SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(4), 401–428 (APRIL 1991)

Reactive C: An Extension of C to Program Reactive Systems

FRÉDÉRIC BOUSSINOT
Ecole Nationale Supérieure des Mines de Paris, Centre de Mathématiques Appliquées, Sophia-Antipolis, 06565 Valbonne, France

SUMMARY

Reactive systems are interactive programs that react continuously to sequences of activations coming from the external world. Reactive programming leads to a new programming style where one programs in terms of reactions to activations and reasons in a logic of instants. This paper describes an extension of the C programming language called RC (for 'Reactive C') to program reactive systems. The language RC is described, then some programming examples are given to illustrate the reactive approach. The main RC notions come directly from the Esterel synchronous programming language. Finally, the Esterel and RC languages are compared.

KEY WORDS Reactive system C programming language Parallelism

INTRODUCTION

Reactive systems have been defined by Harel and Pnueli as systems that are supposed to maintain an ongoing relationship with their environment. [1] Such systems do not lend themselves naturally to description in terms of functions and transformations: they cannot be described adequately as computing a function from an initial state to a terminal state. On the contrary, behaviours of reactive systems are better seen as reactions to external stimuli. The role of reactive systems is to react continuously to external inputs by producing outputs. For example, man–machine interface handlers or computer games fall into the category of reactive systems.

Recently, several languages have been designed for reactive programming. Among these reactive languages, we can cite the imperative language Esterel, [2] two data-flow languages, Lustre [3] and Signal, [4] and the graphical specification formalism Statecharts. [5]

These languages do not use an *absolute* time as is the case, for example, in Ada [6] with the delay statement. Instead, they use a *logical* time divided into *instants*, which are moments when programs react. In Esterel, we would write `await 3 S` to wait for the third instant where the signal S is present (no matter what the signal S denotes). This approach leads to a new programming style where one programs in terms of reactions to activations and one thinks in a logic of instants.

C [7] is a very widespread and simple programming language. It is used in many areas and in particular for interactive programs. This paper describes a new extension

0038–0644/91/040401–28$14.00 *Received 25 August 1989*
 Revised 14 December 1989 and 9 May 1990

of C for reactive programming, * called RC for 'Reactive C'. The intention is to show that reactive programming can be done in C in a rather natural way by introducing only a small set of new concepts. The effort to learn RC, for those who know C, should be very small. Mainly, RC adds parallelism, exceptions and reactive statements to C. The semantics of the reactive part of RC are described in a mathematical framework, using conditional rewriting rules. [8] At present, RC is implemented as a preprocessor which generates pure C.

In this paper, we first give motivation for reactive programming, then we describe the RC language. We define in RC some communication mechanisms such as semaphores, broadcast communications and CCS [9] communication ports. As an illustration of the RC programming style, we give the complete code for a small game to test a person's reflexes, as well as a minimal environment in which to use it. Another example is given that uses parallelism. We compare the RC solution with a solution in Ada for the same example. Finally, a comparison of RC with Esterel is discussed.

MOTIVATION FOR REACTIVE PROGRAMMING

Determinism, parallelism and sequential execution

Most systems decompose naturally into concurrent communicating subsystems. To reflect this decomposition, concurrency and parallelism have been introduced in some recent programming languages (Ada, for example). These language and their compilers become more complex, but programs are clearer and easier to analyse. In general, parallelism and concurrency give rise to non-determinism: a program can have several distinct behaviours and the operating system must choose one of them arbitrarily. Moreover, the semantics of some constructs (e.g. the select in Ada), are explicitly non-deterministic. On the other hand, parallelism leads to execution-time overhead: run-time concurrency and task and communication management lower run-time performance.

There are several advantages if one stays with the deterministic case. Deterministic programs are simpler: when reasoning, there is no need to take into account the choices that the operating system will make. Also, deterministic programs have reproducible behaviours, which is a great advantage for tests and validations. Reactive languages such as Esterel or RC allow parallelism but preserve determinism. Moreover, programs are executed sequentially: run-time concurrency is not needed. One of the major gains with the reactive approach is to reconcile parallelism, determinism and sequential execution.

Termination and cancellation

Concurrency introduces two problems: first, determining a termination notion common to distinct processes; secondly, giving a way to a process to 'kill' another one. Reactive languages are based on an *instant* notion that is shared by all processes.

* A first version of this paper has appeared as Reference l0.

So, termination becomes perfectly clear and means the end of the current instant. Specific statements to manage instants and to define behaviours by reference to instants can be introduced in a safe way.

The 'killing' ability, the means of cancelling an activity, exists in some languages as Ada. However, it is well known that the Ada abort statement has complex semantics which even includes the case where the aborted task continues its execution forever [11] For example, suppose one has to code in Ada a 'watchdog' on a process P. A solution is to create a task that executes an abort(P) statement. Another solution is to control P execution by using the 'rendezvous' mechanism of Ada. In both cases, the semantics are complex and there is no guarantee that P will be killed at the right moment.

Reactive programming is also an attempt to solve the cancellation problem: for example, 'watchdogs' are directly included as first-class concepts, with precise and simple semantics.

RC DESCRIPTION

The time in RC

RC programs are pieces of code made from *reactive statements.* Executing a reactive statement makes it react: reaction means execution. A sequence of reactions is produced when the reactive statement is executed, then executed a second time, and so on. More precisely, a set of *control points is* associated with each reactive statement. These control points retain their values from one execution to the next (they can be viewed as C static variables). Execution of a reactive statement starts from the actual control points and reaches new control points from which the execution will restart at the next execution. Thus, reactive statements do not necessarily behave in the same way at each execution, even if the environment has not changed.

In RC we have a *logical* notion of time; time is made of instants that are the moments where reactive statements are executed. An instant begins when the execution starts and it is finished when the execution ends. With this underlying notion of time, we can speak of the first instant of a reactive statement, i.e. the first time it is executed. Similarly, we can speak of the second instant, the third instant, etc. The behaviour of a reactive statement is defined completely when one can describe how it behaves at the first instant (when it is executed for the first time), at the second instant (when it is executed for the second time), at the third instant, and so on.

In RC as in C, there is nothing that corresponds to the 'real' time, i.e. the time we use in everyday life, and that is counted, for example, in seconds. To deal with real-time problems, one must be able to associate the logical time with the real one, i.e. to 'generate instants' (to execute and re-execute the program continuously) with a sufficient rapidity.

Termination in RC

Some reactive statements have a limited lifetime: they *terminate.* To execute a terminated statement does nothing at all: it produces the empty reaction. A termin-

ated reactive statement remains terminated unless it is reset. When reset, a reactive statement behaves as if this is its first instant. Some other reactive statements, for example infinite loops, never terminate. There is a special case, when a reactive statement terminates during the first instant (that is, it terminates the first time it is executed): in this case, we say that the statement *terminates instantaneously.*

In a sequence of reactive statements, the control leaves a terminated component and during the same instant reaches the next component. For example, in the sequences `s1 s2` the control reaches `s2` when (and if) `s1` terminates. Moreover, `s2` begins execution at the same instant `s1` terminates.

We now give an informal semantics of reactive statements, i.e. we define how they behave at each instant, and when they terminate.

Reactive procedures

Reactive procedures are built from reactive statements and indicate the way to execute them. Reactive procedure instants are identified with their calls.

To illustrate the reactive-procedure notion, consider the following C function that prints 'hello, world' at each call:

```
hello(){
  printf( "hello, world\n " );
}
```

Now, suppose one wants to print 'hello, world' during the first call and 'I repeat: hello, world' during the second call. One writes in RC

```
rproc Hello(){
  printf( "hello, world\n " );
  stop;
  printf( "I repeat: hello, world\n " );
}
```

The `rproc` keyword introduces a *reactive procedure definition.* The `stop` statement is the first example of a *reactive statement.* It stops the execution for the current instant. Execution will restart at the next instant from the statement that follows `stop`. Therefore, the reactive procedure `Hello` reacts at the first instant by printing 'hello, world'. At the second instant, it prints 'I repeat: hello, world' and terminates. At the next instant, nothing will be printed because the procedure is terminated.

C expressions such as C functions calls are considered to be instantaneously-terminating reactive statements. One can identify the `stop` reactive statement in `Hello` with its control point (in general, there is more than one control point associated with a reactive statement, because of the RC parallel operator (described later) that splits the control into several flow paths).

Reactive procedures are called using the `exec` statement.* For example, to call the previous reactive procedure, one just writes

* We choose to use a specific keyword to distinguish reactive procedure calls from C functions calls, to insist on their difference (even though they are related notions). It also simplifies the preprocessor implementation,

```
exec Hello();
```

In a sequence of reactive statements, control reaches the next component as soon as the previous component terminates. Suppose that the Bye reactive procedure is defined by

```
rproc Bye(){
  stop ;
  printf( ″ Bye!\n ″ );
}
```

To execute Bye after Hello, one can write:

```
rproc Seq(){
  exec Hello();
  exec Bye();
}
```

The procedure Seq prints ‘hello, world’ at the first instant. The control point becomes the stop in Hello. At the second instant, Seq prints ‘I repeat: hello, world’ and then Hello terminates. In the same instant, the procedure Bye is started and the new control point becomes the stop statement in Bye. At the third instant, Seq prints ‘Bye!’ and terminates. We can represent Seq behaviour by the drawing in Figure 1.

Looping statements

Loops are essential to code interactive programs that run forever. In RC, there are three kinds of loops: infinite loops, every loops, and finite repeat loops. Infinite loops never terminate, and raising an exception is the only way to exit from them (see below).

loop. Infinite loops are coded with the loop construct: when the body of a loop terminates, it is immediately reset and executed another time. Consider for example the following reactive statement:

```
loop exec Hello();
```

The message ‘hello, world’ is printed at the first instant and the control point becomes the stop statement in Hello. At the second instant, execution starts from

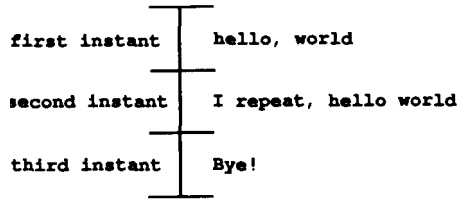

Figure 1.

the previous control point and ‘I repeat: hello, world’ is printed. Now, Hello is terminated; it is immediately reset and restarted and ‘hello, world’ is printed again. Therefore, at each instant except the first one, ‘I repeat: hello, world’ then ‘hello, world’, are printed in this order. This gives the drawing in Figure 2.

This example shows that it is mandatory to be able to reset reactive statements. Otherwise, once the Hello procedure has terminated for the first time, the overall loop would loop forever without printing anything.†

Warning! The execution of a loop construct loops forever during the same instant if its body always terminates instantaneously, as (for example) C expressions do. Thus, the following reactive statement loops forever at the first instant:

```
loop printf("looping !");
```

Its equivalent in C would be:

```
1: printf("looping !"); goto 1;
```

every. In reactive programming, one often wants to repeat an action every time a condition is true. The every reactive statement accomplishes this. Execution of the body of an every statement begins at the first instant the associated condition is true. The body is reset and restarted every time the condition is true, even if it was not terminated. Consider the following reactive statement:

```
every (Cond) exec Hello();
```

The execution of Hello begins at the first instant Cond is true and ‘hello, world’ is printed at that instant.

(a) If Cond is false at the next instant, ‘1 repeat: hello, world’ is printed at that instant. The procedure Hello will be restarted at the first instant Cond will become true.
(b) If Cond is again true at the next instant, Hello is reset and restarted and ‘hello, world’ is printed one more time.

Figure 3 shows an example of this behaviour.

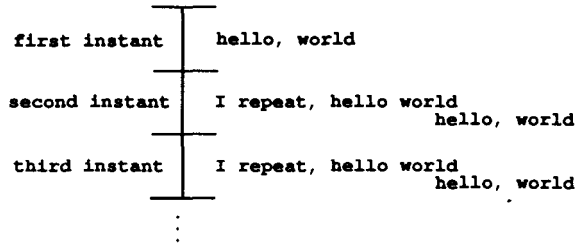

Figure 2.

† We choose not to use the C for(; ;) construct to code reactive loops to simplify RC implementation (otherwise, the body of the loop must be analysed to decide if it is a reactive loop, i.e. one that must reset its body when it terminates).

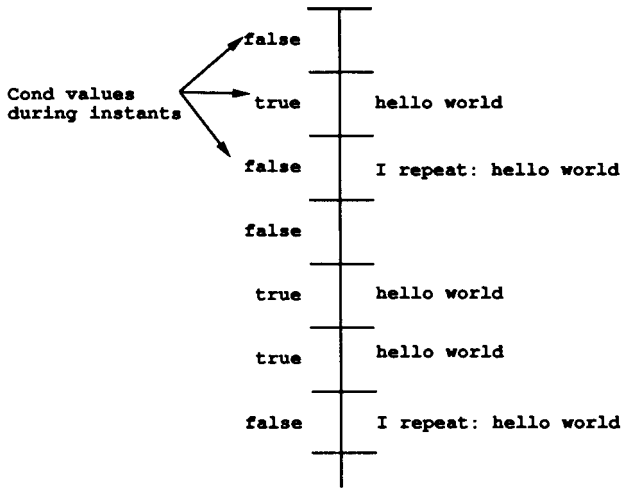

Figure 3.

repeat. Finite loops are coded using the repeat statement. For example, the Work procedure is executed ten times in

```
repeat (10) exec Work();
```

Control statements

When programming in a reactive style, we often want to control *at each instant* the execution of some reactive statements. In RC, the two reactive statements watching and select do this job. In the watching statement, control means supervision; in the select statement, it means choice.

watching. The watching statement allows the definition of 'watchdogs' in RC. A watching statement supervises its body in accordance with a boolean condition: it 'kills' its body as soon as the condition becomes true. It terminates either when its body terminates or when the condition is true. For example, consider:

```
watching (END) exec Work();
```

This statement terminates instantaneously if END is true at the first instant. Otherwise, Work is started. If Work terminates at the first instant, the watching statement also terminates. The behaviour is similar at the next instants: if END is true the overall statement terminates; otherwise, Work execution is resumed. So, Work is 'killed' as soon as END is true.

To wait for a condition Cond to be true, one can write

```
watching (Cond) loop stop;
```

This statement terminates if and only if Cond is true. More simply, one can use the await reactive statement which does exactly this job: the previous statement is in fact equivalent to the reactive statement:

```
await(Cond);
```

A ‘last will’ statement can be defined for the body: this statement is executed when the body is killed. It is prefixed by the timeout keyword because, often, ‘killing’ conditions reflect timing constraints. For example, suppose the ‘last will’ of Hello is to say ‘Bye!’:

```
watching (END) exec Hello();
timeout exec Bye();
```

There are three possibilities:

1. END is true at the first instant. Then the overall statement behaves as Bye.
2. END is neither true at the first instant nor at the second instant. Then the overall statement behaves as Hello.
3. END is false at the first instant, but true at the second instant. Then the overall statement prints ‘hello, world’ at the first instant. At the second instant, END is true so the execution of Hello is aborted and execution of Bye is started. Therefore, the overall statement prints nothing at the second instant. Finally, it prints ‘Bye!’ at the third instant and then terminates.

The behaviour in this last case is shown in Figure 4.

select. The select statement has two components and at each instant it selects which component is to be executed, with respect to a boolean condition. By contrast to if conditionals that make selection only at the first instant, select does it at all instants. The select statement terminates when the selected statement terminates. For example, the two reactive procedures P1 and P2 are executed alternately by

```
select (x = !x)
   exec P1();
   exec P2();
```

Suppose one wants to suspend the execution of a reactive procedure P when SUSP becomes true, and to resume it when RST becomes true. One can simply write

```
select (Control(SUSP, RST))
   exec P();
   loop stop;
```

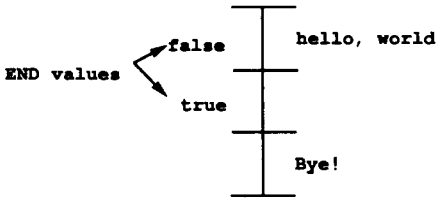

Figure 4.

with the C function Control defined by

```
int Control(S, R)
int S,R;
{
   static act = TRUE;
   if (act && S) return act = FALSE;
   if (!act && R) return act = TRUE;
   return act;
}
```

Exceptions

A general exception mechanism is defined in RC, to treat 'exceptional' cases. When such an exceptional case is encountered, one executes a raise statement which halts the current execution. Control is then recovered at the same instant by the nearest surrounding catch statement. For example, in the following statement, the error exception is raised if the variable X becomes equal to zero before the variable Y:

```
watching (X= =0)
   { watching (Y= =0) loop stop; }
timeout raise error;
```

To handle an exception named abnormal, when it is raised, one writes

```
catch abnormal
   exec Work();
handle exec ReportError();
```

If Work terminates, the catch statement also terminates. ReportError is executed onlv if the abnormal exception is raised during Work execution.

Parallelism

The par reactive statement introduces parallelism into RC: it allows execution of several reactive statements during the same instant. However, the order of execution of the two components of a parallel statement is fixed and specified by the syntax of the par statement. A control point is associated with each component of a par statement. At each instant, the first component is executed, followed by the second component. At the next instant, the execution restarts from the two control points where the execution has been stopped previously. Execution of a parallel statement is halted as soon as an exception is raised. A parallel statement is terminated when its two components are themselves terminated. Consider the statement

```
par
   exec Hello();
   exec Bye();
```

At the first instant, it prints ‘hello, world’. At the second instant it prints ‘I repeat: hello, world’ and ‘Bye!’, in this order. Then, the reactive procedures hello and Bye are both terminated, so the par statement is also terminated. The behaviour is shown in Figure 5.

If one component of a par statement never terminates, the par statement also never terminates. For example, the following statement never terminates (except if Work raises an exception):

```
par
  loop stop;
  exec Work();
```

However, when only one component terminates, the parallel construct behaves in the same way as the other component. For example, suppose Cfunct is a C function. Then

```
par
  Cfunct();
  exec Work();
```

is equivalent to the sequence

```
Cfunct(); exec Work();
```

Warning! Executing a reactive procedure in parallel with itself can lead to unclear situations.* For example, consider the following statement:

```
par
  exec Hello();
  exec Hello();
```

The Hello procedure is called twice at the first instant (and it is not reset between the two calls). The two messages ‘hello, world’ and ‘I repeat: hello, world’ are both printed at the first instant and the statement terminates instantaneously (remember that in contrast, Hello prints ‘hello, world’ at the first instant, ‘I repeat: hello, world’ at the second instant, and then terminates).

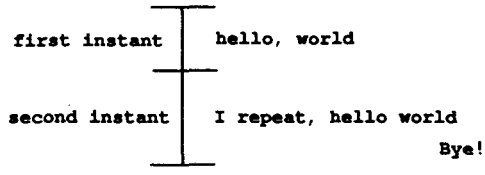

Figure 5.

* Perhaps it would be wise to prohibit such situations. The present RC implementation does not.

Micro instants

RC gives a way to break one instant into several *micro instants*. Conversely, micro instants can be joined together to form one unique instant. The suspend reactive statement suspends the execution for the current instant. But instead of compulsorily creating a new instant as stop does, execution can be restarted in the same instant, using the close statement. The close statement 'jumps over' the suspend statements that are in its body. It can be seen as the dual of suspend: it generates one unique instant (called 'macro instant') from several micro instants. Execution of the body of a close statement is pursued while there exists a control flow path that is suspended on some suspend statement. There is an implicit closure at the outermost level of an executable program, i.e. in the main function, so that micro instants are in fact not observable (which justifies the use of the term 'micro instant'). As example, consider the following parallel statement:

```
par
   { suspend; printf( "1" ); }
      printf( "2" );
```

This statement prints 2 at the first instant. At the second instant, it prints 1 then terminates. The behaviour is shown in Figure 6.

Up to this point, suspend behaves as stop. But consider the closure of the previous statement:

```
close
   par
      { suspend; printf( "1" ); }
         printf( "2" );
```

Now, the close statement terminates instantaneously after printing 2 then 1, in this order. Execution is as follows: the first branch is suspended and the second one is executed, so 2 is printed. Then, the first branch is resumed so 1 is printed, and the par statement terminates. Now, the behaviour becomes as shown in Figure 7.

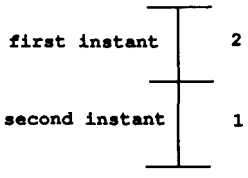

Figure 6.

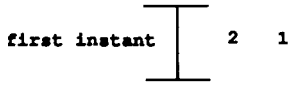

Figure 7.

The real usefulness of suspend comes from parallelism. Suppose that there are two processes placed in parallel in a system, and that each process has to monitor the other process to detect its failure. Here is the code using suspend for the first process

```
rproc P1 (){
  par
    exec 0bserveP2():
    loop { OK1=1; stop; }
}
rproc 0bserveP2(){
  loop{
    suspend;
    if(OK2) OK2=0; else printf( " P2 out " );
    stop;
  }
}
```

The code for the second process is similar:

```
rproc P2(){
  par
    exec ObserveP1 ();
    loop { OK2=1; stop; }
}
rproc ObserveP1 (){
  loop{
    suspend;
    if(OK1) OK1 =0; else printf( " PI out " );
    stop;
  }
}
```

Now the two processes can be put in parallel in any order. A possible version of the overall system is:

```
rproc System(){
  par
    exec P1 ();
    exec P2();
}
```

A failure is detected by a process if the other process does not set its OK variable. Using suspend, each process tests the OK variable *after* the other has the possibility of setting it. Thus, the order of execution of P1 and P2 becomes irrelevant and each process is able to detect immediately the failure of the other one.

The next section gives more examples of the use of suspend, in particular to code several communication and synchronization mechanisms.

COMMUNICATION MECHANISMS

Because of the presence of parallelism in RC, reactive statements can communicate and synchronize. We now describe several communication and synchronization mechanisms in RC. First we define basic semaphores. The division of time into instants leads to new synchronous semaphores which are coded using suspend.

Usually, communication in high-level formalisms such as CSP [12] or Ada is based on a 'hand shaking' mechanism: to communicate, processes must synchronize themselves explicitly and are suspended until the 'rendezvous' takes place. On the contrary, reactive formalisms, (Esterel, for example) are based on a *broadcast* communication mechanism. Broadcasting can be viewed as 'hand raising': to communicate, I raise my hand (with possibly a notice in it) and everybody can see it (and can read information on the notice). Moreover, raising my hand does not prevent me from carrying on with my job. Now, the question is: when may I lower my hand? The partition of time into instants allows a simple answer to this question: I lower my hand at the end of the current instant. We describe in RC two different kinds of broadcast. The first one comes directly from Esterel and is based on signals. The main difference with Esterel is that correct access to signals is dynamically checked in RC. These signals will be used to code the reflex game described later. The second form of broadcast, which we call 'radio', does not need dynamic checking, as emissions and receptions take one instant.

Finally, we consider an alternative to broadcasting that is a one–one communication through 'ports', and we show how to code it in RC.

Semaphores

Semaphores are well-known in parallel programming. [13] A semaphore is free or in use (initially, it is free). To use a semaphore S, a process must execute P(S). If the semaphore is free, the process can continue; if it is already being used by another process, it is stopped. A semaphore S becomes free when the process using it executes V(S). Semaphores can be defined by

```
typedef int Semaphore;
#define P(sem) {await(!sem); sem = 1;}
#define V(sem) sem = 0;
```

There is a problem with semaphores as defined previously: situations exist where a process waiting for a semaphore cannot take it at the instant it becomes free. For example, consider the following statement, where S is a semaphore:

```
par
  { stop; P(S); exec R(); V(S); }
  { P(s); stop; v(s); }
```

The execution of R does not begin at the second instant although S is released during this instant: this is because S is released after the first parallel branch has tried to use it.

Synchronous semaphores eliminate the previous problem: they can be taken as soon as they are released. They are coded using suspend:

```
typedef int SyncSemaphore;

#define SyncP(sem)
  catch Go
    loop
      {suspend; if(sem) stop; else raise Go;}
  handle sem = 1;

#define SyncV(sem) sem = 0;
```

Signals

Signals define global data with restricted access. A signal can be emitted, read and reset. There is the *dynamically checked* restriction that once a signal has been read, no emission of it is allowed unless it is reset. Hence, all emissions must be performed before the signal is read, and all readers necessarily read the same information. Signals therefore implement a broadcast communication mechanism. More precisely, a signal is an object on which the following operations are defined:

1. Emission: emit(S).
2. Cancellation of previous emissions: reset(S).
3. Presence test: present(S) is an expression whose value is 1 if signal S has been emitted; otherwise its value is 0.
4. Valued emission: emitval(S,exp); The signal S is emitted and its value becomes the value of exp. Signal values are always of integer type int.
5. Use of value: valof(S) returns the value of S.

The restriction on signal use is that *all emissions must be performed before the first test for the presence of a signal and before the first use of its value.* A run-time error is raised when this restriction is violated. This is the case for the following three statements:

```
if (present) emit(S);

await (present(S)); emit(S);

emitval(DISPLAY, valof(DISPLAY)+1);
```

The _sigabort function is called every time the signal restriction of use is violated. It can be user-defined, and in the default case it aborts execution by calling the C function abort.

Multiple emissions

A signal can be emitted several times during the same instant, with possibly different values. For example, consider

```
emitval(SIG,1 ); emitval(SIG,2);
```

By default, the last emission overrides the previous ones. So, in the example, the value of SIG would be 2. As in Esterel, one gives the user the possibility to change this situation by associating a combine function with a signal. Then, emitted values will be combined using this function. One associates a combine function f with a signal S by executing the call

```
combine(S,f);
```

Now, if S has been emitted already and has the value v,emitval(S,e) changes the value to f(v,e). As an example, if plus denotes addition, the value of SIG is 3 after executing

```
combine(SIG, plus);
emitval(SIG,1);
emitval(SIG,2);
```

Single signals

Following Esterel, one calls *single* a signal that cannot be emitted more than once without been reset. S becomes single after:

```
combine(S,abort);
```

Notice that, in contrast with Esterel, single signals emitted more than once are checked at run-time.

Counting signal occurrences

In RC, it is possible to react to several signal occurrences using an auxiliary counting variable. For example in the following statement, the error exception is raised if READY is not present in less than LIMIT occurrences of the signal MS:

```
watching (present(MS) && 0 = = LIMIT- -)
   await (present(READY))
timeout raise error;
```

(Notice that LIMIT- - is executed only when MS is present, because of the semantics of && in C.)

Boolean expressions on signals

One easily expresses statements reacting to boolean expressions made of several signal-presence tests. For example, in the following statement, P is executed each time S is absent:

```
every (! present(S)) exec P();
```

In the following statement, P is killed if S3 is present or if S1 and S2 are simultaneously present:

```
watching((present(S1) && present) || present(S3))
  exec P();
```

Radio

The *radio* communication mechanism we now define implements another form of broadcast communication. Contrary to the previous mechanism based on signals, dynamic checking is no more needed to preserve the broadcast discipline. To emit or to read information in 'radio' mode takes one instant. More precisely, radio communication has the following characteristics:

1. EMIT(radio, n): the (integer) message n is emitted on radio and this takes one instant.
2. LISTEN(radio,x): one listens for radio and the variable x receives its value. This takes one instant.
3. Values of messages emitted during the same instant are added.
4. When no message is emitted during an instant, the value that listeners observe is 0. Messages are lost when not caught at the instant they are emitted.
5. Messages are broadcast: everybody receives the same information at the same instant.

The precise definition of the radio communication is:

```
typedef struct radio{ int val; int em; } radio;

#define EMIT(radio,exp) {
  radio. em = 1; radio. val += exp; suspend;
  suspend; radio.val = 0; radio.em = 0; stop;
}

#define LISTEN(radio.var) {
  suspend; if(radio, em) var = radio; stop;
}
```

To emit on a radio, one first notices that there is an emission (radio.em set to 1) and then one suspends execution for two instants. Thus, all emitters have the possibility to emit and all receivers detect that there are emissions and take the same value, after all emissions are concluded. In the end, the radio is reset by all emitters. The ending stop statements in LISTEN and EMIT imply that to listen for a radio or to emit on a radio takes one instant. This feature avoids the necessity of dynamic checking as needed for the other modes of signaling.

Ports

In contrast to broadcasting, CCS [9] communication is one–one. Processes communicate through *ports;* a sender process and a receiver process are associated with each port. Communication is instantaneous and is blocking: the sender must wait for the receiver to be ready to continue; in the same way, the receiver is blocked when the sender is not ready. The coding of ports uses the synchronous semaphores defined above.

```
typedef struct port{
            int val;
            SyncSemaphore send;
            SyncSemaphore ret;
  } port;

#define INIT(port) port. send = port. rec = 1;

#define SEND(port,exp) {
  port,val = exp;
  SyncV(port.rec)
  SyncP(port.send)
}
#define RECEIVE(port,var) {
  SyncV(port.send)
  SyncP(port.ret)
  var = port.val;
}
```

To illustrate the port communication, consider the following procedure which prints 1 and terminates instantaneously:

```
port p;

rproc Com(){
  static int z,z1;
  INIT(p)
  par
    { RECEIVE(p,z) SEND(p,z+1) }
    { SEND(p,0) RECEIVE(p,z1 ) }
  printf( "%d", z1);
}
```

The value 0 is sent then received in z; then 1 is added to z and the new value once more sent and received in z1. All these actions are done in the same (macro) instant.

FIRST EXAMPLE: A REFLEX GAME

In this section we present a small example to illustrate the RC programming style. The example is a reflex measuring game which has a simple normal behaviour and several exception cases. * We describe the RC code for the reflex game. Then, to illustrate how to interface it, we give the C code that has been used to run it.

There are two buttons, three lamps, a bell and a numerical display. The game consists of testing player reflexes by a series of measurements. Each measure is as follows: when ready, the player presses a button; after a while, the game lights up

* This example is coded in Esterel in Reference 14.

a lamp; then the player must press a button as fast as he can. More precisely, the player controls the game with three commands:

(a) putting a coin in a COIN slot starts the game
(b) pressing the READY button indicates that the player is ready to play
(c) pressing the END button ends a measure.

The game reacts in the following way:

1. The GO lamp lights up to signal the beginning of a measure.
2. Reflex times measured are displayed on DISPLAY.
3. The GAME-OVER lamp lights up at the end of a series a measures, i.e. when the game is finished.
4. The TILT lamp lights up and the game is finished when the player tries to cheat or when he abandons the game.
5. The bell RING_BELL rings when the player makes a mistake by confusing the READY and END buttons.

Moreover:

(i) A new game is started afresh every time a coin is inserted.
(ii) Each game is composed of a fixed number MEASURE_NUMBER of measures. When all measurements have been carried out, the average reflex time is displayed.
(iii) Once READY has been pressed, GO lights up after a random delay. The time measure begins when GO lights up and ends when the player presses END.
(iv) The player abandons if he takes more than LIMIT_TIME instants to press the right button. The player cheats if, after he has pressed READY, he presses END before GO lights up.

RC code

We use the previously-defined signals (in a file rcsignal. h) to represent the buttons, the lamps and the bell.

```
#include ″ rcsignal.h ″

signal READY, COIN, END,
   DISPLAY, GO, GAME_OVER, TILT, RING_ BELL;
```

There are three external integer variables: MEASURE_NUMBER, LIMIT_TIME and PAUSE_LENGTH, which holds the delay before the final display of the average measured time:

```
extern MEASURE_NUMBER, PAUSE_ LENGTH, LIMIT_TIME;
```

The reactive procedure GAME is the main procedure. We count instants that we identify with GAME calls, and display numbers of instants. The variable TOTAL_TIME holds the sum of the numbers of instants the player has taken.

GAME begins by printing a quick description of the game, using the external C function PrintOut. As the game is started afresh whenever a coin is inserted, GAME

is coded with an every(present(COIN)) statement. The exception processing is naturally coded with a catch statement that calls the C function abnormal_game_over in case of error. A normal game is a series of measures coded with a repeat statement. The game ends with execution of finalDisplay.

```
static int TOTAL_TIME;

rproc GAME(){
  PrintOut( ″ A reflex game ... c to start, q to stop. \r\n ″ );
  PrintOut( ″ Press e as fast as possible after GO! ,\r\n ″ );
  every (present(COIN))
    catch error
      { repeat (MEASURE_NUMBER) exec measure();
        exec finalDisplay();
      }
    handle abnormal_game_over();
}
```

In case of error, the two signals TILT and GAME_OVER are emitted and TOTAL_TIME is reset to 0.

```
abnormal_game_over(){
  TOTAL_TIME = 0;
  emit(TILT);
  emit(GAME_OVER);
}
```

A measure consists of two phases. During the first phase, one waits for READY. During the second phase, one waits for END. A prompt message is printed at the beginning of each measure. To avoid interference with score printing, this is not done at the first instant but at the second (using a stop statement).

```
rproc measure(){
  stop;
  PrintOut( ″ press r when ready\r\n ″ );
  exec phase1(); exec phase2();
}
```

In phase1, one waits for READY during at most LIMIT_TIME instants. To count instants, we declare the variable COUNTER, which must absolutely be static, otherwise its value would not be retained from one instant to the next. At each instant, COUNTER is decremented and tested for zero by a watching statement. During the waiting, the player's mistakes are detected (reactive procedure beepOn). After READY has been pressed, one waits for a random number of instants (call of the external C function RANDOM). During the waiting, the player's cheatings and mistakes are detected (reactive procedure testEnd).

```
rproc phase1 (){
  static COUNTER;
```

```
  COUNTER = LIMIT_TIME;
  watching (0 == COUNTER- -)
    { watching (present(READY)) loop exec beepOn(END); }
  timeout raise error;
  COUNTER = RANDOM();
  watching (0 = = COUNTER- -) { stop; loop exec testEnd(); }
}
```

The reactive procedure beepOn has a parameter which is a signal. After one instant, the bell rings if the parameter is present.

```
rproc beepOn(s)
signal s;
{
  stop;
  if (present(s)) emit(RING_BELL);
}
```

The reactive procedure testEnd performs two actions: the exception error is raised if END is pressed and the bell rings if READY is pressed.

```
rproc testEnd(){
  if(present(END)) raise error;
  exec beepOn(READY);
}
```

In phase2 one begins by lighting up the GO lamp. Then, one waits for END during at most LIMIT_TIME instants. During the waiting, player mistakes are detected (reactive procedure beepOn). This behaviour is similar to the one in phase1 except that END and READY are exchanged. Finally, the reflex time is displayed and is added to TOTAL_Tl M E.

```
rproc phase2(){
  static COUNTER;
  COUNTER = 0;
  emit(GO);
  watching (LIMIT_TIME == COUNTER++)
    { watching (present(END)) loop exec beepOn(READY); }
  timeout raise error;
  emitval(DISPLAY,COUNTER);
  TOTAL_TIME + = COUNTER;
}
```

After PAUSE_LENGTH instants, the average time TOTAL_TIME / MEASURE_NUMBER is displayed and GAME_OVER is emitted.

```
rproc finalDisplay(){
  static COUNTER;
```

```
    COUNTER = 0;
    await (COUNTER+ + == PAUSE_LENGTH);
    PrintOut( ″ **** final ″ );
    emitval(DISPLAY, TOTAL_TIME / MEASURE_NUMBER);
    emit(GAME_OVER);
    TOTAL_TIME = 0;
}
```

Simulation of the reflex game

We now show the C code that supplies input for the reflex game and that processes its outputs. This simulation environment runs under Unix 4.2BSD.

One first gives suitable values to PAUSE_LENGTH and LIMIT_TIME constants, and to values returned by RANDOM.

```
#include(stdio.h)
#include(sys/types.h)
#include(sys/time.h)
#include ″ rcsignal.h ″

struct timeval t;

int MEASURE_NUMBER = 4;
int PAUSE_ LENGTH = 1000;
int LIMIT_TIME = 100000;

RANDOM(){ return random()% 10000; }
extern signal READY, COIN, END, DISPLAY,
                GO, GAME_OVER, TILT, RING_BELL;
```

Instants are calls of the reactive procedure GAME, so the main function is naturally a for(;;) C loop whose body represents one instant. Each instant consists of the following steps: input processing, GAME activation, output processing and then reset of all the signals:

```
main(){
   TheBeginning();
   for(;;){
      InputProcessing();
      exec GAME();
      OutputProcessing();
      ResetSignals();
   }
}
```

The ‘c’ key represents COIN, the ‘r’ key represents READY and the ‘e’ key represents END. One terminates the simulation session by typing the ‘q’ key. InputProcessing scans the standard input; it does not block when there is no character in this input (C function select is used with appropriate arguments).

```
InputProcessing(){
  int mask = (1((fileno(stdin));
  if(select(1,&mask,0,0, &t)){
    switch(getchar()){
      case 'c': emit(COIN); break;
      case 'r': emit(READY); break;
      case 'e': emit(END); break;
      case 'q': TheEnd();
    }
  }
}
```

Output messages are printed in accordance with the signals that are present.

```
OutputProcessing(){
  if(present(DISPLAY)){
    printf("score: %d",valof(DISPLAY));
    PrintOut("\r\n");
  )
  if(present(GO)) PrintOut( "GO !\r\n");
  if(present(TILT)) PrintOut( "\07TILT! !\r\n");
  if(present(GAME_OVER)) PrintOut(
    "Game over, Press c to restart, q to stop, \r\n");
  if(present(RING_ BELL)) PrintOut( "\07");
}
```

All signals are reset at the end of each instant.

```
ResetSignals(){
  reset(READY);
  reset(COIN);
  reset(END);
  reset(DISPLAY);
  rest(GO);
  reset(GAME_OVER);
  reset(RING_BELL);
  reset(TILT);
}
```

The two C funtions TheBeginning and TheEnd are used to set the appropriate options on the terminal. For good printing, PrintOut flushes the output buffer.

```
TheBeginning(){
  system( "stty raw -echo");
  t,tv_sec = t,tv_usec = 0;
}
TheEnd(){
```

```
    PrintOut( " It's more fun to compete ...\r\n " ).
    system( " stty -raw echo " ); exit(0);
}
Printout
char* ch;
{
    printf( " %s ", ch);
    fflush(stdout);
}
```

SECOND EXAMPLE: THE USE OF PARALLELISM

In this section we present an example using parallelism. After giving the RC code, we compare it with a solution in Ada.

The system we want to code has a very simple specification: inputs are sequences of TOP (synchronization) and REQ (request). Outputs are:

(a) OK for the first REQ between two TOPS, and NOK for the others REQs
(b) ALARM when there is no REQ between two TOPS.

Figure 8 shows how the system behaves.

There is no REQ between the two first TOPS, so ALARM is raised in response to the second TOP. There is only one REQ between the second and the third TOPS, so OK is emitted in response to REQ and no alarm is raised. Two REQs are present between the third and the fourth TOPS, so OK is emitted in response to the first REQ and NOK is emitted in response to the second. As there are REQs, there is no ALARM.

RC approach

We choose to represent input and output events as simple variables (we could equally well use signals, as in the reflex game). We suppose that, at the beginning of each instant, the output variables OK, NOK and ALARM, and the input variables that correspond to absent events, are set to 0. Conversely, we suppose that the input variables corresponding to present events are set to 1.

We give a modular solution: a request handler, an alarm handler and the global system that places the two previous handlers in parallel.

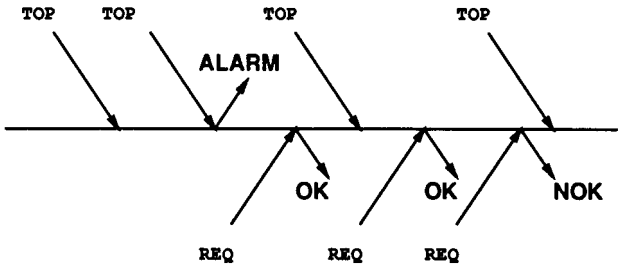

Figure 8.

The request handler has the following behaviour: at each TOP, it waits for a REQ and then sets OK to 1. Then, after the next instant, it sets NOK to 1 each time there is a new REQ. The RC code is

```
rproc REQ_HANDLER(){
  every(TOP){
    await(REQ);
    O K = 1 ;
    stop;
    every(REQ) NOK = 1;
  }
}
```

The alarm handler has the following behaviour: if TOP arrives before OK, then it sets ALARM to 1. In the opposite case, it waits for a TOP. This behaviour is restarted in the instant that follows the arrival of TOP.

```
rproc  ALARM_HANDLER(){
  loop{
    w a t c h i n g {
      await(TOP);
      ALARM = 1 ;
    }timeout await(TOP);
    stop;
  }
}
```

The request handler and the alarm handler are placed in parallel in the global system. The alarm handler uses the OK variable to test if there has been a REQ. At each instant, the request processing must therefore precede the alarm processing. The code is the following:

```
rproc TOP_REQ_HANDLER(){
  par
    exec REQ_HANDLER();
    exec  ALARM_HANDLER();
}
```

It is important to note that to invert the order of the two exec statements would be incorrect.

Ada approach

A modular solution in Ada defines two tasks, one to process requests, the other to process alarms. This architecture leads to two problems:

1. The scheduling of tasks generates non-determinism. There is no assurance that the program will always respect the order of events. For example, it may be possible that the program treats a REQ before a TOP even though the TOP has

arrived before the REQ. This can lead in some cases to forget an ALARM when two TOPS are present without any REQ between them.

2. Run-time task concurrency is necessarily time-consuming and things become worse when tasks communicate. The example exhibits two logically concurrent processes that are better implemented by an equivalent sequential code: this code is more efficient and respects determinism. In this case, run-time concurrency is neither mandatory nor desirable. The problem with Ada is that there is no way to produce sequential code from concurrent tasks. The user is forced to pay for run-time concurrency as soon as tasking is used.

The comparison with Ada shows the benefit gained with the reactive approach:

(a) Reactive statements such as 'watchdogs' are directly present. There is no need to code them in a more or less unnatural way.

(b) Parallelism can be used to code logically concurrent processes. The reactive approach does not force a choice of parallelism and non-determinism on one side, or determinism and sequentiality on the other side. It allows one to have parallelism and determinism and sequential execution together. Also, parallelism in the reactive approach does not imply run-time concurrency.

COMPARISON WITH ESTEREL

In Estere1 [15,2] communication is based on broadcast signals and processes can continue their execution during an instant even after receiving signals (this is called instan-*taneous broadcast* in Reference 16). For example, consider the following program fragment:

```
emit S || present S then emit T end
```

In this fragment, two agents are placed in parallel. The first one emits the signal S and the second one tests for the presence of S; if S is present, it emits the signal T. Following Esterel semantics, the two signals S and T are both emitted at the same instant; in Esterel one says that they are *synchronous.* To receive S (in fact, to test its presence) does not prevent the emission of T during the same instant. This can cause problems called 'causality cycles' in Esterel terminology. The basic example is

```
present S else emit S end
```

There is a contradiction: if we suppose that the signal S is present, then it is not emitted, and so it is not present. On the other hand, if we suppose that S is not present, then it is emitted, and consequently it is present.

Another example of a causality cycle, related to the *deterministic* aspect of Esterel, is

```
present S1 else emit S2 end
|
present S2 else emit S1 end
```

Now, there are two possible solutions: S1 is absent, and then S2 is emitted; and the other solution: S2 is absent and then S1 is emitted. This fragment is not deterministic, having two distinct behaviours. In Esterel, only deterministic programs are allowed. The Esterel compiler is based on mathematical semantics using conditional rewriting rules [15] It rejects programs having 'causality cycles' and transforms a correct program into a sequential automaton whose behaviour is equivalent to the source program. Automata produced can be analysed using existing automata verification systems. In short, we distinguish three important components in Esterel: first, the reactive part based on the division of time into instants; secondly, the broadcast communication; and thirdly, the fact that signal receptions do not block execution.

In this section, we compare RC to Esterel with respect to reactive statements, communication, data and process handling, and execution of programs.

Reactive statements

RC reactive statements come from Esterel. However, RC is more general than Esterel in several aspects:

(a) In RC, reactive statement conditions are boolean expressions and are not restricted to single signal presence tests as in Esterel.
(b) The RC stop and select statements cannot be directly (that is structurally) coded in Esterel.
(c) There is no possibility to define micro instants in Esterel as is the case in RC using the suspend reactive statement.

Parallelism and communication

In contrast to Esterel, the RC parallel operator is not commutative. For example, the following RC instruction will always print 1 then 2 in that order:

```
par
   printf( "1 ");
   printf( "2 ");
```

In Esterel, signals are the only way to synchronize branches of parallel instructions: no global variables are allowed. As in C, global variables and side-effects are fully allowed in RC, so communication can be totally unstructured and users are responsible for managing it. 'Instantaneous dialogues' are possible in Esterel. Consider for example the following Esterel program:

```
present S1 then emit S2 end
||
emit S1; present S2 then emit S3 end
```

First, S1 must be emitted, then it is tested and S2 is emitted; finally, S2 is tested and S3 is emitted. In the same instant the first branch communicates with the second branch, and conversely, the second communicates with the first. It is the Esterel compiler's job to find a proper interleaving allowing the dialogue to take place. By

contrast, interleavings of RC instructions in the same instant must be coded explicitly using the suspend statement.

Esterel instantaneous broadcasting cannot be expressed in RC; one has to find a compromise: if instantaneous reception is wanted, one has to pay for run-time checking (see the communication based on signals defined previously). On the other hand, if instantaneous reception is not wanted, run time checks become unnecessary (as in the radio communication). Note that Esterel 'causality cycles' (see Reference 15) come directly from the Esterel instantaneous broadcast feature.

Data and process handling

Data handling is not part of the definition of Esterel. Esterel has only a few primitive data types (such as integer and boolean) but no compound types such as records or arrays. Structured data must be manipulated by functions and procedures only known by their names in Esterel, and implemented in a host language. On the other hand, RC gives direct access to the whole power of C for data processing.

Dynamic process creation is not possible in Esterel. Further, synchronous and asynchronous processes cannot be mixed in Esterel where only synchronous communication is allowed. On the contrary, RC is much more flexible. It gives a unique frame where dynamic process creation and various communication mechanisms can be efficiently implemented.

Execution of programs

Several tools use the automata generated by Esterel to produce code in other programming languages (in particular C and Ada) or to produce entries for proofs or validation systems such as the AUT0 [17] system. In RC, programs are executed directly in contrast to Esterel where one executes finite automata generated by the compiler. Moreover, in RC there is no possibility of producing entries for proof or validation tools at the present time.

CONCLUSION

This paper describes a new C extension for reactive programming. The use of reactive statements is natural when program behaviours are defined as reactions to activations and by reference to instants. Several such programs are described in the text. RC includes, as first-class members, constructors for complex reactive behaviors such as 'watchdogs' which are difficult to code in other languages. Parallelism can be used in RC to code logically concurrent processes and it does not imply either run-time concurrency or nondeterminism. In RC deterministic parallel processes generate sequential code. Moreover, RC can be used to code specifications in the form of automata in a natural way.

There exists an experimental RC implementation as a C preprocessor. This implementation is written in C using *lex* and *yacc* (it uses a single pass). It runs under Unix on several machines. *

* At present: Sun 3, Sun 4, Vax, Gould and Hp.

RC is at present used as an implementation language for process algebra, in the SPECS (Specification and Programming Environment for Communication Software) consortium of the RACE European project.

It would be interesting to generate automata from the reactive parts of RC programs. Transitions of such automata would be pure C statements. These automata could be input to verification systems. It would also be interesting to extend C++ [18] in the same way RC extends the pure C language, to mix object-oriented programming with reactive programming.

ACKNOWLEDGEMENTS

I would like to thank G. Berry for helpful comments on the previous version of this paper.

REFERENCES

1. D. Harel and A. Pnueli, 'On the development of reactive systems', in K. R. Apt (cd.), *Logics and Models of Concurrent Systems,* NATO ASI Series F 13, Springer-Verlag, New York, 1985, pp. 477–498.
2. 'ESTEREL v3 manuals', Ecole des Mines, Centre de Mathematiques Appliquées, Sophia-Antipolis, 1988.
3. 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.
4. P. Le Guernic, A. Benveniste, P. Bournai and T. Gauthier, 'SIGNAL: a data-flow oriented language for signal processing', *IEEE Trans. Acoust. Speech and Signal Process.,* **ASSP-34.** (2), 362–374 (1986).
5. D. Harel, 'Statecharts: a visual approach to complex systems', *Science of Computer Programming,* **8,** (3), 231–274 (1987).
6. *The Programming Language Ada Reference Manual,* Lecture Notes in Computer Science 155, Springer-Verlag, 1983.
7. B. W. Kernighan and D. M. Ritchie, *The C programming language,* Prentice-Hall Software Series, Prentice-Hall, New Jersey, 1978.
8. F. Boussinot, 'RC semantics using rewriting rules'. *Technical Report,* 1989.
9. R. Milner, 'A calculus for communicating systems', *Lecture Notes in Computer Science* 92. Springer-Verlag, 1980.
10. F. Boussinot, 'A reactive extension of C', *INRIA Research Report 1027,* 1989.
11. A. Burns, *Concurrent Programming in Ada,* Cambridge University Press, 1985.
12. C. A. R. Hoare, 'Communicating sequential processes', *CA CM,* **21,** (8), 666–677 (1978).
13. E. W. Dijkstra, 'Co-operating sequential processes', in *Structured Programming,* Academic Press. New York, 1972.
14. 'ESTEREL v2.2 programming examples'. Ecole des Mines, Centre de Mathématiques Appliques, Sophia-Antipolis, 1987.
15. G. Berry and G. Gonthier, 'The Esterel synchronous programming language: design, semantics, implementation', *INRIA Report 842,* 1988; to appear in *Science of Computer Programming.*
16. G. Berry, 'Real time programming: special purpose or general purpose languages', in G. X. Ritter (cd.), *Information Processing 89,* Elsevier Science Publishers B. V., North Holland, 1989.
17. D. Vergamini, 'Vérification de réseaux d'automates finis par équivalences observationnelles: le systeme AUTO', *Thèse de doctorar,* Université de Nice, 1987.
18. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Company, 1986.