INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Reactive Scripts

Frédéric Boussinot , Laurent Hazard

N° 2868

Avril 1996

THÈME 1

Rapport de recherche

ISSN 0249-6399

Reactive Scripts

Frédéric Boussinot , Laurent Hazard

Thème 1 — Réseaux et systèmes
Projet MEIJE

Rapport de recherche n° 2868 — Avril 1996 — 35 pages

Abstract: A reactive script interpretor is a broadcast event-driven interpretor which can react to several commands in parallel. The basic principle is that absence of an event cannot be decided before the end of the current interpretor reaction. Generating events and waiting for occurrence of events are the basic commands which are composed in several ways to build complex behaviors. Moreover, one can also define objects with associated methods run when a nonblocking order is sent to them. Method execution is immediate (in the same interpretor reaction as the order) and a method can be executed at most once during each reaction. Reactive script interpretors are implemented using the Reactive-C language. Finally, it is shown that reactive scripts are a mix of two formalisms: the SL synchronous language, and the ROM Reactive Object Model.

Key-words: Parallelism, Script, Interpretor, Reactive programming, Object

(Résumé : tsvp)

EMP-CMA, B.P. 207, F-06904 Sophia Antipolis cedex
France-Télécom/Cnet PAA/TSA, 38-40 avenue du Général Leclerc, F-92131 Issy-Les-Moulineaux

Scripts Réactifs

Résumé : Un interpréteur de scripts réactifs utilise des événements diffusés et exécute un ensemble de commandes parallèles. Le principe de base est que l'absence d'un événement ne peut être décidée avant la fin de la réaction courante. La génération et l'attente d'evénements sont les commandes élémentaires, que l'on compose pour construire des réactions complexes. On peut également définir des objets et leur associer des méthodes exécutées lorsque les ordres non-bloquants leurs sont donnés. L'exécution des méthodes est immédiate (elle s'effectue dans la même réaction que l'ordre) et une méthode ne peut être exécutée plus d'une fois pendant une réaction. Les interpréteurs de scripts réactifs sont implémentés en utilisant le langage Reactive-C. Finalement, on montre que les scripts réactifs sont un mélange de deux formalismes : le langage synchrone SL et le modèle des objets réactifs ROM.

Mots-clé : Parallelisme, Script, Interpreteurr, Programmation réactive, Objet

Reactive Scripts[1]

A Language of Reactive Scripts

A *script* is a program represented as a character string and intended to be run by an *interpretor*. There exist a lot of script languages, for example the many Unix "shell" languages. In a script based approach, emphasis is put on programming ease; for example, variables need not to be declared before used, and their types can be changed at will. Generally, interpretors implement infinite *loops* that wait for a command, run it, and then wait for the next command; thus, interpretors are sequential programs which run one command at a time. In the following we shall say that an interpretor *reacts* to a command when it runs it.

In this paper, we consider event-driven interpretors which stores several commands awaiting for events to occur, and fires them accordingly to their arrival. Such an interpretor is an example of a *parallel and dynamic* program: it does not need to have finished to execute the current command to accept new ones, and each new command stored is run in parallel with the others. Actually, the interpretor manages a global parallel program which may be dynamically changed by addition of new commands; it keeps the state of the parallel program and continues to execute it each time it is run. Moreover, events are *broadcast*, meaning that all commands waiting for the same event are all simultaneously fired, as soon as the event occurs (that is, they are all executed during the same interpretor reaction).

Thus, we are considering *broadcast event-driven scripts* and *broadcast event-driven parallel interpretors*, that from now we prefer to call shortly *reactive scripts* (RS) and *reactive scripts interpretors* (RSI).

In the context of reactive scripts, emphasis is put more on behavioral aspects than on data aspects. Actually, the definition of reactive scripts is "parametrized" by an underlying language of external statements and expressions whose task is to manage variables and data. We shall describe data and variables aspects when considering the particular RS interpretor based on the underlying language Tcl/Tk.

The structure of the paper is as follows: first, the notion of a reactive script is introduced. Then, interpretors of reactive scripts based on Tcl/Tk are described and some examples given. Finally, related works (Java and Agent-Tcl) are considered before the conclusion.

1 Basic commands

Waiting for an event `E` to occur is simply written `await E`, and to generate event `E` is written `generate E`. Generation of an event concerns only the current reaction, and is lost for the future (events are not persistent). Accordingly to the interpretor approach, events need not to be declared before being used.

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

To trigger execution of a command by the occurrence of an event `E`, one puts the command in sequence with `await E`. For example, the command

```
await E;{puts Received!}
```

prints "`Received!`" when `E` occurs, that is when

```
generate E
```

is executed. Note the semi-colon symbol used to put commands in sequence, and also the printing statement `puts` enclosed by braces. More generally, exact syntax of external statements like printing statements, assignments or procedure calls is of no interest for the moment. These external statements are always put between "`{`" and "`}`".

We are now considering several properties of basic commands.

Event Broadcast

Events are broadcast that is, execution of `generate E` fires all the `await E` commands that are stored in the interpretor. For example, suppose one enters into the interpretor the command

```
await E;{puts Received1!}
```

then, the command

```
await E;{puts Received2!}
```

Now, when `generate E` is run by the interpretor, the two `await E` commands are immediately fired and thus both messages `Receive1!` and `Receive2!` are immediately printed. The sequence operator does not introduce any delay: the two messages are printed in the same reaction `E` occurs (we shall say that *sequencing is immediate*). This implies for example, that message `E!` is immediately printed by

```
generate E;await E;{puts E!}
```

Multiple Generation

Multiple generation of the same event during a single reaction is equivalent to generate the event once. For example, suppose that the two following commands are stored in the interpretor:

```
await E;generate F;{puts E!}
```

and:

```
await F;generate E;{puts F!}
```

Suppose that `E` is generated; then `await E` is fired, `generate F` is executed and message `E!` is printed. In the same reaction, as `F occurs`, `await F` is fired, event `E` is generated for the second times which has no effect, and message `F!` is also printed. Actually, the two messages `E!` and `F!` are both simultaneously printed as soon as `generate E` or `generate F` is run by the interpretor. Note that the result is the same if `E` and `F` both occur in the same reaction, for example as a consequence of:

```
generate E;generate F
```

Non-determinism

The order in which parallel actions are executed is not fixed (in other words, *non-deterministic* behaviors may appear). This is the case for the following two commands:

```
await E;{puts Received1!}
await E;{puts Received2!}
```

The order in which `Received1!` and `Received2!` are printed depends on the interpretor implementation. The only thing sure is that they will be printed together in the same interpretor reaction, as soon as a `generate E` command is executed.

2 Parallelism and the Stop Command

In this section, one introduces two operators to mimic interpretor reactions and parallelism. Actually, these two operators give the way to *program* reactive scripts having in mind the way the interpretor behaves.

Stop Command

The `stop` command mimics interpretor reactions: a `stop` command stops execution for the current interpretor reaction, and is the new starting point for the next reaction. For example, to print message `First!` during an interpretor reaction, then `Second!` during the next, one can simply enter:

```
{puts First!}
```

then let the interpretor reacts, and then enter the new command:

```
{puts Second!}
```

Another solution is to use a `stop` statement and write directly:

```
{puts First!};stop;{puts Second!}
```

Note that in this last solution, the two printing commands are introduced together, and not one at a time as in the first solution.

The `stop` command is needed to trigger execution of a command by several occurrences of the same event. For example, the following command prints message `Two!` after two occurrences of `E`:

```
await E;stop;await E;{puts Two!}
```

The first `await E` command terminates when `E` occurs for the first time; then execution of the sequence stops as a `stop` is reached. Execution will restart from the `stop` at the next interpretor reaction, and the second `await E` becomes active at that moment. Thus `Two!` is printed after `E` occurs twice, which is the searched behavior.

Sequencing is immediate, the sequence of two `await E` commands without any `stop` between them, is **not** a solution:

```
await E;await E;{puts Two!}
```

Actually, the first `await` command terminates as soon as `E` occurs, and the second `await` command is immediately started. But as sequencing is immediate, the second `await` command also terminates during the same reaction, and thus, `Two!` is printed as soon as the first `E` occurs, without waiting for a second one.

Parallelism

In the same spirit `stop` commands mimic interpretor reactions, one introduces in the syntax a *parallel operator* to mimic interpretor parallelism.

Suppose one enters the two following commands into the interpretor:

```
await E;{puts E!}
```

and:

```
await F;{puts F!}
```

Then, both commands are run in parallel by the interpretor and `E!` and `F!` are printed accordingly to the presences of `E` and `F`. The following command which uses the parallel operator written "`||`" behaves exactly in the same way:

```
   await E;{puts E!}
||
   await F;{puts F!}
```

A parallel command can have more than two branches, is commutative and associative, and terminates when all its branches do. Note that "`||`" has a lower precedence than "`;`" and that parallel commands can be put in parenthesis for precedence purposes. For example, to print `Terminated!` as soon as `E` and `F` have both occurred, one can write:

```
(await E || await F);
{puts Terminated!}
```

Remark that without parenthesis, `Terminated!` would be printed as soon as `F` occurred, independently of `E`.

3 Configurations of Events

One has seen how to await the occurrence of an event: `await E` releases the control as soon as `E` occurs. Event *configurations* extend this to more general situations where one waits for the occurrence of several events, or for the absence of an event.

"And" of several events

The simultaneous occurrences of several events is expressed with the `and` construct. For example, message `OK!` is written as soon as `E` and `F` both occur in the same reaction, by:

```
await E and F;{puts OK!}
```

Note the difference with

```
(await E || await F);
{puts OK!}
```

which prints `OK!` when `E` and `F` have both occurred, but not necessarily in the same reaction.

"Or" of several events

The occurrence of one amongst several events is expressed with the `or` construct. For example, message `OK!` is written as soon as `E` or `F` occur, by:

```
await E or F;{puts OK!}
```

"Not" of an event

The configuration where an event `E` does not occur is written "`not E`". For example, message `OK!` is written in absence of `E`, by:

```
await not E;{puts OK!}
```

An important point however, is that `OK!` is not printed during the reaction where `E` is absent, but *during the next one*. Indeed, when does one know that an event does not occur during a reaction? The answer is: not before the end of the reaction, as before this, the event could be generated later. The end of the reaction is the precise moment one is sure the event is definitely absent. Note that not to delay reactions to event absences would cause trouble (often called "*causality problems*"), as in:

```
await not E;generate E
```

where `E` would be generated during the same reaction it is absent. This would violate the basic broadcast hypothesis of reactive scripts, and this is why only delayed reactions to event absences are allowed.

This is the moment to state the basic principle of reactive scripts, called "*absence decision principle*":

Absence of an event
cannot be decided before
the end of the current reaction

Combinations of `and`, `not`, and `or` are possible. For example, in:

```
await E or F and G;{puts OK!}
```

message `OK!` is printed as soon as `E` and `G` or `F` and `G` occur simultaneously. Note that `and` and `or` are left associative. Parenthesis can be freely used to group sub-expressions.

As another example, consider:

```
await not E or F;{puts OK!}
```

Message `OK!` is immediately printed whenever `F` occurs, and it is printed at the next reaction when neither `F` nor `E` occur. Thus termination of the `await` command is sometime immediate and sometime delayed, depending on `not` operators, but always obeying the absence decision principle.

4 Loops

Cyclic behaviors are defined using the `loop` operator. The loop body is run as soon as the command is entered, and when it terminates it is automatically restarted. For example, the two messages `First!` and `Second!` are printed in turn by the command:

```
loop
   {puts First!};
   stop;
   {puts Second!};
   stop;
end
```

Instantaneous Loops

A problem would appear if a loop body would terminate in the same reaction it is started (one speaks of an "*instantaneous loop*"), as in:

```
loop {puts OK!} end
```

Execution would cycle producing infinitely many `OK!`, and would prevent the interpretor to terminates the current reaction.

Thus, instantaneous loops should be avoided. As we are in an interpretative approach, we choose to detect instantaneous loops at run time, as soon as execution reaches the end of a loop body in the same reaction it has executed its beginning. When an instantaneous loop is detected, the interpretor prints a warning message and behaves as if a `stop` was executed. Thus,

```
loop {puts OK!} end
```

becomes equivalent to

```
loop {puts OK!};stop end
```

except that a warning message is printed at each reaction.

Finite Loops

A finite loop executes its body a fixed number of times. For example, to wait for five occurrences of an event `E` to print a message, one could write:

```
loop {5} times
    await E;
    stop
end;
{puts OK!}
```

The number of times the body is executed is given by an expression which is, as for external statements, enclosed by braces. As opposite to general loops, the body of a finite loop is allowed to instantaneously terminate without generating any warning. For example:

```
loop {3} times {puts OK!} end
```

prints `OK!` three times during the same reaction.

Exiting Loops

Loops can be exited using `break` commands. A `break` acts as a `stop`, but in addition, it forces the loop to terminate. For example, the following command prints `OK!` at each reaction while event `Term` does not occur:

```
loop
   await Term;break
||
   loop {puts OK!};stop end
end
```

Several points are to be noticed.

Break and Parallel

As `stop` does, a `break` does not prevent commands that are in parallel with it to execute. For example, in the previous example, message `OK!` is printed even at the moment `Term` occurs.

Delayed or Immediate Exiting

Exiting of a loop occurs as soon as possible, while obeying the absence decision property. Exiting is immediate if execution of the loop body is not blocked waiting for an absent event (this was the case in the previous example). Thus:

```
loop break end;{puts Exit!}
```

is completely equivalent to:

```
{puts Exit!}
```

On the other hand, exiting is delayed when execution of the loop body lasts during all the current reaction. For example, consider:

```
loop
   await E || break
end;
{puts Exit!}
```

Now, `Exit!` is immediately printed if `E` is generated during the current reaction, but otherwise, the printing action is delayed to the next reaction.

5 Test Commands

In this section one considers two test commands: `if` which tests for boolean conditions, and `when` which tests for events.

Testing Conditions

The need of an `if` command to test a condition does not require deep justifications. As previously, we do not give exact syntax of conditions, which are put between braces. For example:

```
if {$cond} then
    {puts True!}
else
    {puts False!}
end
```

prints `True!` or `False!` accordingly to the value of `cond`. Note that in an `if` statement, the condition is evaluated once, to choose the branch to be executed, and that this branch remains the same afterwards. For example, in:

```
if {$cond} then
    loop {puts True!};stop end
else
    loop {puts False!};stop end
end
```

whichever loop is executed is determined by `cond` value at the first reaction, and later, the printed message remains always the same, independently of the actual value of `cond`.

Testing Events

The `when` command allows to test an event configuration in the current reaction. Of course `when` commands obey in all cases the absence decision principle. For example, consider:

```
when E then generate F else generate G end
```

Event `F` is generated if `E` occurs in the current reaction; otherwise, `G` is generated at the next reaction, as a consequence of the absence decision principle.

Passing control to branches is governed by the absence decision principle. In case of a single event test, this means that `else` branch execution is always delayed to the next reaction. Note however that `then` branches can also be delayed because of `not` in the tested configuration, as in:

```
when not E then
   {puts Absent!}
else
   {puts Present!}
end
```

Message `Present!` is immediately printed as soon as `E` occurs, as in this case evaluation of `not E` returns false before the end of the current reaction. On the contrary, if `E` is absent, message `Absent!` is only printed during the next reaction, as `E` absence cannot be decided before the end of the current reaction.

6 Preemption and Control Commands

The need to force termination of a command (to *preempt* it) when an event occurs, or to control its execution by an event, appears rather naturally in many contexts.

Until Command

We already saw how `break` commands force loop bodies to terminate. In fact, the following is a general shape to preempt a given command `P` by the occurrence of an event `E`:

```
loop
   await E;break
||
   P;break
end
```

However this shape is unsatisfactory for two reasons: first, it uses a loop only to be able to force exiting of its body; second, if `E` is absent, accordingly to the absence decision property, execution cannot be instantly continued when `P` terminates as `E` is yet awaited. This is why the `until` command is introduced: a `until` statement executes its body and it can terminate for two reasons: either because the body terminates, and in this case termination of the `until` is immediate; or because the event occurs, and in this case termination of the `until` depends on the body, accordingly to the delayed absence principle.

For example consider the command:

```
do
   await E;{puts E!}
||
   await F;{puts F!}
until G;
{puts Terminated!}
```

If `G` does not occurs before both `E` and `F` does, then all works as if the `until` command was not there: `E!` is printed as soon as `E` occurs, `F!` is printed as soon as `F` occurs, and `Terminated!` is printed simultaneously with the last event, as then the body of the `until` terminates.

On the contrary, if `G` occurs while `E` or `F` have not yet occurred, then the `until` command is exited at the next reaction and `Terminated!` is printed at that time.

Use of Configurations

As for `when`, arbitrary configurations can be used as preemptive conditions in `until` commands. Of course, behaviors always obey the absence decision principle, which means that termination of a `until` may be delayed if the body execution or the configuration evaluation lasts all the current reaction.

Actual Parts

An "`actual`" part can be added to a `until` command to be executed only in case of actual preemption. For example, in the following command, `Preemption!` is printed only if `E` occurs before `F`:

```
do
   await F
until E
actual
   {puts Preemption!}
end
```

Note that, according to the absence decision principle, `Preemption!` printing is always delayed to the next reaction (as actual preemption implies that the `until` body is still awaiting `F`). Note also that the `actual` part is not executed when `E` and `F` are simultaneous, and that in this case the `until` command terminates immediately.

Control Command

Execution of a command can be controlled by the occurrence of an event, using the `control` operator. Actually, the body of a control command is executed only during reactions where the controlling event occurs. For example, the following command prints a message only when `E` occurs:

```
control
   loop {puts OK!};stop end
by E
```

Note that complex event configurations are not allowed: only single events can be used in `control` statements.

7 Local Events

Local events give a way to restrict event broadcast, as visibility of a local event is restricted to the statement defining it. For example, consider:

```
event E in
   do
      comm1
   ||
      comm2
   until E
end
```

where `comm1` and `comm2` can both force termination of the other by generating event `E`. The point is that termination of the two commands cannot be forced from outside as the local event statement masks all generations of `E` from there.

Several local events can be defined once, with the shape:

```
event E1,...,En in ... end
```

8 Behaviors

A *behavior* is a *declaration* which associates a name to a command. For example the following behavior associates the name `B` to the command that waits for `E` to print `OK!`:

```
behavior B
   await E;{puts OK!}
end
```

We will see in section 9 how behaviors can be parametrized. To run the command associated with a behavior, one uses the `run` command as:

```
run B
```

Actually, each `run` command runs a *a new fresh copy* of the associated behavior, and thus several runs of the same behavior can coexist without interference. For example, consider the following behavior:

```
behavior X
   await E;{puts E!};
   await F;{puts F!};
end
```

Suppose that `E` is generated and that `X` is run in the same reaction by:

```
generate E;run X
```

In response, `E!` is immediately printed. Now suppose one runs an other time `X` by:

```
run X
```

Then the first run is waiting for `F` while the second is waiting for `E`. Thus, if for example both events `E` and `F` are simultaneously generated by:

```
generate E;generate F
```

then, only one message `E!` is printed by the second run while two messages `F!` are printed, one by each run.

Local Variables

The way to define local variables in behaviors depends on the underlying interpretor of external commands and expressions. The case of the RS interpretor based on Tcl/Tk is considered in section 14.

Dynamic Binding

The binding between a `run b` command and the behavior `b` is established dynamically, when the command starts to be executed. Command `run b` has no effect if behavior `b` does not exists when it starts execution.

To re-declare a behavior does not erase existing old bindings. All runs that were using the old behavior, continue their execution without change. For example, suppose the interpretor knows the two following behaviors:

```
behavior B await I;{puts I!} end
behavior C run B end
```

and suppose that `C` is run:

```
run C
```

Of course, `I!` is printed whenever `I` is generated. Now suppose, before that, `B` is changed by:

```
behavior B await J;{puts J!} end
```

Then, message `I!` continue to be printed whenever `I` is generated.

Multiple Declarations

Multiple declarations of the same behavior during a single reaction are rejected as they could generate non-deterministic situations. To detect and reject multiple declarations, one chooses to delay the effect of declarations to the next reaction. At the end of a reaction, one can decide that a behavior has been multiply defined and thus reject the declarations (they have no effect).

Here is the second basic principle of reactive scripts, called "*declaration delay principle*":

**The effect of a declaration
only takes place
at the next reaction**

For example, consider:

```
behavior b comm0 end;
(
   await E;behavior b comm1 end
||
   await E;run b
)
```

In response to `E`, `run b` runs `comm0` as the effect of the second declaration does not take place during the current reaction. To run the new definition `comm1`, one has to delay the execution to the next reaction:

```
behavior b comm0 end;
(
   await E;behavior b comm1 end
||
   await E;stop;run b
)
```

In response to `E`, two declarations of `b` take place in a single reaction in:

```
behavior b comm0 end;
(
   await E;behavior b comm1 end
```

```
    ||
       await E;behavior b comm2 end
    ||
       await E;stop;run b
    )
```

Thus, as these two re-declarations are rejected, `run b` runs the old definition `comm0`.

9 Behavior Parameters

Behaviors can have parameters which are events or values.

Event Parameters

Event parameters are of three kinds:

- Generations of `in` parameters by the environment are known by the behavior, but the converse is false: generations by the behavior are not transmitted to the environment.
- Generations of `out` parameters by the environment are masked to the behavior, but generations by the behavior are transmitted to the environment.
- `inout` parameters are seen as identical by both the behavior and the environment.

For example, in the following behavior, event `action` is generated each time `pressed` occurs:

```
 behavior Button
    in pressed;
    out action;

    loop
      await pressed;
      generate action;
      stop;
    end
end
```

Runs are created from a parametrized behavior by associating a list of arguments to the parameters. For example:

```
run Button(Push,Move)
```

Note the positional association of arguments to parameters.

Value Parameters

Value parameters are introduced by the `val` keyword. For example, the following behavior prints its parameter value at each reaction (for the moment, do not consider in detail what is put between braces, and just consider this is a way to use variables and values):

```
behavior Z
val param;
   loop {puts $param};stop end
end
```

A run of this behavior is:

```
run Z({$arg})
```

Then, at each reaction, value of `arg` is printed. Note that parameters are evaluated at each reaction, not just at the first one.

10 Objects and Methods

Traditionally, object encapsulate data which are processed by their methods. In reactive scripts, the task of defining and using variables and data is transferred to an underlying interpretor of external statements and expressions (how to define object data parts is described for the RSI-TK interpretor considered in section 14).

Objects

In the context of reactive scripts, an object simply gives a name to several commands and methods which are said to be *attached* to the object. Commands attached to an object are executed at each reaction. On the opposite, methods attached to an object must be explicitly called to be executed. Actually, a method is a behavior run, whose execution is controlled by the object to which it is attached: the only way to execute it is to send the behavior name to the object. Moreover, an object can be removed from the interpretor or frozen, which means that its state is saved before removal, allowing the object to be used in other contexts.

Suppose behaviors `m1` and `m2` defined by:

```
behavior m1
   loop {puts m1!};stop end
end

behavior m2
   loop {puts m2!};stop end
end
```

and an object `x` defined by:

```
object x
   loop {puts x!};stop end
methods
   m1 m2
end
```

This command attaches

```
loop {puts x};stop end
```

and the methods `m1` and `m2` to object `x`. At each reaction, the command attached to `x` is executed and thus message `x!` is printed. On the opposite, `m1` is run, and thus `m1!` is printed, only when the command

```
send m1 to x
```

is executed. The situation is similar for the other method `m2`.

Multiple Attachments

Previously attached commands and methods are not removed by an object statement; instead, the new command and methods introduced by the statement are added to the set of components attached to the object. For example, consider:

```
object x loop {puts 1!};stop end end
```

Message `1!` is printed at each reaction. Now, suppose that the statement

```
object x loop {puts 2!};stop end end
```

is also executed. Then, both messages `1!` and `2!` become printed at each reaction.

Methods

Here are the basic characteristics of `send` commands and of methods.

Asynchrony of Methods

The `send` command terminates immediately, and does not wait for the called method to start to execute. The semantics is a "send and forget" order in which the caller can immediately continue to execute (thus, it is an *asynchronous* call). Note that non-determinism can occur, as to send two orders in sequence does not prevent the second one to be processed before the first.

Execution of Methods

Method are "one shot" and are immediately executed: once a method called during a reaction, it is executed during this reaction. Moreover, only the first call to a method is effective, the others having no effect. Thus, during a given interpretor reaction, each method is either executed once, if called during the reaction, or otherwise not executed at all. This "one shot" property is important to prevent objects to enter into interblocking situations where for example, two objects call each other for ever. As instance, consider two graphical objects shown on figure 1, which must move together, when receiving a `move` order. A symmetric solution consists in transmitting each received `move` order to the other object. Note that it would of course generate an interblocking situation if method `move` were not "one shot" (for example, on reception of a `move` order, `O1` would transmit it to `O2`, which in turn, would send the `move` order to `O1`, and so on forever).

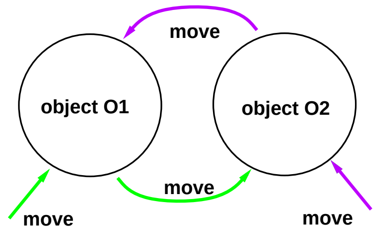

Figure 1: Two graphical objects with a `move` method

Semantics of Objects

Semantics of objects can be given using events in a rather direct way. In fact, the semantics of:

```
object O
    comm
method
    m1 ... mn
end
```

is:

```
do
    comm
  || control run B1 by O-m1
  || ...
  || control run Bn by O-mn
until O-destroy
```

Event `O-destroy` is used to destroy the object (see next section). The semantics of `send m to O` is simply `generate O-m`.

The present discussion shows that objects and methods enter in a rather natural way into the broadcast event driven approach, although different in spirit from it, as method calls are not broadcast but sent to precise targets. Reactive scripts give both way of programming in an unified framework.

11 Removing Objects

Objects are removed from the interpretor either by destroying or by freezing them. Use of freezing for migration is shown in section 15. In both cases, the removal of an object does not prevent it to execute for the current reaction: the removal becomes effective only at the next reaction.

Destroying Objects

An object can be destroyed using the `destroy` command. Note that to define an object with no method at all can be useful just to be able to destroy commands attached to it. For example, after entering the command:

```
loop {puts OK!};stop end
```

there is no way to prevent the printing of `OK!` at each reaction. But after creation of the object:

```
object x
   loop {puts OK!};stop end
end
```

it becomes possible to stop the printing actions by executing:

```
destroy x
```

which destroys object `x` (by generating event `x-destroy`).

Freezing Objects

To freeze an object means to remove it from the interpretor after having saved its state, to be able to recover the object later. The effect of a freezing command of the form:

```
freeze x
```

is to assign to a variable (whose name is implementation dependent) the script of "what remains to be done" by x. For example, consider:

```
object x
    await E;{puts E!}
  ||
    loop {4} times
       await F;{puts F!};stop
    end
end
```

Messages `E!` and `F!` are printed ins response to:

```
generate E;generate F
```

Now, freezing `x` removes the object from the interpretor and produces the script:

```
object x
   await F;{puts F!};stop;
   loop {2} times
      await F;{puts F!};stop
   end
end
```

Note that waiting of event E has vanished and that three waitings of F still remain (the loop counter is 2 because of the loop expansion).

12 The Next Command

The `next` command forces the interpretor to go on automatically with the next reaction as soon as the current one is over. The `next` command gives a way to define an *active* mode for the interpretor, in which a new reaction takes place as soon as the previous one terminates, without any delay between them. This active mode can be useful for example to count the time before an event occurs, as in:

```
do
  loop
     next;{incr TIME};stop;
  end
until b1
```

The `incr TIME` external command adds one to variable `TIME`. As a result of the next command, the interpretor runs without any interruption, while event `b1` does not occur. This code fragment comes from the reflex-game example described in section 13.

An other important use of the `next` command is related to the basic absence decision principle of reactive scripts. Indeed, message `Absent!` is printed in absence of event `E` by:

```
     await not E;{puts Absent!}
```

However as a result of the absence decision principle, the printing action takes place during the reaction that follows the absence of `E`. Actually, the printing action is delayed as long as the interpretor is run for a second time. Using the `next` command one can force this second reaction to take place immediately after the first one, and thus perform the printing action without any delay. The code is:

```
loop
   next;
   when not E then
     {puts Absent!}; break
   end;
   stop
end
```

Interpretors of Reactive Scripts

Reactive scripts are "parametrized" by a language of external commands and expression. We are going to describe an implementation of reactive script with Tcl/Tk[10] as underlying interpretor. Actually, in all the examples previously given, expressions and external commands were in Tcl/Tk syntax.

13 RSI on Top of Tcl/Tk

The reactive script interpretor on top of Tcl/Tk is named RSI-TK. Here is a session using it:

```
rsi-tk (version 1)
1-: await I;{puts OK!}.
2-: .
3-: generate I.
OK!
```

The "." symbol is used to enter a command into the interpretor. Note that commands can be entered on several lines without any trouble. The interpretor reacts when a command is entered, or when a single "." is typed. The "prompt" shows the current reaction number (starting to one); it is printed after each reaction, when the interpretor is waiting for a new command.

Implementation

The RSI-TK interpretor is implemented as a list of parallel components, all executed at each reaction. A parallel component get stuck on an event while it is not generated. Execution of a stuck component continues as soon as the awaited event becomes generated by an other component. Thus, interpretor reactions progress like a wave which fires stuck components as and when events become generated; the firing of a component generates new event, which in turn release new components, and so on. The current reaction is over when all components either have finished to react (they terminate or reach `stop` statements), or are stuck on non-generated events; then, all these events can safely be considered as absent. The next reaction is immediately started if a `next` statement has been executed; otherwise, the interpretor waits for a new command to be entered, add it to the list of parallel components, and then starts the new reaction.

Interface with Tk

To put a reactive script interpretor on top of Tk allows one to use Tk graphical primitives as external commands. Moreover, it also gives a way to drive the RS interpretor with commands output by the Tk graphical objects. Here is an example of behavior using Tk:

```
behavior tkbutton
val n;

{button .$n -text $n -relief flat};
{.$n configure -command rsi "generate $n"};
{.$n configure -bd 10};
{pack .$n -expand 1 -fill x};
{update};

loop
```

```
   await {$n};
   {puts "$n pressed!"};
   stop
end

end
```

The first external command creates a button whose text is the value of the variable `n`. The others external commands, except the second one, are used to configure the button graphical aspect. The second command deserves special attention as it gives the way to drive the RSI-TK interpretor by clicking in the button. Actually, the predefined `rsi` Tcl command takes a string as parameter and gives it as input to the RSI-TK interpretor. Thus, the command generates the event whose name is the value of `n`, each time the button is clicked in. Finally, the loop defines a behavior in which a message is printed each time the event `n` is generated; note that this is indeed the case when the button is clicked in.

The RSI-TK structure is described on figure 2.

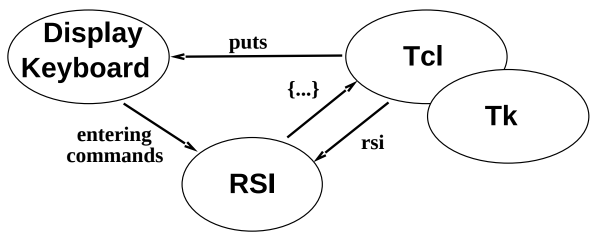

Figure 2: The reactive script interpretor on top of Tcl-Tk

The effect of the Tcl command "`{rsi comm}`" is to force the next interpretor reaction and to give the script `comm` as input for it.

The `tkbutton` example shows a use of the `rsi` command, associated to a Tk object by `-command`, to drive the interpretor from the graphical level. Note that clicks into graphical objects are bufferized by Tk and are not instantaneously delivered as input to the interpretor: there is an "asynchronous" link beetwen RSI-TK and Tk.

To summarize, there are three ways to force the next reaction to take place immediately after the current one:

- to internally execute the RSI `next` command, defined in section 12;
- to receive an external `rsi` command comming from Tk;
- to internally execute a Tcl statement of the form "`{rsi ...}`".

We now code the example of a small reflex-game program.

The Reflex Game

The reflex game we consider is made of two buttons: first, the player must click in the button whose text is `READY`; then after a while, text will be changed by `GO!` to indicate that the reflex measure starts. In the same reaction the text of the other button is changed in `STOP`, and the user must click in it as soon as possible. After `GO!` appears the interpretor enters a cyclic loop, and the number of reactions until `STOP` is clicked in is finally printed. There is one additional feature to prevent the user from cheating: the game is over if `STOP` is clicked in before `GO!` appears. Here is the reactive script which is rather self-explanatory[2] (file `tk-button.rs` contains the behavior `tkbutton` defined previously):

```
load {tk-button.rs}.
run tkbutton({b1}).
run tkbutton({b2}).

{.b1 configure -text "  "};
loop
   {set TIME 0;};
   {.b2 configure -text READY};
   await b2;
   do
     loop {YFRNG} times next;stop end
   until b1
   actual
      {.b2 configure -text CHEAT!};
      break
   end;
   {.b2 configure -text GO!};
   {.b1 configure -text STOP};
   do
     loop
        next;{incr TIME};stop;
     end
   until b1;
   {.b1 configure -text $TIME}
end.
```

14 Data and Variables

In this section we describe how local variables in behaviors and data in objects are implemented in the RSI-TK interpretor.

Local Variables

Variables used in external Tcl statements or expressions are defined at the upper level and are thus global variables. For example, in:

```
  behavior b
```

[2] YFRNG: Your Favorite Random Number Generator.

```
    {set v 0};
    loop
       {puts $v}; stop; {incr v}
    end
  end
```

all runs of `b` share the same global variable `v`.

The RSI-TK implementation gives a way to define variables which are local to behaviors. Actually, only one such variable is sufficient when defined as a Tcl *array*. This variable is called `local` and the implementation assigns it a new name for each run. Thus, in:

```
    behavior b
       {set ${local}(v) 0};
       loop
          {puts [set ${local}(v)]};
          stop;
          {incr ${local}(v)}
       end
    end
```

all runs of `b` own a distinct array whose name is the value of `local`, and local variables can simply be defined as components of this array.

Object Data

Encapsulated data in objects cannot be implemented as global Tcl variables because they would not be protected and would be accessible from everywhere.

The RSI-TK implementation gives a way to define data for an object, which are only accessible by commands and methods attached to the object. Actually, as for `local`, only one variable is sufficient. This variable is called `self`. In the following object, the `state` data is automatically set to 0 when the object is created, and it is shared by the two methods attached to the object:

```
behavior on_off
  loop
    {set ${self}(state) 1};
    stop;
    {set ${self}(state) 0};
    stop;
  end
end.

behavior state
  loop
    if {[expr [set ${self}(state)] == 1]}
      then {puts "On"}
```

```
      else {puts "Off"}
    end;
    stop;
  end
end.

object toaster
  {set ${self}(state) 0};
methods
  on_off state
end
```

15 Remote RSI

Remote reactive scripts interpretors can receive scripts through the network. A remote version of RSI-TK, called REM-RSI-TK, has been implemented on Unix platforms. REM-RSI-TK is implemented as a Unix process and identified by a name and the machine on which it runs. The following Unix command:

```
rem-rsi-tk R
```

runs a remote RSI called `R`.

The `send-rsi` Unix command transmits a string to an interpretor designed by a name and the machine on which it runs. For example:

```
send-rsi cma R "generate I"
```

sends the script "`generate I`" to the RS interpretor named `R` running on machine `cma` (in absence of machine name, the local host name is taken as default). Note that the link beetwen the sender and the target interpretor is asynchronous (the script is transmitted through the network).

To internally send a script to a remote interpretor, one uses a Tcl command of the form:

```
{exec send-rsi M R "..."}
```

The REM-RSI-TK structure is described on figure 3.

Note the three input sources for REM-RSI-TK: the keyboard, the network with the `send-rsi` command, and the Tk graphical level with the `rsi` command.

Object Migration

Object migration is basically achieved by the two commands `freeze` and `send-rsi`. The following behavior implements migration:

```
behavior migrate
```

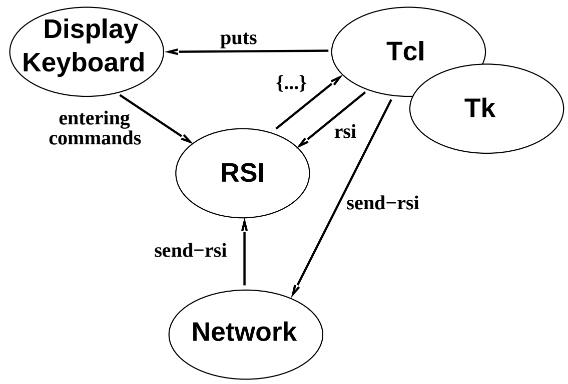

Figure 3: The remote reactive script interpretor on top of Tcl/Tk

```
val obj, machine, interp;
freeze {$obj};
next;stop;
next;stop;
{exec send-rsi\
     $machine $interp \
           [set rest\_$obj]}
end
```

First the object is frozen. Then one waits for the next reaction as freezing actions are not immediate. Assignment of the remainder of the object to the Tcl variable `rest_obj` is performed during this reaction. Finally, the remainder of the object is sent to the remote interpretor using `send-rsi`.

For example, the following command makes object `x` migrate to the interpretor named `RSI` running on machine `duick` by:

```
run migrate({x},{duick},{RSI})
```

Transportable Objects

A transportable object has the facility to migrate when it decides to do so. The natural way to implement transportable objects is to attach a `transport` method to the object. This method cannot directly be the `migrate` behavior because the first action performed by `migrate` is a freezing action. Thus, `migrate` would freeze the object without letting it the possibility to perform the transfer on the remote interpretor. One defines the `transfer` method by:

```
behavior transfer
val machine, interp;
{\
   rsi "run migrate \
       ({[set ${self}(name)]},\
```

```
            {$machine},{$interp})"\
};
next
end
```

Several points are to be noticed:

- The `rsi` command inputs the `run migrate` command into the interpretor, and thus it becomes executed *in parallel* with the other commands.
- Component `name` of `self` contains the object name.
- `next` insures that the `rsi` command is performed.

Now, an object with the `transfer` method can decide to migrate at will. For example, the following object reacts 5 times and then migrates spontaneously:

```
object x
   loop {5} times
        next;{puts x!};stop
   end;
   send transfer to x;
   stop;
   loop {puts x!};stop end
methods
   transfer({duick},{RSI})
end
```

Related Works and Conclusion

16 Basis of Reactive Scripts

The basis of reactive scripts are on one side, the SL synchronous language, and on the other side, the Reactive Object Model (ROM). Reactive script interpretors are (as SL and ROM) implemented in Reactive-C. We are now going to briefly describes these three formalisms (one can find more informations on Internet at URL `http://cma.cma.fr/RC/rc-project.html`).

The SL Language

The SL[6] language is one of the family of *synchronous formalisms* which contains several programming languages such as Esterel, Lustre and Signal, and also graphical specification formalisms such as Statecharts (see [1] for a general overview of these formalisms). The basic paradigm shown on figure 4 considers *reactive systems* which when activated by an input event, react instantaneously by producing an output event. In synchronous formalisms activations are called *instants* and are global to all parallel components.

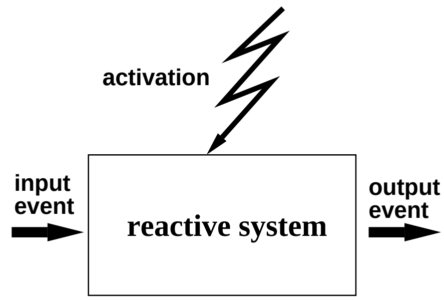

Figure 4: The Reactive System Paradigm

In SL, which is strongly linked to the ESTEREL synchronous language[3], communication is based on broadcast signals. At each instant, a signal is present if it is a member of the input event or it it is emitted by some component during this instant; otherwise it is absent. Moreover, all components get the same information about the signal presence or absence (this is the broadcast characteristic). In SL as in other synchronous languages, there is no dynamic creation of any kind, and programs are guaranteed to be deterministic.

The relation of SL with reactive scripts is as follows:

- SL instants correspond naturally to interpretor reactions (actually, reactive script interpretors are examples of reactive systems), and SL signals correspond to events.
- SL syntax is very close to reactive scripts. Some operators are identical: sequence, parallel, `stop`, `loop`, `run`, and `when`. In SL, external commands must always have the C syntax. Translation of others SL operators is as follows:

SL Language	Reactive Scripts
`wait`	`await`
`emit`	`generate`
`kill`	`until`
`signal`	`event`
`module`	`behavior`

- Semantics are the same with only one exception: in SL, when preemption is actual then termination of the `kill` statement is in all cases delayed to the next instant.

In this respect, the two main differences between reactive scripts and SL are:

- Event configurations have no counterpart in SL. Moreover, termination of the `until` operator is less constraining than in SL (note that this does not introduce causality problem).
- Dynamic creation is the rule in reactive scripts, where one can dynamically add new commands at any time, although this is forbidden in SL.

Reactive Objects

The "*Reactive Object Model*" (ROM)[5], is an attempt to mix an object based approach with the reactive approach in which there are instants global to all parallel components. Objects encapsulate data shared by their methods which, as in the ACTOR model[2], are called by sending asynchronous non-blocking orders. In the ROM model, transmission of an order is required to be processed *during the same instant* the order is sent (orders are said to be *instantaneous*). Moreover, at each instant, only one order can be processed, the other ones being simply rejected. This property is called, "*one call per instant*" property. Actually, the fact that order are instantaneous and the "one call per instant" property are exactly the same for reactive scripts, where methods are "one shot" and executed in the same reaction they are called.

The ROM model is currently studied and developed under contract with the FRANCE TÉLÉCOM company[3]. Its formal semantics is described in [8].

There are two main differences between reactive scripts and the ROM model. First, objects and methods cannot be sent as method parameters in reactive scripts, as it is the case in ROM. Second, there is presently no analog in RS of the "cloning" mechanism of ROM .

Reactive-C

REACTIVE-C (RC)[4] was designed to allow a reactive programming style in C. The basic notion is that of code execution *up to* `stop` statements which end the current reaction. At the next activation, execution will resume from beyond these `stop` statements. Reactive statements are used to code reactions to activations, and `stop` is of course the simplest reactive statement. The `merge` reactive statement can be considered as a primitive parallel operator, which at each instant makes its two branches both react in a fixed order (the first branch, then the second).

RC gives provision to break one instant into several micro-instants: `suspend` freezes the control flow, but unlike for `stop`, execution can be reactivated in the same instant. While a branch of a `merge` statement is suspended the other may still react, the first one be awaken later.

Finally, RC encapsulates a "remote procedure call" mechanism which allows to define remote reactive processes reacting and communicating via a network.

RC has been used as a "reactive assembly language" to implement several reactive models, specially SL and the ROM model. It is also the implementation language of reactive script interpretors. All these implementations are similar in spirit: the basic idea is to delay decisions concerning absence (event absence for reactive scripts, signal absence for SL, and method call absence for ROM), accordingly to the absence decision principle. The structural translation goes on as follows: parallel commands (we use the reactive script terminology) are mapped into `merge` constructs. The `suspend` primitive opens the way to delay absence decisions: an execution flow which has to test an unvalued event gets suspended, while the event keeps a chance to be generated by other commands. With this strategy, parallel commands are executed as far as possible; control switch from one command to the next either when the command `stop`s, or terminates forever, or hangs suspended after an event. The current reaction is over when all parallel execution flows lay either suspended, terminated

[3] Contract France Télécom-CNET 93 1B 141, #506

or stopped, and then all pending events which were not generated are recognized as absent, so that suspended commands can be cleaned in preparation of the next reaction.

Finally, inter-process communications used for remote reactive script interpretors are implemented with RC reactive processes.

Implementations of RSI-TK and REM-RSI-TK in RC are about two thousands lines of code each.

Figure 5 summarizes the situation.

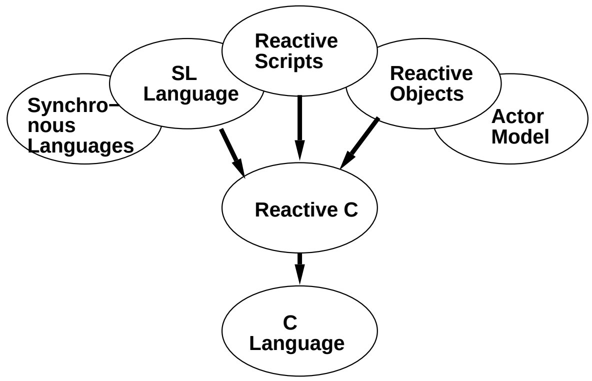

Figure 5: Implementations in Reactive-C

17 Related Works

In this section, one compares reactive scripts with two formalisms Java, and Agent-Tcl.

The Java Language

The Java language[7] allows object oriented programming of scripts suited to Internet and WWW. Java looks like C++ but is far more simple and manageable. The Java compiler generates bytecodes, a technique which is claimed to be architecture neutral and favoring portability. Java supports parallel programming based on a multithreading capability. Finally, emphasis is put in Java on robustness and security aspects.

Object Orientation

The Java language is object oriented. It uses the notion of a *class* and supports data encapsulation, polymorphism, and simple inheritance. Actually, as it is also the case for C++, emphasis is put on data aspects: object states are the values of the object data and methods are basically functions. On the opposite, reactive scripts put the emphasis on behavioral aspects: methods have states (the `stop` statements reached by execution) and object states are basically their method states. In reactive scripts, behavioral aspect are not necessarily coded by data, as it is the case for Java, and objects offer mainly a way to structure behaviors. To sum up, one could say that in Java objects are mainly data while in RS they are mainly behaviors: Java is *data oriented* although RS is *behavior oriented*.

Parallelism

Parallelism is in Java achieved at the language level by the `Thread` class and the notion of a `synchronized` method. The paradigm is the one of multiple threads which synchronize using monitors, an old construct introduced by C.A.R. Hoare in 1974. Two main criticisms can be made to this approach: first, the programmer must deal with low-level primitives such as condition variables for synchronizations and communications. Such tools are not well suited to the high-level abstractions in object design. Second, this approach is also incompatible with a modular approach. Adding a new parallel component often leads to the redefinition of the other components e.g. to avoid deadlocks.

Reactive scripts define a full parallel operator with a very powerful and high-level broadcast communication mechanism. Semantics of programs do not depend on the underlying operating system and can be made formal (at least for those which are translatable into SL). Actually, parallel branches are some kind of "logical" threads, with no need of any multithreading support from the operating system. Broadcast of events gives a very powerful mechanism for modularity:

- to add an observer in a system is totally transparent for the other components;
- all components have the same vision of event simultaneity.

Thus, in reactive scripts modularity comes from objects, as in Java, but moreover it is also fully compatible with parallelism.

Security

The Java compiler and the Java bytecode interpretors implement several levels of security. First, the language forbids error-prone notions such as pointers, or delete operators, which exist in C++. Second, interpretors perform a bytecode verification phase before running any program. This verification phase assures that execution will not produce run-time errors. Finally, various network protocols can be used to increase security.

Reactive scripts presently do not consider security questions. However, reactive scripts in themselves (that is, not considering external commands and expressions) are *robust*, meaning that reactions never diverge and that run-time errors are always trapped. More work has to be done to add security features to RS.

Agent-Tcl

Agent-Tcl[11] is an extension of Tcl in which transportable agent can be defined. A transportable agent is a Tcl script which can migrate from machine to machine and communicate with each other. A transportable agent chooses when and where to migrate, and the systems handles the transmission details. Agents can use Tk to create graphical user interfaces on their current machine. Agent-Tcl uses Internet through the TCP/IP protocol and is implemented as two components: the first one is a modified Tcl interpretor, and the second one is a server whose task is to accept incoming agents. Security in Agent-Tcl means that all agents run with the permissions of the server, and that the server refuses requests that do not come from an "approved" machine.

On the opposite to RS and Java, Agent-Tcl does not provide any object oriented features. Parallelism in Agent-Tcl is handled only at a very low level; it uses a `fork` primitive analogous to the one of Unix, to clone agents. However, not all of the state of an agent is captured by the `fork` primitive.

The strong point of Agent-Tcl is the capability of script migration over the network. As in RS, one can define in Agent-Tcl transportable scripts which migrate when they choose to do so. Agent-Tcl gives some powerful primitives to get control of migrations over the network (for example, timing and retry mechanisms are available). More work has certainly to be done to add such network control capabilities to RS.

18 Ongoing Work

The RSI-TK interpretor presented in this paper is the first step in the implementation of the RS model; ongoing work pursues now two different objectives: first, provide the programmer with a better data management and second, offer a wider range of possible interactions between the different modules in a distributed system.

Better data management

Back to the discussion on Java, we definitely think that a proper (i.e. wrt object-oriented programming) data management is required and that the RSI-TK prototype is not well adapted to various and complex data abstractions. To this respect, the means offered to the programmer by the Java language, and even more by JavaScript (which is fully interpreted, and not pre-compiled) are quite promising and we are currently investigating, in order to build a new Reactive-Java-Script interpretor, which will associate a fully object-oriented data management with the high-level and portable means for concurrent programming featured by the RS model.

Cooperation in a distributed system

In REM-RSI-TK, communications means are restricted to interpretor commands sent to interpretors. We have found it interesting to use the concept of an "Object Request Broker" (ORB) to provide a fully generic communication mechanism between objects. A new prototype, integrating the services of a home-made (but still conform to the CORBA standard [9]) object platform, has been tested. Each interpretor now exports (to platform's trader) an interface (the "interpretor control" interface), which may then be invoked by any other object in the system, including other interpretors. This mechanism is equivalent to the `send-rsi` command presented in section 15; note that the invoking (client) object does not need to know the localization of the invoked interpretor anymore, but only the name of its interface. Furthermore, each object or behavior running in an interpretor may, at any time, export one or several interfaces. These interfaces, created and destroyed dynamically, are registered by the trader, and may then be invoked by any other modules in the system:

all system components are able to use the trader services, including objects or behaviors in any RS interpretor. Each invocation generates a corresponding event in the context of the interpretor supporting the behavior which has exported that precise interface. Only event occurrences can be generated this way; data communications have not been implemented yet. For example, this mechanism already enables dynamic binding between an external physical clock and a reactive object. As a benefit of the distributed platform, reactive server objects may migrate, transparently, still offering the same services to their clients.

19 Conclusion

We have presented the notion of a reactive script and the basic principle of it, which forbid immediate reaction to event absences. As a consequence of this principle, which is also the basis of several other reactive formalisms, causality problems do not appear anymore.

Reactive scripts are specially adapted when an event driven programming style is wanted. Broadcast of events provides a very powerful mechanism for modularity. Moreover, script approaches are extremely flexible and useful in distributed contexts, where program should be transmitted from sites to sites.

Reactive scripts interpretors have been implemented in REACTIVE-C as very small programs.

Acknowledgments

The idea of interpreting SL programs is Jean-Bernard Stefani's one. Many thanks to him and also to Amar Bouali, Guillaume Doumenc, Eric Debonne and Jean-Ferdinand Susini for their comments on a first version of this paper. Special thanks to the Tcl/Tk "guru" Xavier Fornari for his help in Tcl programming.

References

[1] Another Look at Real-Time Programming. Proceedings of the IEEE, vol. 79, 1991.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[4] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[5] F. Boussinot , G. Doumenc and J.B. Stefani. Reactive Objects. Research Report RR-2664, INRIA, October 1995.

[6] F. Boussinot and R. de Simone. The SL Synchronous Language. Technical Report RR-2510, INRIA, March 1995. Accepted for publication in IEEE Transaction on Software Engineering.

[7] James Gosling and Henry McGilton. The Java Language Environment - A White Paper. Technical report, Sun Microsystems, October 1995.

[8] C. Laneve and F. Boussinot. Two Semantics for a Language of Reactive Objects. Technical Report RR-2511, INRIA, March 1995.

[9] Object Management Group. The Common Object Request Broker: Architecture and Specification. *OMG Document 91.12.1*, December 1991.

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

[11] Robert S. Gray. Agent Tcl: Alpha release 1.1. Department of Computer Science, Dartmouth College, December 1995.

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, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
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