# Reactive shared variables bases systems

Frédéric Boussinot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Reactive shared variables based systems*

Frédéric BOUSSINOT

**N° 1849**
Février 1993

PROGRAMME 2

Calcul Symbolique,
Programmation
et Génie logiciel

# Reactive Shared Variables Based Systems

# Systèmes réactifs à base de variables partagées

FRÉDÉRIC BOUSSINOT
Ecole des Mines de Paris-CMA
B.P. 207
06904 Sophia Antipolis Cedex

email: fb@cma.cma.fr

**Abstract**

Systems made of distributed reactive processes communicating with shared variables are described. Execution of these systems is in two phases: first variables are written, then read. Couples of these two phases define system instants, and during one instant, all readers of a given shared variables read the same value. One thus get deterministic systems with a kind of shared variable coherency.

**Résumé**

On considère des systèmes de processus réactifs distribués communiquant par variables partagées. L'exécution est en deux phases : une phase d'écriture suivie d'une phase de lecture. Un couple formé de ces deux phases définit les instants du système. Pendant un instant, tous les lecteurs d'une même variable partagée lient la même valeur. On obtient ainsi des systèmes déterministes qui assurent une cohérence des variables partagées qui les composent.

# 1 Introduction

Communicating systems which behave in an activation/reaction mode have recently been considered and studied. Examples of such systems are "reactive systems" introduced by Harel and Pnueli[8] that are supposed to maintain an ongoing relationship with the environment. "Synchronous systems" introduced by Berry[2,1] are reactive systems where reactions and activations are sufficiently close to be considered as synchronous. In fact, reactive systems are called synchronous when one wants to insist on response time. Several languages have been introduced to deal with reactive[7,10] and synchronous[2,6,9] systems.

"Reactive C"[3] is a C based language which allows an activation/reaction programming style. Several classes of systems have been considered and implemented in RC[5], for example nets of reactive processes communicating through unbounded fifo files[4].

Shared variables can be used for communication between parallel components provided write and read actions are atomic. Shared variables are often considered as the lowest level communication mechanism. During execution, read and write actions may be interleaved in any order and this may cause nondeterminism (a writer may update a variable before a reader gets the value, or conversely, the reader may read first). Moreover one cannot insure that several readers of the same variable all read the same value (this can be view as a lack of data coherency).

In this paper, one considers a class of activation/reaction systems where components (named modules) use shared variables to communicate. Each reaction of a system is made of one reaction of its modules (we say that execution is "synchronous"), and so there exists a global notion of instant for the systems we consider. There exists now the possibility to structure reading and writing action interleavings during each instant. More precisely, all writers write into variables before readers read out of them. Assuming such executions, system behaviors are deterministic and we get data coherency: for each instant, all readers necessarily read the same value. This is very similar to the Esterel point of view: Esterel signals are shared variables that at each instant, cannot be emitted (that is written) after being read.

The paper's structure is the following: the model is introduced first and two simple examples are given; then we discuss the implementation in RC and finally give some sessions to show how to run systems.

# 2 The model

The model we consider is based on the following notions:

**Reactivity.** *Systems* are made of *modules* that are put in parallel and that use *shared variables* to communicate and synchronize. Modules and systems are

*reactive*, that is they react under activation. More precisely, module are *reactive programs* and systems are executed in a *synchronous way*, that is one reaction (one *instant*) of a system correspond to one reaction of each module in it.

**Communication.** Shared variables hold values that can be read and written by modules. A *coherency property* is verified: during one instant, all modules reading a variable necessarily get the same value. There can be more than one writer for a given variable during one instant and in that case, values are combined using a commutative/associative function (as in Esterel). As an example of such function, consider the function that generates an error after the first writing; another example is the max function over integers.

**Execution.** Execution is divided into *two phases*. During the first phase, modules are free to write variables and cannot read them, and conversely, during the second phase, modules are free to read variables but cannot write them. The reading phase begins when all modules have terminated their writings. During the reading phase, attempts to write into a variable are postponed to the next instant.

**System structure.** Modules can be dynamically created or removed at the beginning of each reading phase. Also, shared variables can be dynamically created at the beginning of each reading phase. More precisely, the following activations are possible:

1. Start the system (**start** command).
2. Get one system reaction (**go** command).
3. Add a module and maybe some variables (**internal** or **external** commands).
4. Remove a module out of the system (**remove** command).

Several points can be discussed:

**Atomicity.** There is an atomicity aspect: one system reaction is made of several reactions of modules in it, and a new reaction cannot start before the previous one is finished. Thus, system reactions are in a sense, *atomic*. We can speak of a global system reaction made out of module *micro-reactions*, or alternatively of global instants divided into several *micro-instants*.

**Distributed termination.** The end of each phase and thus the termination of system reactions needs all modules to synchronize. A *distributed termination* technique is thus used.

**Determinism.** System behaviours do not depend on the way modules are executed and nondeterminism only comes from modules themselves. In particular, assuming deterministic modules, one gets deterministic systems whose behaviours are *reproducible*. In this case, we thus have *deterministic parallelism*.

**Comparison with Esterel.** As noticed before, there exists links between shared variables and signals. At each instant, a signal `S` cannot be emitted after being read, thus, there exists for `S`, writing and reading phases. But as opposite to shared variables, termination of signal writing phases are not globally synchronized: one signal can be in a reading phase although another one is in writing phase. In Esterel the only global synchronization needed is at the end of instant, although with shared variables, two are needed, one at the end of each phase.

In Esterel, equivalent of writing into an already read variable is called "causality cycle" and is detected at compile time; on the contrary, writing in a shared variable during read phase is postponed to the next instant. The Esterel programming style can no longer be used: for example, Esterel instantaneous dialogs[1] are not possible with shared variables. On the other hand, system dynamic structure changes are not expressible in Esterel that deals only with static objects.

## 2.1 Examples

Two examples are considered: one is a system to compute the fibonacci sequence, and the other is an arbitration protocol.

**Fibonacci sequence.** The system considered computes the fibonacci sequence defined by: $fib(0) = fib(1) = 1$ and for all $n > 1$, $fib(n) = fib(n-1) + fib(n-2)$.

The system is made of 2 variables `A` and `B` and 4 modules. Combine function associated to shared variables does not matter, as for each variable, there will be at most one writing per instant.

Modules are the followings:

- `Const11` writes 1 in the two variables `A` and `B` and then vanishes.
- `Plus` cyclically reads `A` and `B`, and writes their sum into `A`.
- `Shift` cyclically reads `A` and writes it into `B`.
- `Print` cyclically prints `A` when it has been written.

Read/write access to `A` and `B` is pictured in figure 1

The system is deterministic and computes the fibonacci sequence `1 2 3 5 8 13 21 34 55 ...` More precisely, it prints a new element (of course, without

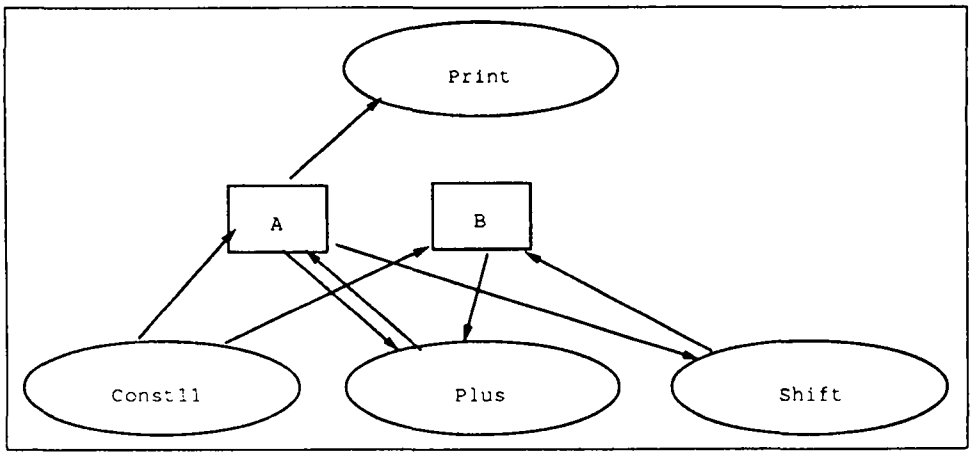


Figure 1: read/write access in the fibonacci example

recomputing the previous ones) every time it is activated by a go order. One can see that system behaviour does not depend on module execution order:

- At each instant, **Plus** writes the sum of previous instant **A** and **B** values (they are got in reading phase, so writing of their sum is postponed to the next instant). Similarly, at each instant **Shift** copies **A** previous value into **B**.
- In writing phase, **Shift** writes into **B** and **Plus** writes into **A** so the result does not depend on the order of these actions.
- In reading phase, **Plus** reads **A** and **B**, and **Shift** and **Print** reads **A**. Again, the result does not depend on the order these actions are done.

Let us note $A_n$ the value of $A$ and $B_n$ the value of $B$ at the end of instant $n$. It is clear that $A_n = A_{n-1} + B_{n-1}$ and $B_{n-1} = A_{n-2}$, and thus, $A$ holds the fibonacci sequence.

**Arbitration protocol.** Let us consider a system in which modules have their own *arbitration numbers* to determine priorities[1]. Variables are used to implement some kind of "semaphore with priority": when several modules are trying to get a semaphore, only the one with the highest arbitration number can get it and others are blocked until the winner releases the semaphore. Arbitration numbers are integers with two new values $TAKE$ and $RELEASE$ added. The combine function is the max function with for all $n$, $max(TAKE, n) = max(n, TAKE) = TAKE$ and $max(RELEASE, n) =$

[1]This example is inspired by the IEEE Futurebus proposal.

$max(n, RELEASE) = RELEASE$[2]. To get a semaphore, a module writes its arbitration number into it, and then reads the value. If it reads its own arbitration number, the module wins access and it writes $TAKE$ in the semaphore to prevent other modules to take it. Otherwise, the module waits until it reads $RELEASE$. It can then reenter the competition by writing its arbitration number again.

# 3 Implementation

Intuitively, implementation is as follows: modules are reactive procedures; they are suspended when trying to read in write phase and blocked until the next instant when trying to write in read phase. At each instant all modules present in the system are activated twice, one time during the writing phase and one time during the reading phase. The order they are activated during these two phases is not relevant as values written are combined with an associative/commutative function. Modules are put in a list that changes dynamically when new modules are added into the system, or when modules are removed from it.

We are not going to describe the RC language here (see [5] for a complete description) but only cite the primitives used for implementing systems.

- Each system module is activated at each instant using the **merge** statement. More precisely, combining **merge** and recursive reactive procedure definitions allows to define a kind of dynamically n-ary merge operator to deal with module add and remove actions.

- Suspension of modules during writing phase is obtained with the **suspend** statement. On the contrary, the **stop** statement is used to block execution up to the next instant. The **close** statement allows suspended modules to resume and to build one instant from one writing phase and one reading phase.

- Code distribution is obtained using reactive processes and reactive tasks. Reactive processes (introduced by the **rprocess** keyword) are similar to reactive procedures but are run as stand alone processes. Reactive tasks (introduced by the **rtask** keyword) are the way to use reactive processes as if it was a reactive procedure. Parameters are transfered through the network when reactive tasks call reactive processes.

Several parts have to be implemented:

- An *engine* that makes phase changes.

- A *scheduler* that makes all modules react.

[2] $max(TAKE, RELEASE)$ is undefined as well as $max(RELEASE, TAKE)$

- A *kernel* that process orders.
- Various systems made of modules and shared variables.

This architecture is described on figure 2.

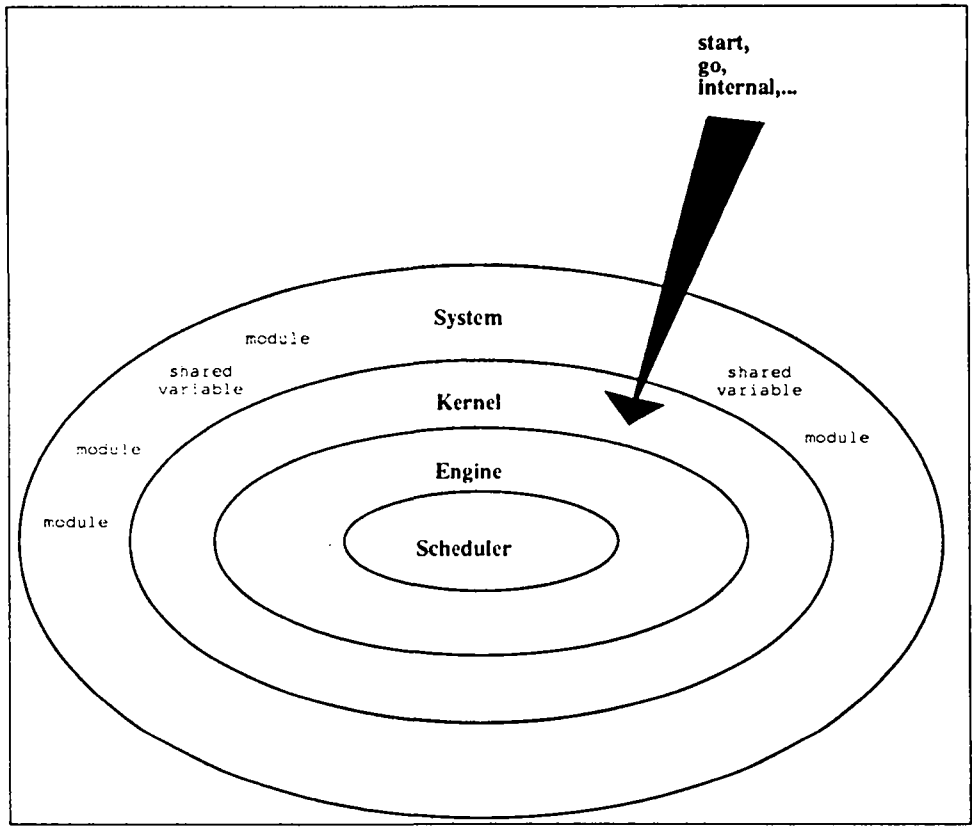


Figure 2: global architecture

In this section, we describe all components but the kernel that will be described in next section.

## 3.1 The scheduler

The scheduler executes in order elements of a list of tasks given as parameter. This list may dynamically change when a new task is added in the system, or when a task is removed from it. In these cases, the global variable **Change** is set to 1. The code is:

```
rproc SchedulerEngine(list)
TaskList list;
{
  every(Change){
    if (list){
      merge
        exec Head(list);
        exec SchedulerEngine(Tail(list));
    }
  }
}
```

The **Head** reactive procedure executes the first list element. The **Tail** C function returns all but the first list element. The **SchedulerEngine** reactive procedure is recursively defined and it is restarted every time (use of **every** statement) the list changes (**Change** variable equals 1).

## 3.2 The engine

The engine executes the scheduler and decides of phase changes. First the writing phase begins. The scheduler is executed, so each task present in the system is activated. During this phase, modules trying to read are suspended, waiting for the read phase. Then the read phase begins and modules suspended on reading operations can resume. During this phase, modules trying to write are stopped till the next instant.

Code for read/write operations is the following:

```
rproc int Read(var)
SharedVar var;
{
   if (InWritePhase) suspend;
   return Value(var);
}
```

The **Value** C function returns variable values.

```
rproc void Write(var,v)
SharedVar var;
int v;
{
   if (InReadPhase) stop;
   Combine(var,v);
}
```

The **Combine** C function combines previous and written values.

Code for the engine is:

```
rproc void Engine(){
   close
      merge
         for(;;){
            BeginReadPhase;
            stop;
         }
      merge
         exec Scheduler();
         for(;;){
            BeginWritePhase;
            stop;
         }
}
```

Notice the use of the **close** operator to make one unique instant from the two reading and writing phases.

## 3.3 Systems

System declarations consist in sets of *internal modules*. For example the fibonacci system is obtained by the declaration :

```
System((4,Print,Shift,Const11,Plus))
```

Notice that the number of internal modules appears as first parameter of the **System** macro.

Module are implemented as RC reactive procedures having shared variables as parameters. They are declared using the **module** macro (ended by the number of variable parameters).

**Fibonacci example.** Modules **Const11**, **Plus** and **Shift** are implemented as the following RC reactive procedures.

```
module2(Const11,A,B)
{
   exec Write(A,1);
   exec Write(B,1);
}
endmodule

module2(Plus,A,B)
{
   rauto int v1, v2;
   for(;;){
```

```
        into v1 exec Read(A);
        into v2 exec Read(B);
        exec Write(A,v1+v2);
   }
}
endmodule
```

Notice the use of a `rauto` variables to keep values from one instant to the next.

```
module2(Shift,A,B)
{
   rauto int v;
   for(;;){
      into v exec Read(A);
      exec Write(B,v);
   }
}
endmodule
```

**Arbitration protocol.** The two **Take** and **Release** primitives are implemented as the following reactive procedures.

```
rproc void Take(semaphore,arbitrnum)
SharedVariable semaphore;
int arbitrnum;
{
   rauto int v;
   for(;;){
      exec Write(semaphore,arbitrnum);
      into v exec Read(semaphore);
      if (v == arbitrnum){
         exec Write(semaphore,TAKE);
         break;
      }else{
         for(;;){
            into v exec Read(semaphore);
            if (v == RELEASE) break;
            stop;
         }
      }
   }
}

rproc void Release(semaphore)
SharedVariable semaphore;
```

```
{
   exec Write(semaphore,RELEASE);
}
```

# 4 Execution

We are now going to describe the last component, that is the kernel, and give some sessions using the fibonacci system.

## 4.1 The kernel

The kernel is implemented as a RC *reactive process* At each time it is activated, it decodes the order received and if needed, an associated message (of type `wrapstring`). Orders are the following:

- `START` to let the system know about the internal modules.
- `GO` to get one reaction of the system.
- `INTERNAL` to add an internal module in the system.
- `EXTERNAL` to add an external module in the system. This feature will be described later.
- `REMOVE` to remove a module known by its "pid" (process identifier). This "pid" has been returned by the kernel when the module has been added in the system.

The kernel is implemented by the following reactive process:

```
rprocess int Kernel(order,msg)
int order; wrapstring msg;
{
   int pid;
   for(;;){
      rswitch(order){
      case START:
         CreateInternalModules();
         break;
      case GO:
         exec Engine();
         break;
      case INTERNAL:
         DecodeIntern(msg);
         pid = AddInternalTask(InternProgram,ArgList);
         break;
```

```
      case EXTERNAL:
         DecodeExtern(msg);
         pid =
           AddExternalTask(ExternProgram,ExternMachine,ArgList);
         break;
      case REMOVE:
         RemoveTask(msg);
         break;
      }
      stop pid;
   }
}
```

Commands use the **Kernel** reactive process as a *reactive task* For example, code for the **internal** command is:

```
char Host[128];

rtask int Kernel(order,msg)
int order; wrapstring msg;
{rprocess named "Kernel" on Host;}

main(argc,argv)
int argc;
char *argv[];
{
   int pid;
   if (argc!=2){
      printf("usage: %s command\n",argv[0]);
      exit(1);
   }
   gethostname(Host,128);
   into pid react Kernel(INTERNAL,argv[1]);
   printf("pid:%d\n",pid);
   exit(0);
}
```

## 4.2 Execution of the Fibonacci system

Here is a session using the fibonacci system: First, the **fibonacci** command is run and the **start** command is executed to start the system. Then the four internal modules are added in (the kernel returns their pid), and finally the system is activated four times.

```
cma$ fibonacci&
cma$ start
cma$ internal "Print,A"
pid:0
cma$ internal "Const11,A,B"
pid:1
cma$ internal "Plus,A,B"
pid:2
cma$ internal "Shift,A,B"
pid:3
cma$ go
1 cma$ go
2 cma$ go
3 cma$ go
5 cma$
```

Notice that shared variables are declared as soon as their names appear for the first time in **internal** commands

The **go n** command is equivalent to running **go n** times. Thus, one have:

```
cma$ go 5
8 13 21 34 55 cma$ go 5
89 144 233 377 610 cma$
```

The system does not work anymore when a module is removed (here we remove **Shift** whose pid is 3):

```
cma$ remove 3
cma$ go 5
610 610 610 610 610 cma$
```

All becomes correct as previously when the missing component is put in the system (notice that a new pid is returned):

```
cma$ internal "Shift,A,B"
pid:4
cma$ go 5
610 987 1597 2584 4181 cma$
```

A new internal **Print** module sharing **A** can be added:

```
cma$ internal "Print,A"
pid:5
cma$ go 5
6765 6765 10946 10946 17711 17711 28657 28657 46368 46368 cma$
```

**Distribution.** Modules instead of being internal, can be external, that is running as stand alone processes (may be on different machines). In this case, shared variables that are parameters of external modules are transmitted through the network. The important point is that the only thing to change to define an external module instead of an internal one, is to use the **autonomous** macro instead of the **module** macro. For example, the following code defines **Shift** as an external module (notice that **autonomous**, as **module**, is ended by the number of shared variable parameters).

```
autonomous2(Shift,A,B)
{
   rauto int v;
   for(;;){
      into v exec Read(A);
      exec Write(B,v);
   }
}
endautonomous
```

This code can be compiled and run as it is.

In the following session, the fibonacci system is defined with **Shift** and **Print** as external modules running on the "shiva" machine and with **Const11** and **Plus** as internal modules.

```
cma$ fibonacci&
cma$ start
cma$ external "shiva,Shift,A,B"
pid:0
cma$ external "shiva,Print,A"
pid:1
cma$ internal "Const11,A,B"
pid:2
cma$ internal "Plus,A,B"
pid:3
cma$ Print&
[4]     3647
cma$ Shift&
[5]     3648
cma$ go 10
1 2 3 5 8 13 21 34 55 89 cma$
```

Of course, the external modules may be run on distinct machines.

# 5 Conclusion

We have considered a class of parallel and distributed systems where communication is through shared variables.

One main characteristic of these systems is the existence of a notion of a global instant shared by all system components. During one instant, all readers of a shared variable always get the same value, that is the combination of all values written in it during the instant. One thus get a model in which data coherency and determinism are preserved.

The other main characteristic is that systems can dynamically change by adding or removing components.

The model has been implemented using Reactive C. In this implementation, system components can be freely distributed on distinct machines communicating through a network.

# References


[1] G. BERRY, 'Real Time Programming: Special Purpose or General Purpose Languages', *Information Processing 89*, G.X. Ritter ed., Elsevier Science Publishers B.V. (North-Holland), pp 11-17 (1989)

[2] G. BERRY, G. GONTHIER, 'The Esterel Synchronous Programming Language: Design, Semantics, Implementation', INRIA Report **842** (1988), to appear in *Science of Computer Programming.*

[3] F. BOUSSINOT, 'Reactive C: An Extension of C to Program Reactive Systems', *Software-Practice and Experience*, vol. **21**(4), 401-428 (1991)

[4] F. BOUSSINOT, 'Réseaux de Processus Réactifs', INRIA Research Report **1588**, (1992)

[5] F. BOUSSINOT, 'Reactive Programming and the "Reactive C" Language', EMP-CMA Report (1992)

[6] P. CASPI, D. PILAUD, N. HALBWACHS AND J. PLAICE, 'Lustre, a Declarative Language for Programming Synchronous Systems', *Proceedings ACM Conference on Principles of Programming Languages*, Munich (1987)

[7] D. HAREL, 'Statecharts: a Visual Approach to Complex Systems', *Science of Computer Programming*, **8**(3), pp 231-275 (1987)

[8] D. HAREL, A. PNUELI, 'On the Development of Reactive Systems: Logic and Models of Concurrent Systems', *Logics and Models of Concurrent Systems*, NATO ASI Series F **13**, K.R. Apt, ed., Springer-Verlag, New-York, 477-498 (1985)

[9] P. LE GUERNIC, A. BENVENISTE, P. BOURNAI AND T. GAUTHIER, 'Signal: a data-flow oriented language for signal processing', *IEEE Transactions on ASSP* **34**(2), 362-374 (1986)

[10] F. MARANINCHI,'Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra',International Workshop on Automatic Verification Methods for Finite State Systems,LNCS **407**,Springer Verlag (1989)

Unité de Recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399