Real time programming : special purpose or general purpose languages

Gérard Berry

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N°1065

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

REAL TIME PROGRAMMING : SPECIAL PURPOSE OR GENERAL PURPOSE LANGUAGES

Gérard BERRY

Août 1989

RR-1065

Real Time Programming: Special Purpose or General Purpose Languages

Gérard Berry

Centre de Mathématiques Appliquées
Ecole Nationale Supérieure de Mines de Paris
Sophia-Antipolis
06560 VALBONNE, FRANCE

INRIA
Sophia-Antipolis
06560 VALBONNE, FRANCE

e-mail: berry@mirsa.inria.fr

Résumé

Nous étudions l'influence des langages sur la programmation des systèmes temps-réel. Nous analysons deux types de langages: les langages généraux de type ADA et les langages synchrones de types ESTEREL. Les langages généraux sont essentiellement asynchrones et non déterministes, ce qui peut conduire à des difficultés sérieuses dans les applications temps réel. Les langages synchrones sont spécifiquement adaptés au traitement d'interruptions et concilient parallélisme et déterminisme. Ils sont tout indiqués pour programmer les noyaux réactifs des applications. Les applications complexes requièrent la coopération des deux types de langages.

Abstract

We discuss real-time programming in two kinds of high-level programming languages: general purpose languages such as ADA and special purpose synchronous languages such as ESTEREL. General purpose languages are inherently asynchronous and non-deterministic; this yields severe difficulties in real-time applications. Synchronous languages specifically deal with real-time reaction to interrupts; they are better to program compact real-time applications or application parts. Complex applications require the cooperation of both kinds of languages.

Reference

Proc. IFIP 89 World Computer Congress, San Francisco, 1989. (Invited Talk.)

Real Time Programming: Special Purpose or General Purpose Languages

Gérard Berry

Centre de Mathématiques Appliquées
Ecole Nationale Supérieure de Mines de Paris
Sophia-Antipolis
06560 VALBONNE, FRANCE

INRIA
Sophia-Antipolis
06560 VALBONNE, FRANCE

e-mail: berry@mirsa.inria.fr

1 Introduction

Real-time programming is an essential industrial activity. Factories, planes, cars, and a wide variety of everyday life objects are or will be computer controlled. Real-time programs receive external interrupts or read sensors connected to the physical world and build commands as output for it. When doing so, they have to react to their inputs within externally fixed timing constraints. *Safety* is a crucial concern for most real-time programs. In this area, a single bug can have extreme consequences.

Historically, real-time applications evolved from the use of analog machines and relay circuits to the use of microprocessors and computers. The programming tools are still often low-level and specific (e.g., assembly programming or hand-coding of automata). However, the situation is changing rapidly. Low-level programming techniques will not remain acceptable for large safety-critical programs. Real-time programming will follow the modern tendency to make systems hardware independent: software has a longer lifetime than hardware.

We informally discuss which kind of high-level programming language is suited to program real-time systems. We insist on the following specific requirements: concurrency, interrupt handling, and respect of timing constraints.

Modern programming languages provide high-level data, control, and program structuring constructs. Operating systems realize basic functions such as interrupt handling, process and communication management, and resource allocation. Languages, systems, and language-system interactions are *compromises* between different

objectives: human readability and maintainability of programs, portability, and efficiency.

A common compromise is to use classical sequential languages like C or PASCAL to program sequential tasks that are run concurrently by operating systems. The sequential tasks call system services to create tasks, kill them, and communicate with each other. This trade-off relies on widely available tools. But it sacrifices clarity since concurrency and communication are somewhat hidden from the programmer. Program analysis and verification are hard.

Language designers have therefore developed a different compromise: they have introduced concurrency and communication *inside* the languages as first class concepts. The underlying operating system becomes hidden; it realizes basic interrupt handling, converts physical signals into logical programming language communications, and handles tasks as specified by the language compilers. The languages and compilers become more complex, but the programs become clearer and easier to analyze.

Most present concurrent languages are *general purpose languages*, GPLs for short. They intend to be usable in all kinds of computer applications on all kinds of hardware configurations and operating systems. At least in theory, they permit the same program to run equivalently on a single processor or on a network of processors. This is very useful for portability. However, we shall argue that there is a severe drawback: because of their generality, GPLs are somewhat *inadequate for real-time programming*.

We shall develop two arguments to support this rather strong claim, which has already been made by other authors [7,8]. First, fully portable communication must be *asynchronous* to deal with hardware distribution. This forbids proper interrupt handling, accurate time manipulation, and proper exception handling. Second, GPL concurrent programs must be *non-deterministic*, while most real-time applications should be driven by deterministic programs. Testing and verifying non-deterministic programs are at least one order of magnitude harder than testing and verifying deterministic ones.

One needs other compromises. Some authors propose to put fine-grain system primitives at the language level [7,8]. This can give much more control over program execution. Here, we shall follow others who propose to ease the programmer's task by giving him "ideal" real-time programming primitives that permit him to reason as if the device he programs had *instantaneous reactions*. This requires putting much more effort into the compilers. The special purpose *synchronous languages* ESTEREL [2,3], LUSTRE [6], SIGNAL [9], and STATECHARTS [10] take this approach. Their program units behave conceptually as people in a room when they treat problems "in real time": they instantly broadcast information to each other, instantly interrupt each other, instantly provoke exceptions.

Synchronous languages allow the programmer to focus on the logic of reactions. They make deterministic concurrent programming possible. Their mathematical semantics has been finely tuned [3,6,9,11]; it is the basis of their implementations. Synchronous languages can be very efficiently implemented, either by hardware or by

translation to finite automata that have predictable performance. In this translation, concurrency and communication are completely *compiled away*. Correctness proofs can be performed using existing automata verification systems. There are of course drawbacks to this picture. Efficient implementation is, as of yet, only possible for "finite state" programs, having a fixed pattern of process creation and communication. The object code cannot be easily distributed. Controlling the size of the resulting automata needs some skill.

We shall finally argue that a practical solution is to combine the respective skills of GPLs, synchronous languages, and operating systems, which are complementary rather than antagonistic.

2 Reactive and Real-Time Programs.

It is convenient to distinguish roughly between three kinds of computer programs. *Transformational programs* compute results from a given set of inputs; typical examples are compilers or numerical computation programs. *Interactive programs* interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive. *Reactive programs* also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not by the program itself. Interactive programs work at their own pace and mostly deal with communication, while reactive programs only work in respond to external demands and mostly deal with accurate interrupt handling.

Real-time programs are usually reactive. However, there are reactive programs that are not usually considered as being real-time, such as protocols, system drivers, or man-machine interface handlers. All reactive programs require a common programming style.

Complex applications usually require establishing cooperation between the three kinds of programs. For example, a programmer uses a man-machine interface involving menus, scrollbars or other reactive devices. The reactive interface permits him to tell the interactive operating system to start transformational computations such as program compilations.

Real-Time Constraints

In man-machine interface, fast response time constraints are merely for the user's convenience. In most real-time applications, timing constraints are more essential. In some cases (e.g., slow physical processes), programs written without special care just happen to work fast enough to be real-time. In other cases, timing constraints are hard to fulfill and require caring about all details of the executed code. In all cases, real-time constraints ask for *performance predictability*.

Concurrency

From an architectural point of view, most interactive and reactive systems decompose into concurrent communicating subunits. Programs should follow the architectural decomposition to be readable, maintainable, and built of reusable components. This requires concurrency within programming languages. An architectural concurrent component will be called an *agent*.

From an implementation point of view, programs are often implemented as sets of *tasks* communicating by operating system or hardware devices. The tasks can sit on a single processor or be distributed in networks.

The actual task implementation need not follow the architectural agent description. Run-time concurrency is not always a good way of increasing run-time performance, since task and communication handling can cause important overhead. A set of logically concurrent agents can often be advantageously implemented as an equivalent sequential task, see Section 5.

Determinism

We say that an interactive or reactive program is *deterministic* if its behavior only depends on its (timed) inputs. This is an observational point of view: the user wants to reason about a deterministic system as if it makes no internal choice; he wants the system's behavior to be fully reproducible. Most real-time programs are deterministic in this sense. On the contrary, most interactive programs are non-deterministic: an operating system can make arbitrary choices between executable processes. Determinism should be preserved whenever possible.

Termination and Abortion of Activities

There are two well-known difficulties in concurrent programming: determining when a distributed activity is completed and canceling concurrent agents. The cancellation problem is central in real-time programming. If some event requires to cancel agents that act on the outside world, these agents should usually stop doing side-effects and die as fast as possible, ideally *at once*. In real-time operating systems, killing a task should be a very fast operation. In programming languages, killing an agent must at least be allowed.

3 Human Communication

As human beings, we have a considerable experience in concurrency and communication; the notions of transformational, interactive, and reactive activities make perfect sense for us. Efficiently organizing concurrent human activities is still a hard problem. We see no definite reason for this problem to become simpler on computers. We therefore spend some time studying the communication media we use in everyday life. This will be useful for later comparisons.

Communication Media

medium	type	asyn/syn	address
mail	$1-1$	*asynchronous*	*physical*
electronic mail	$1-1$	*asynchronous*	*logical*
telephone	$1-1$	*synchronous*	*physical*
electronic talk	$1-1$	*synchronous*	*logical*
bulletin board	$1-n$	*asynchronous*	*physical*
electronic b.b.	$1-n$	*asynchronous*	*logical*
cable broadcast	$n-n$	*synchronous*	*physical*
radio broadcast	$n-n$	*synchronous*	*logical*

Figure 1: Communication Media

We use a wide variety of communication media that combine different characteristics in almost all possible ways, see Figure 1 (a synchronous medium has no delay between message emission and reception; in a physical addressing scheme, the receiver is identified by a geographical position). Other characteristics could be added in a finer analysis. For example, some devices memorize the message, some don't: a fax is a telephone with memory; a blackboard is a broadcasting device with memory.

Media Simulation

to simulate:	by:	use device:
mail	telephone	*answering machine*
mail	broadcast	*personal call*
electronic mail	mail	*mail forwarding*
electronic mail	telephone	*answering machine with remote call-in*
telephone	broadcast	*walkie-talkie*
talk	telephone	*call forwarding*
bulletin board	mail	*mass mailing*
electronic b.b.	broadcast	*broadcast news*

Figure 2: Media Simulation

We use a variety of devices to simulate one communication medium by another one, see Figure 2 for examples. Some simulations are impossible: *one cannot accurately simulate a synchronous medium by an asynchronous one*, e.g. telephone or radio by mail. However, virtually all feasible simulations are of constant use. Devices permitting to use logical addresses instead of physical ones become increasingly more important.

A compound procedure such as an acknowledge mailing requires *atomicity*: the procedure must be executed either completely or not at all. Accepting an acknowl-

edged letter automatically provokes sending the acknowledgement. This is guaranteed by the postal system.

Choosing Media

The choice of a medium depends on various parameters: the nature of the task, the urgency and importance of the message, geographical constraints, and cost constraints. Interactive activities such as ordering parts to a company can be done by mail. Reactive activities such as taking fast decisions require interrupting other people using synchronous media such as telephone or broadcasting. We use broadcasting (speech) in a room, with the major advantage that ***all the participants have the same information at the same time.***

Termination and Abortion of Activities

The way we decide that an activity is terminated and the way we operate to cancel an activity strongly depend on the media we use. There is no problem when all participants are in the same room. There are of course major problems if mail is the only available medium.

4 General Purpose Languages

We first present the basic facilities offered by all GPLs; we then study their concurrency facilities.

Basic Facilities

Data structuring tools help in organizing and naming data. Static type checking forbids mixing up data of different kinds, which usually produces nasty errors. *Control structures* help in writing understandable programs. The basic sequential control structures are now well-understood (exception handling is unfortunately not always available). *Program organization primitives* help in organizing and maintaining programs. They range from simple procedure constructs to fancy module or package structures. They are essential for "programming in the large". *Implementation* on a wide variety of target machines is of course essential for a language to be usable. The compilers must be of high quality: in real-time programming, bugs in compilers are unacceptable.

One must admit that very few actual languages are good with respect to these four aspects [13]. Choosing a GPL to treat a big application is still a problem. Unfortunately, the development of a GPL is a long and expensive task; even if knowledge improves, one cannot expect many better widely used GPLs to be available in the near future.

Concurrency in GPLs

Concurrency in most GPLs is based on three assumptions:

- Agents and source-level communications are directly mapped to run-time tasks and run-time communications.
- The behavior of a program should remain the same be it implemented on a single processor or on distributed processors, independently of the technique used to connect these processors.
- Interrupts are transformed into standard communications.

Notice that logical addressing is the rule. The connection between physical and logical addresses is done by "pragmas" that do not formally belong to the languages.

At first glance, the above assumptions seem excellent for execution control, hardware independence, and portability. But they enforce *non-determinism* and *asynchronous communications*, since communication should be feasible even for asynchronously linked tasks[1]. Referring to Section 3, this is like having mail as the unique communication medium. Interrupt handling, and therefore reactive programming, cannot be properly handled. Let us use ADA [1] as a support language for a more precise discussion. The arguments would be similar for other GPLs. Here are the ADA communication tools.

- *Shared variables*: the human analogue is a single checking account with several checkbooks held by different people. The difficulties are well-known[2].
- *Rendezvous*: If A wants to send a message to B, he goes to B's waiting room and queues for B to take him into consideration (A can possibly go back home if B ignores him for too long). When B accepts the message, A watches him read it, and only then goes back home to continue his own work. If B wants to reply to A, he follows A to his place and queues in the same way in A's waiting room.

Of course, the comparison with human communication is a bit caricatural. Since electronic processes are cheaper than human beings, the above tools can also be used as *primitives* to implement more elaborate communication structures. For instance, using rendezvous, one can easily improve upon shared memory by hiring bookkeepers, called *monitors* in programming. One can also improve upon the basic rendezvous by hiring a sufficient quantity of auxiliary messengers, e.g., buffers. But the cost can be too high for real-time applications. Deriving communication procedures also calls for atomicity, which cannot be ensured at low cost.

[1] Rendezvous is often said to be "synchronous"; message passing is indeed synchronous, but the establishment of the rendezvous is asynchronous, unlike for example in radio broadcasting.

[2] Shared variables violate the hardware-independence principle. Their inclusion in ADA is often considered unfortunate [5]. However, shared memory is of course efficient and can be safely used by some specific algorithms or by automatically generated programs.

Time and Interrupt Handling

Accurate time manipulation is of course essential in real-time programming. As many concurrent GPLs, ADA gives access to a notion of ***absolute time***. But consider the trivial problem of running a task A that signals minutes to a task B by counting seconds:

```
loop
   delay 60.0;
   B.MINUTE
end
```

where MINUTE is an entry of B. The intuitive meaning is ***not*** the actual meaning. For B to receive MINUTE, we need the conjunction of three events: A must have counted 60 seconds; B must enter a state where it actively listens to A; the rendezvous must be completed. The time taken by any of these steps is ***unpredictable***. Furthermore, MINUTE is sent by rendezvous and cannot be broadcast. If A also wants to send MINUTE to a third process C, he must call and entry C.MINUTE. The only thing that is sure is that B and C ***never*** receive MINUTE at the same time.

The same arguments apply to all interrupts. To reduce drifts with timing, some languages provide ***priorities***. Priorities are certainly compulsory for interrupt handling at the operating systems level. In our opinion, their use in a language makes the understanding of programs even more difficult.

Aborting Agents

Aborting a task in ADA can be done in principle. However, the "abort" ADA statement has a hairy semantics, which even includes the case where the task continues its execution forever. Exceptions cannot be used to abort groups of agents, since they don't propagate outside an agent. (See [5] for an extensive explanation of all the related difficulties.)

To avoid these problems, some GPLs do not allow agent abortion. Instead of being treated at the language level, even with difficulty, the real abortion problem is then pushed back to each user with no tool to treat it.

5 Synchronous Languages

Synchronous languages choose to make a big step away from direct implementation. They provide "ideal" real-time primitives on conceptually "infinitely fast" processors. For any agent, output is then synchronous with input. We shall discuss ESTEREL, which is an imperative language easy to compare with classical GPLs (LUSTRE and SIGNAL are data-flow languages, STATECHARTS is a hierarchical description of automata).

The Basic Concepts

ESTEREL has classical control statements: sequencing, conditional, loops, and exceptions (called *traps*). They transmit control in *no time*. The parallel statement

$$S_1 \;||\; S_2 \;||\; \cdots \ldots \;||\; S_n$$

instantaneously starts all its components as parallel agents. It terminates *exactly* when all the agents have terminated. The || operator can be used anywhere in statements.

As people in a room use blackboards, ESTEREL agents communicate by *instantaneous broadcasting* of, possibly valued, signals; all agents have the same exact information at the same time. Here is a way to build exact minutes:

```
every 60 SECOND do
   emit MINUTE
end
```

Physical time is a signal among others. One can use any signal to define a "time unit", as in

```
every 100 CENTIMETER do
   emit METER
end
```

To wait selectively for events, one writes:

```
await
   case 2 SECOND do ...
   case 3 METER do ...
end
```

This statement is *deterministic*: the first event indicates the action to take. If both events appear at the same time, only the first one in the list is selected.

Instantaneous agent abortion is a basic primitive. Here is an exact speed measure statement that emits the speed every meter:

```
loop
   var TIME := 0. : float in
      do
         every MILLISECOND do
            TIME := TIME+1.
         end
      watching METER;
      emit SPEED (1000./TIME)
   end var
end loop
```

When it receives the control, the "do ... watching" watchdog construct instantaneously starts its body as an agent. It instantaneously aborts the agent when the mentioned event occurs. Any other agent can read the speed at any time by evaluating the expression ?SPEED.

Exception (trap) handling can also be used to abort agents. For example, in

```
trap END in
    S1 || S2 || ··· || Sn
end
```

any of the S_i can execute an "exit END" statement at any time; then all the S_i are instantaneously killed and the whole trap construct is terminated. The killed agents can perform instantaneous "last will" operations as advocated for in [7].

We shall not go further into the description of ESTEREL, see [3,2] for more details. We just want to convince the reader that synchronous languages provide exact temporal manipulation and make concurrent deterministic programming possible.

Mathematical Semantics and Implementation

ESTEREL is defined by a mathematical semantics that determines a unique reaction to each input [3]. This only holds in the absence of *race conditions*, which closely correspond to race conditions in circuits and to deadlocks in asynchronous languages; all races are detected by the semantics.

The mathematical semantics is directly implemented in a simulator, which has good but not real-time performance. Compiling consists in simulating all possible behaviors of the program. The compiler transforms a concurrent program into a sequential automaton[3] that acts on a shared memory. The only statements that remain on automaton transitions are the indispensable run-time data-handling ones.

When actually run, the object code is of course not synchronous. However, assuming synchrony for scheduling and agent synchronization makes perfect sense: these operations generate *no run-time code*! The object code is highly efficient. Its reaction time is *predictable* since all transitions are sequential. Automata verification systems can perform various kinds of correctness proofs (obeying the WYPIWYE principle, "What You Prove Is What You Execute" — the proofs are done on the actual object code, not on a more or less accurate model).

For pure synchronization programs that manipulate no data, one can also inplement an ESTEREL program on a circuit in such a way that a reaction takes exactly one clock cycle. This is of course the best possible approximation to pure synchrony.

To summarize, even if it is not fully reachable in practice, synchrony is a good paradigm: it permits better programming, better code generation, and better program verification.

[3]Translation to automata is presently implemented for ESTEREL and LUSTRE; it could also be used for other synchronous languages.

6 Making SPL and GPL Cooperate

Synchronous languages don't solve all problems. They cannot yet deal with arbitrary distribution nor with recursive process creation. We think that a good solution is to use synchronous and asynchronous languages as complementary tools, just as we use broadcast and mail. Synchronous languages are better to write the well-identified reactive parts of an application. These reactive parts can be asynchronously connected using GPLs and operating systems. Our solution can be called "Communicating Reactive Processes".

In practice, the ESTEREL (and LUSTRE) compilers can produce code in various GPLs. Code production relies on the basic GPL facilities presented in Section 4. ESTEREL has no built-in data structuring facility. Except for the very basic ones, types and operations are abstractly known by name; they are implemented in the target GPL. This simplifies the compiler and makes programs fully portable from one target GPL to another.

At run-time, the compiled automata act as *services* to be called by GPL programs using a simple procedural run-time interface. The only restriction is that each automaton call must be guaranteed to be atomic. Synchrony can indeed be viewed as an elaborate form of atomicity.

7 Conclusion

Our compromise is applicable now to existing problems. It has already given excellent practical results. More fundamental solutions would require a better understanding of the relation between synchrony and asynchrony. Process calculi and theories of atomicity seem a good mathematical tool for such a study [4,12].

References

[1] *The Programming Language ADA Reference Manual.* ANSI / MIL-STD-1815A, also Lecture Notes in Computer Science 155, Springer Verlag, 1983.

[2] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems: an introduction to Esterel. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 35–55, Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[3] G. Berry and G. Gonthier. *The Esterel Synchronous Programming Language: Design, Semantics, Implementation.* Research Report 842, INRIA, 1988. To appear in Science of Computer Programming.

[4] G. Boudol and I. Castellani. Concurrency and atomicity. *Theoretical Computer Science*, 59:25–84, 1988.

[5] Alan Burns. *Concurrent Programming in Ada.* Cambridge University Press, 1985.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre, a declarative language for real-time programming. In *Proceedings ACM Conference on Principles of Programming Languages*, 1987.

[7] Marc D. Donner and David H. Jameson. Language and operating system features for real-time programming. *Computing Systems*, 1:33–62, 1988.

[8] Stuart R. Faulk and David L. Parnas. On synchronization in hard-real-time systems. *Communication of the ACM*, 31(3), 1988.

[9] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gauthier. SIGNAL: a data-flow oriented language for signal processing. In *IEEE Transactions on ASSP, ASSP-34 No 2*, pages 362–374, 1986.

[10] D. Harel. Statecharts : a visual approach to complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[11] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proc. Symposium on Logic in Computer Science*, pages 54–64, 1987.

[12] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[13] Stephen J. Young. *Real Time Languages.* Ellis Horwood Limited, 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique