described using high-level primitives providing by the reactive framework.

In our platform, the VRML machinery is not connected to the Java3D scheduler but is triggered by individual reactive objects within a reactive program. For example, VRML TimeSensor nodes used to animate elements of the scene graph are activated through calls to the *simTick(time)* method. Within the same reactive instant, all TimeSensors are activated with the same *time* value which allows fine grained synchronization between different objects.

6 Related Work

The reactive approach described in this paper has been used in several contexts beyond behavior control of 3D objects. One of them is called *icobj programming*; it proposes a new fully graphical programming technique, which has been used for designing several reactive applets available on the Web [11].

Junior is closely related to SugarCubes[1] and is a descendant of it. Actually, Junior is the kernel of SugarCubes, with a formal semantics expressed using rewriting rules[6]. SugarCubes is compared to Java threads in [2]; comparison remains valid, replacing SugarCubes by Junior.

Actually, Junior and SugarCubes belongs to the family of synchronous/reactive formalisms. The common point of these formalisms is the presence of instants and of a synchronous parallel operator (synchronous because all parallel components are run at each instant). Well-known synchronous languages include Esterel, Lustre, and Signal (see [3] for a survey of these languages), and Statecharts[5]. These languages put the focus on verification and validation of embedded systems; for that purpose, they forbid all forms of dynamicity.

The aim of reactive formalisms is basically to add dynamicity to the synchronous approach. Note that dynamicity is mandatory in the context of VWs, where new objects can appear at any time.

7 Future work

As part of the European IST PING project[10], we are in the process of integrating the reactive programming framework presented in this paper within an infrastructure for large-scale multi-user VWs.

PING will follow and enhance an approach based on a distributed reactive programming model. This is a hybrid synchronous/asynchronous model in which synchronous groups of objects (i.e., sets of reactive objects sharing the same logical instant as envisaged so far in this paper) communicate with one another asynchronously. This model is well suited for large-scale distributed environments, where a tight synchronisation between the active objects in the system seems hopeless. This model is also well adapted for the programming of object behaviours in a shared virtual world: synchronous groups of active objects can be set dynamically according to the grouping and interaction models of the virtual environment (e.g., objects whose perception and influence capabilities are quantified by ”auras”).

A distributed reactive object is made up of several replicas, each being hosted by a simulation process. One of these replicas is distinguished as a “master replica”. In this context, the problem is to maximize coherency of local simulations while minimizing synchronizations between masters and their replicas (because network is involved). The reactive behavior of the logical object can be split in arbitrary fashions between the master and its replicas.

One possible solution to this problem is called *dead-reckoning*[13]; it consists in predicting objects moves using extrapolation. We plan to investigate how dead-reckoning strategies can be implemented at object behavior level, and not, at lower level, by platform. This should lead to flexible systems capable for example of a certain kind of *reflexivity* for solving the coherency/synchronization problem. Our approach has some similarities with the work carried out within the DIS-Java-VRML working group [14] though we are not tied to the DIS protocol.

For example, consider master objects with an inertial+collision behavior, and replicas with only an inertial behavior. This falls into the family of dead-reckoning strategies because replicas can be seen as extrapolating master movements, which is completely safe in absence of collision. A solution to deal with collision would be to force synchronization of replicas each time a master detects a collision. Despite the fact that replicas would not actually process collisions, they would be forced by masters to behave as if they where doing so. This is an example where replicas can have “lighter” behaviors while preserving coherency.

8 Conclusion

We have presented a new approach for programming behaviors of VW objects based on a reactive and synchronous formalism allowing to get fine control over behaviors with a clear and sound semantics. Reactive behaviors reactions are defined as *instants* and inter-behavior communication is achieved according to an instantaneous broadcast model. Reactive programming has several advantages for programming behaviors of objects in VWs:

- Instants naturally match the basic discretization time step dt appearing in models of physical phenomena.
- Broadcast events are a powerful and modular means for synchronizing behaviors and for establishing communication between them.
- Reactive programming allows users to get fine control over behaviors, without suffering problems of Java threads. This, for example, allows to program new specific behavior scheduling algorithms.

The reactive programming framework proposed here can be considered as complementary to the behavior support in VRML which is strongly biased toward the control of the graphical apperance of VW objects. We are convinced that the reactive programming approach allows programmers to concentrate on the semantics of the application rather than on its graphical appearance and provides a cleaner support for application-level events that correspond to an arbitrary logic.

References

[1] F. Boussinot, J-F. Susini, *The SugarCubes Tool Box - A reactive Java framework*, Software Practice & Experience, 28(14), 1531-1550, 1998.

[2] F. Boussinot, J-F. Susini, *Java threads and SugarCubes*, Software Practice & Experience, 30(5), 545-566, 2000.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Pub., 1993.

[4] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, in K. R. Apt (ed.) Logics and Models of Concurrent Systems, NATO ASI Series F, Vol. 13, pp. 477-498, Springer-Verlag, New York, 1985.

[5] D. Harel, *StateCharts: A Visual Approach to Complex Systems*, Science of Computer Programming, 8(3), 1987.

[6] L. Hazard, J-F. Susini, F. Boussinot, *The Junior reactive kernel*, Inria Research Report 3732, July 1999.

[7] L. Hazard, J-F. Susini, F. Boussinot, *Programming with Junior*, available at http://www.inria.fr/mimosa/rp/Junior, July 2000.

[8] JavaSoft, *Why JavaSoft is Deprecating Thread.stop, Thread.suspend and Thread. resume*, JavaSoft Documentation, available at URL: http://java.sun.com/products/jdk/1.2/docs/-guide/misc/threadPrimitiveDeprecation.html.

[9] http://www.inria.fr/mimosa/rp

[10] http://www.arttic.com/projects/ping/

[11] http://www.inria.fr/mimosa/rp/Icobjs

[12] VRML97 International Standard ISO/IEC 14772-1:1997, the VRML Consortium, 1997.

[13] IEEE Standard for Distributed Interactive Simulation-Application Protocols, IEEE Std 1278.1-1995.

[14] Distributed Interactive Simulation DIS-Java-VRML Working Group: http://www.web3D.org/WorkingGroups/vrtp/dis-java-vrml