A Visual Reactive Framework for Dynamic Behavior Creation

Christian Brunette
INRIA EMP-CMA/MIMOSA
2004 route des Lucioles BP 93
06902 F-Sophia Antipolis France
christian.brunette@sophia.inria.fr

October 17, 2002

Abstract

This paper presents Icobjs, a simple, Java-based, visual framework for the programming of reactive applications. In the Icobjs ("Iconic Object") formalism, graphical objects represent basic and hierarchically composed reactive behaviors that can be easily combined to produce more complex ones. The reactive language, on which Icobjs is based, is called Junior. It provides parallelism and broadcast events communication.

1 Introduction

This paper presents a way to visually create behaviors using a Java-based reactive approach called Icobjs. Icobjs is based on graphical entities with reactive behaviors (the graphical aspect is not bound to any specific modeling language). There are two kinds of icobjs: elementary icobjs which have an elementary behavior and Icobjs builders which allow creating new icobjs by composing behaviors of elementary icobjs using control instructions. Our goal is to give end-user a way to easily create behaviors for entities in a game or more generally in virtual worlds. This paper presents works at the behavioral level of what is studied in [1]. More precisely, a reactive programming and execution model is proposed that fulfills the following features:

- There is a formal deterministic semantics for Junior, the reactive language with which the behaviors are coded. The semantics assures reproducible execution.
- It is expressive enough in order to allow fine control over behaviors and the definition of complex synchronization constraints.
- It allows the construction of complex behaviors by the combination of more elementary ones. One can re-use constructed behaviors to create more complex ones.
- Programs can be added at runtime.
- A large number of events and reactive programs can be executed.

The structure of this paper is as follows: section 2 gives a brief overview of the work around Icobjs and other visual languages allowing building visually autonomous entities. Section 3 gives an overview of the reactive approach and of Junior, the Java-based reactive language on which Icobjs is based. The section 4 describes the new version of Icobjs and the way it works. After presenting some elements of the Icobjs API, we describe how to use Icobjs and how to create complex behaviors using elementary ones.

2 Related Works

2.1 Different visual language

The aim of DSVL is obviously to help users to program applications more easily, more intuitively and/or to hide the syntax of complex language. We will not be exhaustive on it, but only give some examples. First example of this kind of languages, Icon Author allows to create multimedia applications in an easy way. To program applications, one uses a visual environment with no programming or scripting aspects. The language is based around a flowchart of ”parametrable” icons in a window system. Some information on this product can be found at [2].
Java beans (cf. [3]) can be visually composed into composite components, applets, and applications using visual application builder tools. All properties are visually accessible. One can modify java beans (behaviors and appearances) without writing any line of code. The use of introspection mechanisms allows this.
Squeak (cf. [4]), a graphical open language based on SmallTalk-80, provides powerful introspection mechanisms due to its language specificities. It allows drawing entities (called morphs) and then to give them some behaviors by drag-and-drop. There are too many useless features to each morph and this limits the number of objects that can be put together in a simulation.
Finally, we can consider that Integrated Development Environments (IDEs) are DSVL too in a certain way. An IDE indeed provides some features that help users. Visual languages, like Visual Basic, Visual C++... provide some features that help people to design graphical interfaces and to help coding.

2.2 Previous work on Icobjs

There are several versions of Icobjs. The first one is based on reactive scripts built on top of C (see [5]). Thereafter, several Java-based versions were realized to be able to use Icobjs on the Internet and to work on a true object-oriented model (test [6]).
The main problem of those versions comes from the difficulty, for an object-based language, to add dynamically new information, new fields to a class progressively with construction; this is certainly a limitation to modularity. With these Java versions, some behaviors inherit other behaviors that are not necessarily dependent. One falls into cases where icobjs have useless information and behavior parts.
A version of Icobjs, which is based on SugarCubes (cf. [6]), has been used to simulate physical systems (cf. [7]). The goal was to represent complex systems by a set of linked but independent, elementary entities sharing a common time. This gives a very modular model.

3 Reactive approach

The reactive approach proposes a flexible paradigm for programming reactive systems (cf. [8]), especially those that 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 is bound to the notion of instants, and reactive machines whose purpose is to execute reactive instructions in an environment made of instantaneously broadcast events.
SugarCubes and Junior [9] are Java-based languages for programming reactive behaviors. Junior is a descendant of SugarCubes. In this paper, we focus on Junior. Basically, programming with Junior means:

- writing a reactive instruction, which describes an application program.
- 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 increased 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 strong coherency property: during a machine reaction instant, the same event cannot be tested both present and absent, even by two distinct concurrent instructions.
Junior is pure Java. It is provided via an API named Jr[10]. Using Jr, programmers can define reactive instructions and reactive machines, and have possibility to run them. Junior can be seen as a Java programming framework. From this 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 [11] for a description of those problems, and [12] for a comparison of Java threads with the related SugarCubes formalism).

3.1 Reactive instructions

Reactive instructions are state-based statements, activated by reactive machines. Some cyclic instructions are never ending across instants, while others are completely terminated after one or several activations. Reactive instructions are not reentrant, because they have a state. When they are added to the reactive machine, they must be copied, in order to get new execution instances. Reactive instructions are Java objects implementing the `Program` interface. They are built using static methods of the class Jr. Reactive instructions are composed from a small set of basic instructions, control instructions, event instructions, instructions used to interface with Java and migration instructions.

Basic Instructions

- `Nothing()` does nothing. It is the beginning instruction of the reactive machine.
- `Stop()` stops the execution up to the next instant (only its execution branch, not all the program).
- `Seq(A,B)` puts two reactive instructions A and B in sequence. The second is executed when the first is finished. The `Seq` instruction is finished when the two instructions are terminated.
- `Par(A,B)` puts two reactive instructions A and B in parallel. The two instructions begin their execution at the same instant. The `Par` instruction is finished when the two instructions terminate.
- `Repeat(n,A)` and `Loop(A)` allow looping on a reactive instruction A, respectively n times or infinitely.

Control instructions

- `If(cond, A, B)` tests cond. If it is true, then it executes A, else B.
- `When(event, A, B)` is like the If-instruction, but it tests the presence of one event.
- `Until(event, A, B)` executes A until the event arrives, and then branches to B.
- `Control(event, A)` executes A only during instants where event is present.

Instructions interfacing with Java

- `Atom(action)` executes a Java program that implements the `Action` interface. This is an atomic action that possibly performs some interaction with the Java environment. It begins and finishes at the same instant.
- `Link(Object, A)` links reactive instructions to a Java object.

Event instructions

- `Generate(event)` adds event to the environment for the current instant.

- `Generate(event, value)` adds event to the environment for the current instant and associates a value to this generation.
- `Await(event)` stops the execution until the presence of event.
- `Scanner(event, scanAction)` executes, for each occurrence of event, the java action `scanAction` (it is like an `Action`, except that it reacts to an event parameterized by the value associated to the occurrence of event).

Migration instruction

- `Freezable(event, A)` allows freezing A when event is present. It stops the execution of A and puts the residual of A out of the reactive machine. The extracted program can be retrieved and put again in a reactive machine.

3.2 Reactive machine

The role of a reactive machine is to load and execute reactive instructions and to decide ends of instants. The end of an instant is decided when all events are processed and when all programs in the reactive machine are stopped, terminated or waited some events. 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 the next instant.

3.3 Events

Events 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. 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*. The absence of events is decided at the end of an instant, so the reaction to the absence of an event is done at the next instant (see [3] for details).

4 Icobjs

Icobjs is a means to combine reactive behaviors in a visual way, using elementary behaviors and ”Icobjs builders”. The way to create or to model the graphical aspect is not detailed here. The goal of our work is rather to describe elementary behaviors, to make them as most modular as possible and to create mechanisms to combine them easily.

4.1 The API

To solve the problems introduced in Section 2.2, we have realized a new more modular version of the Icobjs API. This version is based on the use of the `ExtensibleObject` interface that gives the means to dynamically add new fields (as for Java beans) and to avoid creating an inheritance tree for elementary behaviors. To add and store dynamically new fields, we use a Java Hashtable. A hash table can contain all types of object, which are accessible by a string identifier. This solution makes the various behaviors much more modular. This solution allows adding, removing and modifying fields or information of involved objects during execution.

```
public interface ExtensibleObject{
    Object getValueOfField(String fieldName);
    void setValueOfField(String fieldName,Object aValue);
    void removeField(String fieldName);
}
```

Our aim is to define a minimal API as general as possible. We will not describe the whole API, but only two main points: the `Icobj` class and the `Workspace` interface. Both implement `ExtensibleObject` so everybody can add some new information and then the environment become more modular. `Icobj` represents all entities in the simulation and `Workspace` is the container of all Icobjs. Each icobj has the following fields:

- a name to identify it.
- a reference towards the workspace that contains it. An icobj can only ”live” in one workspace at a time.
- its zone and its appearance for the graphical part. The zone represents the position of the icobj in the workspace and the area or volume it takes.
- for the behavioral part, two behavioral fields: the Cloneable behavior and the Not-Cloneable one. We chose to make a difference to be more accurate in the description of behaviors. Icobjs builders (cf. Section 4.2) copy the Cloneable field and use it to create more complex behavior. The Not-Cloneable describes a specific behavior that can't be duplicated. This is useful to define appearance in the behavior.

The second main notion is the Workspace Java interface. It represents the virtual world or the simulation. It is the interface with users; it manages and generates, for example, all mouse or keyboard events. It adds or removes behaviors from the reactive machine and makes it react to execute all contained behaviors. The workspace is also in charge of the graphical part. A workspace has the following methods:

- `getName()` gives the identifier of the workspace.
- `registerIcobj(Icobj)` and `destroyIcobj(Icobj)` to add or remove Icobjs.
- `listOfOverlappingIcobj(Zone)` gives the list of all icobjs present in the specified zone of the workspace.
- `getMachine()` gives a link to the reactive machine.
- `needToRender()` and `renderScene()` to refresh the graphical part.
- `getDimensionNumber()` gives the number of dimension of the simulated world (if it is a 2D or 3D world and if the workspace uses X, Y and/or Z axes).

4.2 Building icobjs

There are two kinds of Icobjs: elementary ones and the Icobjs builders that are the means to create new icobjs by composing behaviors of the others ones. All elementary icobjs and Icobjs builders form a kind of toolbox to build new icobjs with a more complex behavior (see Figure 2).
Elementary icobjs are icobjs whose behavior is an atomic action. Constructed icobjs can be considered as elementary icobjs, because they can be re-used for new construction. On the figure 1, two elementary behaviors are represented: the behavior of the first one is to make the icobj move right and the other to make it move down. The constructed icobj has a behavior, which makes the icobj move to the right and then down.
Icobjs builders are the way to use Icobjs at the applicative level and thus to build complex behaviors starting from the elementary ones. A builder is an icobj whose behavior allows building other behaviors, to put elementary behaviors together and add some control structure in order to create a more complex one. Here are detailed some builders with some related icobjs:

 This represents the basic builder. It allows users to create new icobjs and to give them a behavior. Figure 1 and figure 2 proposes an example of its using. At each click:

- If another icobjs is in the same zone as the builder (this means that the zone of the builder and the zone of the icobj intersect), the builder clones the behavior of this icobj and puts it in sequence with what it had previously cloned. Figure 1 show a scenario where, first, the builder is put on

the icobj ”right” and clones its behavior (go right). Then it is put on the icobj ”down”, clones its behavior (go down) and put it in sequence with the first cloned behavior. Finally, it goes on a space where there is no icobj and create a new one whose behavior is to go right and then down.

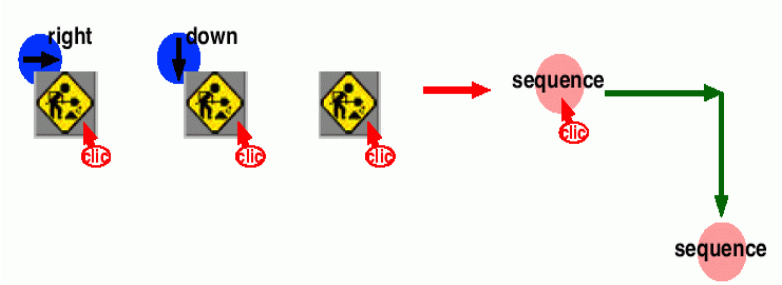

Figure 1: Construction of a sequence of two behaviors

- If several icobjs are in the same zone, the builder makes a copy of the behavior of all icobjs, puts them in parallel and puts this behavior in sequence with what it had previously cloned. As seen on figure 2, the builder clones the behaviors of the two icobjs (go right and go down) and creates an new icobj whose behavior is to go right and down at the same time.

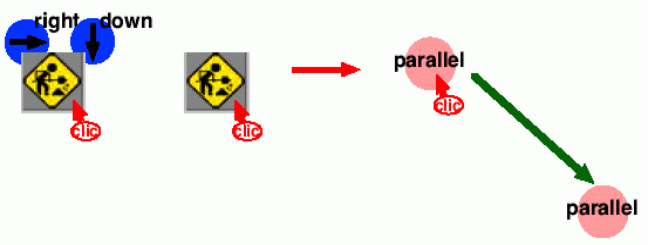

Figure 2: parallelization of two behaviors

- Finally, if no icobj is in the same zone than the builder, the construction ends and a new icobj is created. Its behavior is what the builder had previously cloned. Now, another construction can start again with the default initial behavior: `Nothing`.

 This represents the event emission builder. It corresponds to the reactive instruction `Generate`. The name of the emitted event can be changed via the Textfield builder described below. If one puts this builder in the same zone than the Textfield builder and clicks on it, the name of the event becomes the character string written in the Textfield.

 This builder, the Event-Waiting one, is the counterpart of the previous one. It corresponds to the reactive instruction `Await`. As previously, the name of the event can be changed via the Textfield builder.

 This Icobjs builder allows preempting behaviors as the reactive instruction `Until`. The event which starts the preemption can be modified via the Textfield builder. This builder works in the same way as the creation builder, except that the resulted behavior can be preempted on reception of the associated event.


This builder corresponds to the reactive instruction `Control` that filters the control according to the presence of a specified event. The controlled behavior is executed only when the event is present. The use of this builder is identical to the use of the preemption builder.

The Textfield icobj allows changing the event parameter used by builders. It also allows giving a name to the new icobj created with the creation builder.


This is another kind of builder because it modifies the behavior of existing icobjs. It gives the mouse control behavior to the Icobjs on which it is applied.


This is a builder that produces a `WorkspaceIcobj`. It is an icobj with a workspace as appearance. This new workspace is totally independent of its containers. This creates a new clean environment in which all generated events don't affect its container workspace and vice-versa. A drag-and-drop mechanism allows to move icobjs from a workspace to another.

Figure 3: Icobjs factory

5 Conclusion and perspectives

Icobjs is a means that enables to build reactive behaviors by visual composition. This system makes it possible to program the behaviors of entities by a rather simple means, without needing to learn a "real" programming language, but a lighter version of it. We create some elementary behaviors that are general enough to pass in 2D and 3D worlds while keeping the same behavior.
Another version of Icobjs, carried out within the framework of project IST-PING (cf. [13]) and dedicated to distributed simulations was realized. The goal of this project was to define a platform for large-scale interactive multi- participants applications on the Internet, especially game. This version was a way to

make a kind of collaborative work by giving the capacity to create behaviors, which can be exported on all simulations. Each one can create a part of the distributed virtual world.
A tool allowing the visualization of terms of the Ambients calculus (cf. [14]) has been realized using the Icobjs API. It uses the model of Mobile Ambients, Robust Ambients and Controlled Ambients and represents it in a funny way. It is accessible on the website of the MIMOSA project (cf. [15]]).

Next work will define a way to modify a behavior that was built or to know exactly what it does. Therefore, it is planned to be able visualizing this behavior in the form of a tree whose leaves are the elementary behaviors. Using this tree would allow removing certain parts of the behavior or to freeze the execution of it. Introspection mechanisms will be added and this will provide a mean to parameterize elementary behaviors.
For the moment, it is not possible to program, visually and at runtime, new atomic behaviors. We plan to use a kind of console that would help to write some new Java code, compile these new creations and add it at runtime in the simulation.
Finally, we want to add them a mechanism to type the field present in the Hashtable (indeed, for the moment, each elementary behavior must verify the type of the recovered fields).

References

[1] Frédéric Boussinot, Jean-Ferdinand Susini, Frédéric Dang Tran, and Laurent Hazard. A reactive behavior framework for dynamic virtual worlds. In *Proceedings of the sixth international conference on 3D Web technology*, pages 69–75. ACM Press, 2001.

[2] http://www.pps.com.au/iconauthor.html.

[3] http://java.sun.com/docs/book/tutorial/javabeans/index.html.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *Proc. of the ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, October.

[5] Frédéric Boussinot. Icobj programming. Technical report, Octobre 1996.

[6] http://www-sop.inria.fr/mimosa/rp/Icobjs.

[7] http://www-sop.inria.fr/mimosa/rp/SimulationInPhysics/.

[8] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and Models of Concurrent Systems*, NATO ASI Series(13):477–498, 1985.

[9] Laurent Hazard, Jean-Ferdinand Susini, and Frédéric Boussinot. The junior reactive kernel. Technical Report RR-3732.

[10] Laurent Hazard, Jean-Ferdinand Susini, and Frédéric Boussinot. Programming with junior. available at http://www-sop.inria.fr/mimosa/rp/Junior, 2000.

[11] JavaSoft. Why javasoft is deprecating thread.stop, thread.suspend, thread.resume and runtime.runfinalizersonexit.

[12] Frédéric Boussinot and Jean-Ferdy Susini. Java threads and sugarcubes. *Software–Practice & Experience*, 30(5):545–566, 2000.

[13] http://www.pingproject.org.

[14] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[15] http://www-sop.inria.fr/mimosa/ambicobjs/.