INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Icobj Programming

Frédéric Boussinot

N° 3028

Octobre 1996

THÈME 1

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Icobj Programming

Frédéric Boussinot

Thème 1 — Réseaux et systèmes
Projet MEIJE

Rapport de recherche n ° 3028 — Octobre 1996 — 18 pages

Abstract: A simple and fully graphical programming method is presented, using a powerful means to combine behaviors. This programming is based on the notion of an "icobj" which has a behavioral aspect ("object" part), a graphical aspect ("icon" part), with an "animation" aspect. Icobj programming provides parallelism, broadcast event communication and migration through the network. An experimental system based on this approach is described in details. Its implementation with reactive scripts is also presented.

Key-words: Parallelism, Icon, Script, Reactive programming

(Résumé : tsvp)

EMP-CMA, B.P. 207, F-06904 Sophia Antipolis cedex

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

La programmation par icobjs

Résumé : On présente une programmation simplifiée, complètement graphique, avec un mécanisme puissant de combinaison de comportements. Cette programmation repose sur une notion d"'icobj" qui regroupe dans une même entité un aspect comportemental (caractéristique "objet"), un aspect graphique (caractéristique "icône") et un aspect "animation". Cette programmation par icobjs permet le parallélisme, la communication par événements diffusés ainsi que la migration à travers le réseau. On décrit en détails un système expérimental fondé sur cette approche ainsi que son implémentation à base de scripts réactifs.

Mots-clé : Parallélisme, icône, script, programmation réactive

Icobj Programming[1]

1 Introduction

In ordinary life every one programs household equipments (washing machines, ovens, etc.), digital watches or video recorders. In this context "programming" simply means to perform some sequence of elementary actions (to put a tape into a deck, to push a button, to set a cursor) in a fixed order, possibly respecting some constraints (to push a button simultaneously with another one, or in a fixed delay). Regarding personal computers most users only deal with software applications such as word processors, spreadsheets or Web navigators, which do not require actual writing of programs. The writing of program is left to professional programmers as a complex task which needs training and knowledge of programming languages.

The complexity of usual domestic objects obviously tends to increase (some audio player instruction sets contain more than 70 pages !). Software tools are becoming more and more complex and have more and more interactions; for example, a word processor can be used jointly with a spreadsheet to produce an accounting document in a rather simple way. Moreover, the advent of Internet and of the Web introduces new interaction possibilities.

In the technical computing domain significant effort is made, based in particular on the use of graphics, to reduce complexity of programmer's work. Graphical man/machine interfaces have become essential and a lot of graphical programming environments are available to ease the developer's job. However, concerning the specific program writing activity there are few graphical tools, which all follow the same path by providing a *graphical syntax* closely linked to the textual syntax. Actually, they are a kind of "graphical dressing" for the program syntax[2]. For example, instead of writing: "`A;B`" one draws figure 1, in which the sequence operator ";" is symbolized by an arrow.

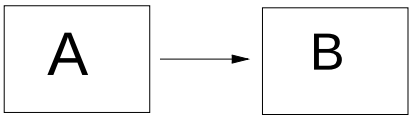

Figure 1: Graphical Representation of "`A;B`"

In case `A` is recursively made of the sequence "`C;D`" one gets the hierarchical drawing of figure 2.

In such an approach, a drawing associated with a program contains all the information needed to recover the program text. This deep connection between texts and drawings possibly explains the relative lack of success of this approach: to program in the graphical framework is as difficult as to program using texts; graphical programming is just changing the representation, which is actually a matter of taste.

[1]Work supported by France Télécom/Cnet and Soft Mountain, Grant 93 1B 141, #506

[2]The case of specification languages is not considered here since there the goal is not to write programs.

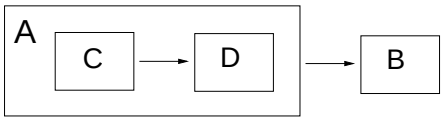

Figure 2: Graphical Representation of "`(C;D);B`"

One may wonder whether the increasing complexity of currently used objects, including software tools, will still allow users to avoid some kind of real programming in the future. Possibly, Web users will also ask for new tools extending navigators which will need some kind of advanced programming (based for example on programming primitives to follow hypertext links or to migrate agents through the Net).

It seems out of the question to propose a textual based programming formalism to the larger audience outside computer scientists. Instead, we need a simple and natural programming approach which can be used without training (or at least, with a minimum of it) in the current life, and specially fitted for Internet.

From the user's point of view, such an approach should:

1. be simple: simple problems should be simply solved (which does not mean, of course, that complex problems have necessarily to be solved in the same way).
2. be graphical. But graphics should not be a way to mask syntax to users (as in figures 1 and 2). Actually, there should not exist any grammar, nor text to write, even in a graphical form.
3. not be based on translations into textual languages; objects built should be *run* directly.
4. be natural: for example, behaviors should be sequentialized only when there is a logical reason to do so.
5. allow the reuse of already built objects when building new ones.

In technical terms, we need a graphical approach (point 2), interpreted (point 3), concurrent (to avoid sequentiality when parallelism is natural, points 1 and 4), object-based (point 5), and with a clear and simple semantics (point 1). To define a formalism with these characteristics is the goal of "icobj programming" described in this paper. The basic idea is as follows: from the user's point of view, programming is an unpleasant activity because tools provided to perform it are not powerful enough, not because it is too complex a task. The point is thus to design a powerful programming model able to deal with complex problems in a simple way, not to weaken an existing one to propose it to a larger audience.

The rest of the paper has the following structure: section 2 describes the main characteristics of icobj programming. An experimental system of icobjs is described in section 3 which ends with a list of demos. Finally, implementation of this system, based on reactive scripts, is given in section 4.

2 Icobj Programming

We define new objects called icobjs (for "*iconic object*"), which are animated graphics. An icobj consists of an object part that determines its behavior and an icon part that determines its graphical look. We provide a simple and natural way to *compose behaviors* through operations related to objects look and animation. To sum up, an icobj is a behavior with animated graphics.

The animation aspect is essential. We build some sorts of "cartoons" in which animation reflects object behavior. This contrasts with classical graphical or iconic programming where one draws programs made of static graphical components, which can be icons.

Let us consider the example of a Web search engine. An information search has a behavioral aspect (the chosen filter), a localization aspect (the engine address), and a dynamical aspect (the return of information). A natural way to express a search could be to create an icobj whose graphical aspect would define the filter, and to activate it by placing it on some "map" of possible engines; finally, the icobj would move in a place symbolizing the user when the information is found.

Animation is especially natural for migration as one can visualize agents moving across the network.

2.1 The Notion of an Icobj

To be more precise, an icobj has several aspects:

- Its graphical look on the screen.
- Its scope of influence, which is the graphical area in which the icobj may interact with other icobjs.
- Its behavior, which is run when the icobj is activated.

An icobj can be created, activated or destroyed. An icobj behavior combines its own behavior with interactions from other icobjs. Execution of an icobj may change its look, animate it, or act on icobjs which are in its scope of influence. In some cases, creation of an icobj needs extra parameters. Some icobjs are called "creators": their main purpose is to create others icobjs.

2.2 Behavior Combination

The essential aspect of icobjs is the possibility to combine their behaviors. Let us return to the Web search engine example. Let us assume an icobj whose life-time is limited to a fixed delay, and which destroys itself when the delay is over. By combining it with the search icobj, one gets a new combined icobj which behaves as the search icobj but self-destroys if the searching takes too much time.

The icobj behavior combination must verify some intuitive properties. For example, let us combine the two icobjs `left` and `right` whose behaviors consist in moving in the direction corresponding to their name. One expects the combined icobj to be at the same place before

and after execution. There are two ways to get this result: in the first one, the combined icobj executes the two behaviors in sequence, and combination means vectorial sum. The second way consists in not moving the combined icobj at all.

Let us take a more mathematical example. Icobj `sin` trajectory is defined by $y = sine(\theta)$, θ from 0 to 2π. Icobj `cos` trajectory is defined by $x = cosine(\theta)$, θ from 0 to 2π. A circle is of course the expected combination of these two icobjs; informally: *sin + cos = circle*. Note that combination definitely does not mean vectorial sum. This example shows the difficulty of combination: there is the need for an underlying coordination between icobjs during execution. To combine icobjs thus implies a means to *synchronize* them.

2.3 Communication between Icobjs

Icobjs can communicate by interactions through their scopes of influence. However, icobjs whose scopes of influences are not overlapping still have the possibility to communicate by events. Events, which are non-graphical objects, are awaited or generated by icobjs during execution and it is possible to control icobjs by events which are generated by other ones. For example, the same event can fire several icobjs, independently of their graphical representations. In the search engine example, suppose one runs the same search simultaneously on several engines. Using an event, one can for example stop all the searches as soon as the first one succeeds, independently of the actual icobj positions on screen.

The notion of an event is very powerful: generation of an event fires all the icobjs which are awaiting it. Icobj senders and receivers of a common event need not know each others. This kind of communication is simpler than "rendez vous" based communications, which are 1-1 and imply that each part has a knowledge of the other. One can add a new icobj waiting for an event without changing other icobjs. Event-based communication means *broadcast*: all icobjs waiting for an event receive it simultaneously when it is generated, just as in radio broadcasting all receivers receive the same informations at the same instants. This communication implies a certain form of simultaneity: the firing of icobjs waiting for an event is simultaneous with its generation.

2.4 The Reactive Approach

Using the *reactive approach*[2][3] in which all icobjs share a logical global clock defining *instants*, one can simply implement the synchronization needed to combine icobjs, as well as broadcast events:

- All icobjs must have finished to execute the current part of their behavior before the next instant takes place. Thus, the end of the current instant is a synchronizing point for all the icobjs in the system.
- The notion of an instant gives meaning to the simultaneity of event generations and waitings, and makes it possible to implement broadcast communication. An event is present at a given instant if any of the system icobjs generates it during that instant;

[3] This approach is strongly related to *synchronous languages* and especially to the Esterel language. See [1] for an overview of these formalisms.

otherwise, the event is absent for that instant. Thus, icobjs fired by generation of a given event at a given instant are precisely known.

3 Description of an Icobj System

We present an experimental icobj system designed as a play framework to experiment with icobjs.

In our system we focus on the "cartoon" aspect of icobjs. In particular, in the examples given, behavioral and animation aspects are merged and emphasis is put on building complex animations out of simpler ones.

The graphical aspects are reduced to a minimum, only using basic icons (colored circles or squares) with associated labels (generally the icobj name). Technically, the system is implemented on top of Tcl/Tk[5] (which the user does not need to know, to use the system).

The system is available on Internet (presently, only for Unix SunOs4 platforms) at URL `http://cma.cma.fr/RC/rc-project.html`.

The system is shown on figure 3 and is made of two windows: the one at the bottom is a control panel, described in 3.2; it contains icobj creators. The second window, at the top, is the place where icobjs are created and animated.

Figure 3: An icobj system

3.1 Graphics and User Interface

We use a 3-button mouse. Icobjs are activated by the left button; the center button is used to move icobjs; the right button changes icobj sizes.

Graphically, an icobj is a simple geometrical frame (circle, rectangle) and its scope of influence is associated to its "bounding box". More precisely, an icobj A has an influence on an icobj B if the icon of B overlaps with the bounding box of A. In the following, to simplify, one does not distinguish anymore the scope of influence of an icobj and its bounding box. Figure 4 shows three `delete` icobjs with distinct sizes, whose scopes of influence are surrounded by rectangles to make them visible (sizes of icobjs and of their scopes of influence are changed with the right button).

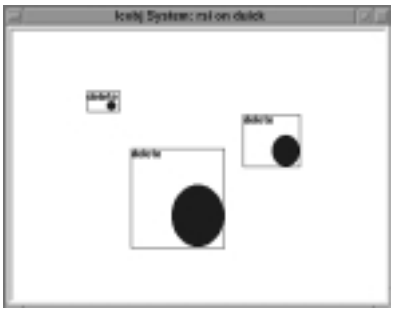

Figure 4: Three Icobjs `delete` with their Scopes of Influence

On figure 5, the two icobjs `right` and `down` are placed in the scope of influence of the icobj `delete`. The three icobjs will simultaneously disappear when `delete` is activated by clicking the left button.

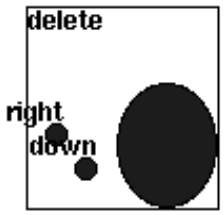

Figure 5: Scopes of Influence

3.2 The Control Panel

The control panel is shown on figure 6. It contains mainly creator icobjs and created icobjs are placed in the animation window. We now consider the individual control icobjs in turn.

Exit

Activating the `exit` icobj exits the system.

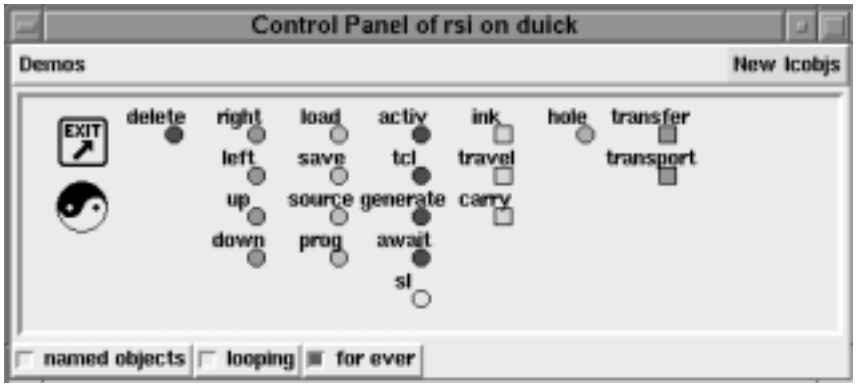

Figure 6: The Icobj System Control Panel

Joker

The `joker` icobj represented by the "yinyang" icon[4] is the central one in the system. Mainly, it provides behavior composition. It has several uses, in particular:

- to give an icobj a name;
- to create a "clone", that is a copy of an existing icobj;
- to put icobj behaviors in sequence;
- to combine icobj behaviors;
- to create cyclic icobj behaviors.

The full description of `joker` is given in the next section.

Delete

Each time it is activated, the `delete` icobj creates a destructor icobj in the animation window. When activated, this destructor icobj destroys all the elements present in its scope of influence, including itself.

Right, left, up, down

At each activation, each of these control icobjs creates a new icobj with the same name in the animation window. When activated for the first time, the new icobj moves for a short distance in the corresponding direction; then it terminates, while its icon remains inactive on screen.

Load, save, source

With the `save` icobj one can store in a file an icobj which can be loaded with `load`. When activated, these two icobjs ask the user for the name of the icobj to store or to load. Names

[4] For lack of a joker icon...

are entered through a dialog box appearing at the bottom right of the control panel. The `source` icobj is used to load Tcl/Tk command files.

Prog

The `prog` icobj is used mainly for debug purposes. It displays a reactive script (see 4.1) at the bottom of the animation window; this script describes the behavior of the icobj whose name is given by the user, and it is actually the semantics of this icobj.

Activ

The `activ` icobj activates all the icobjs present in its scope of influence, and then terminates. Figure 7 shows four icobjs `right`, `left`, `up` and `down` influenced by an icobj `activ` put in the middle. Figure 8 shows the result of the activation of `activ` (whose color has changed, after activation).

Figure 7: Before activation of `activ`

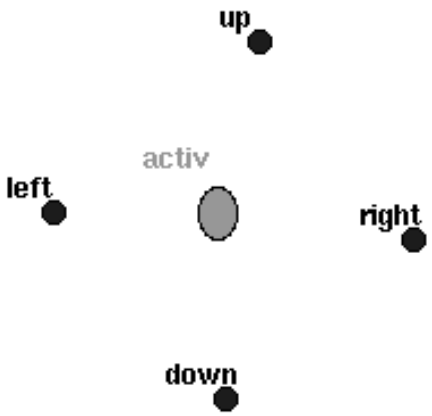

Figure 8: After activation of `activ`

Tcl

The `tcl` icobj prompts for a Tcl command, executes it when activated, and then terminates.

Generate, await

The `generate` and `await` icobjs deal with events. Icobj `await` creates a new icobj "`await E`" after the user provides the event name `E`. When activated, "`await E`" waits for the event `E` and terminates when `E` occurs. Icobj `generate` creates a new icobj "`generate E`" after the user provides the event name `E`. When activated, "`generate E`" immediately generates `E` and terminates.

Sl

In response to activation, icobj `sl` creates a new set of icobjs, corresponding to a subset[5] of reactive scripts presented in 4.1. We do not consider this in details and refer to [4] for more information.

Ink, travel, carry

When first activated, these three icobjs store a path drawn by the user with the mouse. Then, at next activation:

- `ink` redraws the path;
- `travel` moves its icon along the path;
- `carry` transports along the path all icobjs inside its scope of influence.

Hole

The `hole` icobj creates a new icobj which, when activated, attracts towards its center all icobjs in its scope of influence. The scope of influence is here a special one: it is a circle whose radius increases with each activation. The behavior never terminates and it runs at each instant (to stop it, one must use the `delete` icobj).

Transfer, transport

The two `transfer` and `transport` icobjs are used for migration. The user first provides a machine name and a remote reactive script interpretor (see 4.1) running on it. When activated, `transfer` migrates all the icobjs which are in its scope of influence to the remote interpretor. When activated, the `transport` icobj is the only icobj which migrates to the remote interpretor. Examples of uses are given in section 4.2.

[5] Actually, the synchronous language SL[3].

3.3 Joker

The main purpose of the `joker` icobj (represented by the "yinyang" icon) it to combine icobj behaviors. Three "radio buttons" at the bottom of the control panel ("`named objects`", "`looping`", and "`for ever`") are used to parametrize the combinations.

In the standard situation, activation of `joker` creates a new icobj whose icon is a black circle in the animation window. This new icobj has name `jok_`n, where n is a new integer. At each activation, the joker stores the combination of all the icobjs which are in its scope of influence; the joker construction ends when it is activated while its scope of influence is empty; then, the joker turns to blue and the sequence of all the previously stored combinations becomes its new behavior. Now, it will execute its behavior when activated.

When "`named objects`" is on, the joker is created with a user-given name provided in a special area of the control panel. When "`looping`" is on, the joker behavior becomes cyclic. If "`for ever`" is on, the loop is infinite; otherwise, the user has to enter the number of cycles.

Creation of a clone is achieved by placing a joker on the icobj to be copied, then by activating it, and finally by activating it a second time in a place where there is no other icobj. The result will indeed be a new icobj which behaves as the old one. The clone has a new name given by the user if "`named objects`" was on.

Examples

Here are some examples of joker uses:

1. Create 4 icobjs `right`, `down`, `left`, and `up`. Move these icobjs so that their scopes of influence are disjoint.
2. Create a joker. Move and activate it on each of the four previous icobjs. Finally, activate the joker at a place where there is no other icobj, which ends its definition.
3. Activate the joker. As one may check it then draws a square and terminates.

Same thing, but with "`looping`" on; the joker behavior becomes cyclic.

Same thing, but add to the sequence a `delete` icobj; then the joker self-destroys after the square is drawn.

If during the second phase the joker overlaps `right` and `down` at first activation, and `left` and `up` at second, then one gets a moving following the bottom/right, then the left/up diagonals.

3.4 Demos

The "*Demos*" button (at the left on the top of the control panel) provides a list of demos using the icobjs defined in the control panel. We now consider these demos in turn and, for each of them, suggest some possible user extensions.

Hello world

This demo creates two icobjs `hello` and `world`, built with `ink`. In response to activation, they write the corresponding words (this is shown on figure 3).

Suggestion: create a new icobj which writes the sentence "hello world", by putting the two previous icobjs in sequence with a joker.

Up/down

This demo creates two icobjs in the animation window. The first one, named `x`, has a cyclic top/down behavior. Each time event `change` is generated, the behavior switches back and forth from top/down to right/left. Icobj `ch` generates event `change` at each activation.

Suggestion: create several clones of `x` and activate them. They all simultaneously change their behaviors each time `ch` is activated. This is an illustration of event broadcast.

Cycle

In this demo, two `hole` icobjs are created and transported, using `carry`. Then, an `hcycle` icobj is created and activated; it travels forever from one hole to the other.

Suggestion: destroy one hole during the travel, then the other; observe `hcycle` drifting (`hole` icobjs were preventing the drift, by centering `hcycle` movings).

Pulse

This demo creates four icobjs `R`, `L`, `U`, and `D`, and a fifth one, `A`, which destroys itself (using `delete`) after a fixed number `N` of instants. These five icobjs are transported by an icobj `C`, and are simultaneously activated. The system made of these icobjs then begins to pulse. The system is strongly synchronized: all icobjs are destroyed if `A` self-destructs too soon, because `N` is too small. In the demo `N` has value 10. Figure 9 shows the situation just before `A` self-destructs (icobjs `A` and `C` are stacked at the center).

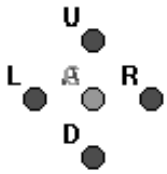

Figure 9: The "pulse" Demo

Suggestion: change the value of `N`, using icobj `tcl`, by producing the new icobj "`set N 5`", then executing it. Try also the demo with the value 40.

Migration

Demo "migration" shows an example of transportable object. First, a remote reactive script interpretor `rem-rsi-tk` (see 4.2) must be run. An icobj `migr` is created after the user enters the machine and remote interpretor names. Once activated, this icobj draws a rectangle, then migrates on the remote interpretor, and finally draws an other rectangle on it.

Suggestion: let an icobj with the same behavior go for ever back and forth between two interpretors.

3.5 New Icobjs

The button "*New Icobjs*" (at the right on the top of the control panel) provides the system with new icobjs, directly built out from reactive scripts (see 4.1).

Sine/cosine

One has two new creators in the control panel; by activating them, one creates the icobjs `sin` and `cos` described in section 2.2. One easily sees that they verify the equation *sin* + *cos* = *circle* by running a joker which combines their behaviors.

Suggestion: combine several instances of `sin` and `cos`.

Border/brush

Two new creators are introduced in the control panel; the first one creates a `border` icobj which beeps at each instant, when placed next to the side of the animation window. Moreover, when placed near a window border, `border` generates the corresponding event (`right` for the right border, `left` for the left one, and so on). The second creator produces a `brush` icobj which takes away all the icobjs it finds when traveling from one window side to the opposite.

Suggestion: create a sequence of "`await right`" and of `delete` and combine it with a `border` and a `brush`. Place the new icobj on the left side after resizing it. When activated, it drags all the icobjs on its way up to the right side, and then destroys them (including itself). Figure 10 shows this icobj named `clean` created that way and ready to run (joker `jok_17` is the sequence of "`await right`" and `delete`).

Reflex game

A `game` icobj is created. When activated, it changes its associated label to `WAIT`; then after a delay it changes it again to `GO!` and starts to count elapsed instants before the user activates it another time. The result becomes the label and the game returns to its initial state. Moreover a cheat attempt is detected whenever the user anticipates `GO!` and activates the game while the label is `WAIT`.

Suggestion: combine `game` with a moving icobj; it may turn to be quite hard to play with it...

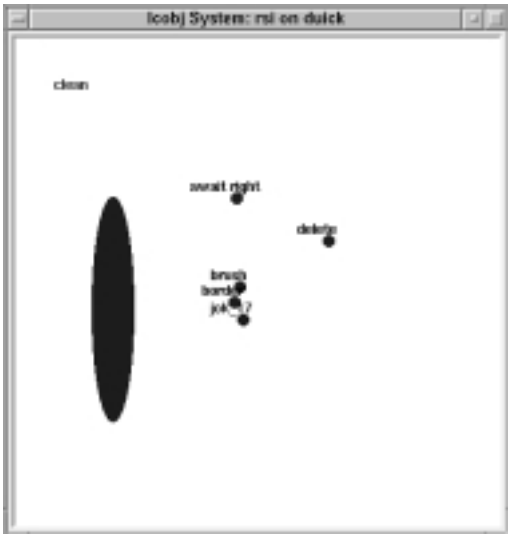

Figure 10: Icobj `clean` ready to run

Planet/meteor

Two creator icobjs are produced. `planet` creates an icobj similar to `hole`, except that it attracts only `meteor`[6] icobjs and destroys them when they touch it. An icobj created by `meteor` tends to maintain its speed. Its initial speed is defined by the user after the first activation, by moving it slightly in the good direction, with the middle button (warning: this is a sensitive driving, to avoid to give the meteor a speed too fast).

The goal is to put meteors into orbit, around a planet. Figure 11 shows two meteors around the same planet placed at the center of the window.

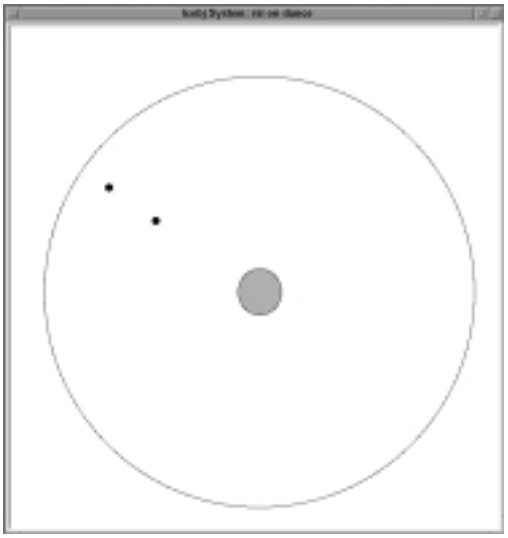

Figure 11: Two meteors in orbit

[6] More precisely, icobjs whose names start with the character string "meteor"

Suggestion: after succeeding to put a meteor in orbit, destroy the planet; then, the meteor follows a straight line as the planet attraction has disappeared. Much more difficult: create two icobjs named `meteor1` and `meteor2` each of them combining a planet and a meteor; then activate both and make them turn together, one around the other...

4 Icobj Implementation

Reactive scripts[4] are used to define the semantics of icobjs (their behaviors), and also to implement them. Note that users of the system defined in the previous section do not have to know anything about these scripts (even their existence). We first quickly introduces reactive scripts then describes the implementation.

4.1 Reactive Scripts

Reactive scripts are programs using broadcast events. A reactive script interpretor stores a set of scripts which are waiting for events (`await` statement), and it fires them accordingly to the generations of events (`generate` statement). Such an interpretor is an example of a parallel and dynamic program: it accepts new commands at any time and executes them in parallel with those which are already stored.

Events are broadcast: all scripts waiting for an event are simultaneously fired (in the same interpretor reaction), as soon as it is generated. Finally, the life time of an event is the interpretor reaction during which it is generated.

Reactive scripts can be put in sequence (with ";") or in parallel (with "||"), and finite or infinite loops can be defined (`loop` keyword). The `stop` statement stops execution for the current reaction and `next` forces the interpretor to immediately restart for the next reaction, as soon as the current one is over.

External statements, such as assignments or procedure calls, are always put between braces; in the following their syntax is the one of Tcl/Tk.

Reactive scripts are structured in modules (`behavior` and `run` keywords) and in objects (`object` and `methods` keywords). An object is a script with a name to which it is possible to attach methods which are scripts run when explicit orders are sent to them (`send` statement). An object can be frozen (`freeze` keyword): in this case its execution is stopped and a new script corresponding to "what remains to be done" is produced (and assigned to a Tcl variable).

4.2 Implementation with rsi-tk

The icobj system is implemented with the `rsi-tk` reactive script interpretor, described in [4] and built on top of Tcl/Tk. It is made of about 1500 lines of Tcl/Tk code and 300 lines of reactive scripts.

The behavior of the icobj named `x` has the form "`object x ... end`" and this is the content of the Tcl variable `BEHAVx`. The `prog` icobj introduced in 3.2 prints this variable. The graphical representation of `x` is a Tk object whose name is also `x`.

Joker

A joker is implemented as a sequence of parallel statements which are the behaviors of the influenced icobjs. Thus, the reactive script associated to a joker has the form:

$$(i_1^1||...||i_n^1);...;(i_1^p||...||i_k^p)$$

In case of a cyclic joker, this statement is put into a loop (finite or infinite, depending on the status of "`for ever`").

Reactive Scripts

Reactive scripts associated to icobjs `generate` and `await`, as well as those got from the icobj `sl`, are in direct correspondence with their definition (see [4] for a precise description).

Migration

To transfer an icobj means to freeze it, then to get the residual script, and finally to send it with its icon to the remote `rem-rsi-tk` interpretor[7], using a command named `send-rsi`. Here is a suggestion of use:

- Run a `rem-rsi-tk` interpretor with name `RSI` on the machine `remote` and load the icobjs "sin/cos" (by "*New Icobjs*").
- Create a `rsi-tk` interpretor and a joker `circle` which combines `sin` and `cos`, and activate it.
- Create a `transfer` icobj on `RSI` and `remote`, and activate it while `circle` crosses over it. Icobj `circle` is then transferred on `RSI`.

Using the `transport` icobj, one can build "transportable agents" which migrate when they decide to do so. Here is an example of such an agent, based on the previous `circle` icobj:

- Create a `transport` on `RSI` and `remote`, named `trans`.
- Put `circle`, `trans`, and `circle` in sequence, then run the joker. It draws a circle on the current interpretor, then migrates on the remote one, and finally draws another circle.

5 Conclusion

The notion of an icobj with the following characteristics has been defined:

- There is no syntax for icobjs, which are exclusively used through graphics.
- There is no intermediate translation: an icobj can be run as early as it is created (nothing to compile; this is an interpretor based approach).

[7] `rem-rsi-tk` is the version of `rsi-tk` which can be used through the network

- Icobjs can be put in sequence, in parallel, or inside loops, in a natural way: actually a single icobj performs all these actions.
- Broadcast events can be used to control icobjs, which gives them a powerful means of communication.
- Icobjs have a clear and immediate semantics, based on reactive scripts.
- Icobjs are modular and can be reused without constraints.
- Icobjs can migrate through the network in a simple way, including as transportable objects.

Icobjs are based on the reactive approach and icobj behaviors are reactive scripts. A reactive script interpretor on top of Tcl/Tk has been used to implement an experimental icobj system. Several demos are provided to show what icobjs are. A set of icobjs directly built out of reactive scripts is also given; a very meaningful example consists in combining two icobjs sine and cosine, to get a circle.

Icobjs induce a new programming style, which seems to be well suited to the Web. It would for example be interesting to embed a reactive script interpretor into a Web navigator, to allow the use of icobjs to program searches through the Net. It is also possible to associate a reactive script interpretor to servers, in order to build html pages by icobj programming.

Acknowledgments

Many thanks to Gérard Berry, Laurent Hazard, Valérie Roy, Robert de Simone, and Jean-Ferdinand Susini for their remarks on a first version of the paper.

References

[1] Another Look at Real-Time Programming. Proceedings of the IEEE, vol. 79, no 9, September 1991.

[2] *Reactive-C Project*, http://cma.cma.fr/RC/rc-project.html.

[3] F. Boussinot and R. de Simone. The SL Synchronous Language. *IEEE Transactions on Software Engineering*, 22(4):256–266, April 1996.

[4] Frédéric Boussinot and Laurent Hazard. Reactive scripts. In *Proc. RTCSA'96, Seoul.* IEEE, October 1996.

[5] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399