On-the-fly replay : a practical paradigm and its implementation for distributed debugging

O. Gerstel, Michel Hurfin, Noël Plouzeau, Michel Raynal, S. Zaks

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

On-the-fly replay: a practical paradigm and its implementation for distributed debugging

O. Gerstel, M. Hurfin, N. Plouzeau,

M. Raynal and S. Zaks

N° 1984

Août 1993

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

On-the-fly replay: a practical paradigm and its implementation for distributed debugging

O. Gerstel*, M. Hurfin**, N. Plouzeau**,
M. Raynal** and S. Zaks*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n ° 1984 — Août 1993 — 15 pages

Abstract: This paper presents a practical paradigm, called *on-the-fly replay*. This paradigm consists of running a distributed program twice at the same time: an original computation driving a twin execution whose non-deterministic choices have not to be evaluated. This paradigm has several interesting uses. Among them, distributed debugging is particularly noteworthy. The integration of this paradigm into a distributed debugging facility, called EREBUS, is described. This implementaton was run on a distributed memory parallel machine (Intel Hypercube iPSC2) and experimental results, showing gains provided, are exhibited.

Key-words: Distributed debugging, Execution replay, Probe effect, Measurements

(Résumé : tsvp)

This work has been partly supported by a cooperation grant from French and Israeli goverments.

*Computer Science Department - Technion Haifa 32000 ISRAEL
**IRISA - Email address: <name>@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Réexécution au vol : un paradigme pratique et son implémentation pour déverminer des programmes répartis

Résumé : Ce papier présente un paradigme pratique appelé *réexécution au vol*. Ce paradigme consiste à exécuter au même instant deux fois le même programme: un calcul initial pilote une exécution jumelle où les choix non déterministes n'ont plus à être effectués. Ce paradigme a plusieures utilisations intéressantes. Parmis celles-ci, il convient de citer le déverminage des programmes répartis. L'intégration de ce paradigme dans un outil de déverminage de programmes répartis appelé EREBUS est décrite. Cette mise en oeuvre a été réalisée sur une machine parallèle à mémoire distribuée (L'Hypercube iPSC2 de Intel) et des résultats expérimentaux montrant les gains obtenus sont présentés.

Mots-clé : Déverminage des programmes répartis, réexécution de programmes, effet de sonde, mesures

1 Introduction

Debugging asynchronous distributed programs is notoriously a difficult and error-prone task [10], mainly because of two reasons. First, distributed programs execute non-deterministically, since they may contain non-deterministic statements and also since inter-processor communication delays are unpredictable; hence, given an input data set, a distributed program may behave differently when executed several times, and even may produce different results, impairing the debug task of the developer. Consequently, some debuggers for distributed programs are based on the replay technique [9]: during one execution, useful data is logged, and then used during next executions to guide them and enforce a deterministic behavior. Such replayed executions are immune to probe effects and are then perfect patients for detailed analysis.

The second reason is the fact that recording information in order to replay the initial execution induces probe effects on it (both in CPU consumption and in storage space required to store the data needed for deterministic reexecutions). Probe effects are almost unavoidable without extra hardware dedicated to monitoring tasks. In this paper we give the flavor of a general technique, named *on-the-fly replay*, which uses a regular CPU as ancillary hardware to monitor main computations while avoiding probe effects as much as possible. This is an alternative scheme to the superimposition paradigm[1], implicitly used by current logging techniques and basically used to detect stable property detections, such as termination or deadlocks.

This paper studies the use of the *on-the-fly replay* paradigm in the debugging of distributed programs. It is divided into four main parts. Section 2 introduces the computational model for distributed programs. Section 3 presents the basics of the *on-the-fly replay* paradigm. Section 4 presents a debugging facility, called EREBUS, and Section 5 describes the integration of the *on-the-fly replay* into EREBUS; this Section displays also results that exhibit benefits of the *on-the-fly replay* paradigm in limiting probe effects when analyzing distributed executions.

2 The computational model

The programs we analyze and debug are composed of a finite set of processes that communicate only by messages exchanged along unidirectional channels. These channels are assumed to be reliable (no loss and no spurious messages); they are not necessarily FIFO and may exhibit unpredictable transmission delays (communication is asynchronous). Processes may contain non-deterministic statements (e.g., com-

binations of *select* and *accept* constructs of ADA); they may use random number generators and read wall clocks. An execution of each process is modeled by a sequence of events and the whole program execution is modeled by a partial order relation on these events [8].

The computational model is the classical asynchronous model of distributed computations. For brevity's sake, and without any loss of generality, we assume in this paper that distributed programs are executed on machines with one processor per process and one physical link per logical channel.

3 The *on-the-fly replay* paradigm

The *on-the-fly replay* paradigm basically generates two identical executions (of the same program) at the same time. The second one is deterministically driven by the first one. This second execution can be used to take snapshots, evaluate properties on the computation, do measurements, etc, while leaving the first execution undisturbed.

Structural part. Given a network C, we construct its twin network $I(C)$ as follows. For each process p, a twin process $I(p)$ is added and a unidirectional communication channel is established from p to $I(p)$; process $I(p)$ is loaded with p's code. Moreover, if processes p and q are interconnected by a communication channel then processes $I(p)$ and $I(q)$ are also connected by a communication channel. The structure so designed is shown on Figure 1.

Behavioral part. During an execution, each process $I(p)$ imitates the behavior of its twin process p. In order to faithfully do the imitation, each process p informs its twin $I(p)$ of events occurring in p. Of course, only events related to non-deterministic choices have to be signaled in this way, since p and $I(p)$ execute the same code. As the communication between p and $I(p)$ is performed asynchronously by message-passing, the $I(C)$ network follows the behavior of the C network with some delay. We call this technique *on-the-fly replay*.

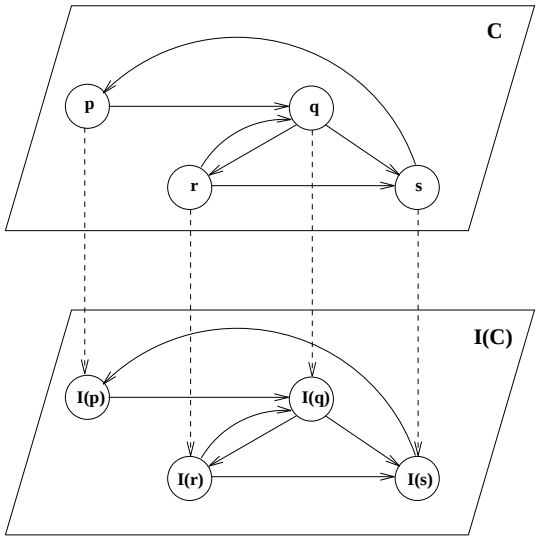

Figure 1: *On-the-fly replay* network

Advantages of this scheme.

- **Minimizing the probe effect:** The *on-the-fly replay* scheme was designed to minimize probe effects, more precisely to reduce the CPU load increase due to monitoring activities and also to reduce network side effects induced by supplementary control messages.
- **Increase in computational power:** Experiments show that processes spend a substantial part of their time evaluating conditions of progression (guards). As $I(p)$ is fully driven by p, $I(p)$ does not spend time evaluating guards and thus saves CPU time. This time can be used by $I(C)$ for event monitoring and logging tasks. Experimental results demonstrate the usefulness of such an approach.
- **Increase in communication ability:** A significant reduction in the load on the $I(C)$ communication network can be achieved by adding the contents of arriving messages to the data that p reports to $I(p)$. By this, the $I(C)$ network is almost free of messages, and can be used for heavy load distributed control algorithms (e.g. snapshot).

Related schemes. Other schemes for running control algorithms in parallel have been proposed, for instance the superimposition scheme [1], whose main application field is stable properties detection, such as deadlock or distributed termination. While such a technique is implemented by interleaving main computation and observation activities, the *on-the-fly replay* technique ensures a truly parallel execution of these activities, in order to lower probe effects. This is particularly important when dealing with observation and detection of unstable properties [3, 2, 5].

4 The EREBUS distributed debugging facility

The EREBUS debugger (described in more details in [4]) controls execution of distributed programs written in a subset of the ISO Estelle language [6].

4.1 The Estelle language

Estelle is a cross-breed between Pascal programming language and communicating automata. An Estelle program is made of several processes which may exchange messages. Those messages travel along FIFO channels which ensure transmissions without loss, duplication or alteration. Each process has his own set of Pascal local variables and his own set of communication ports where messages received but not yet consumed are queued. The behavior of an Estelle process is given by an automaton, described as a set of transitions from one control state to another one. Each transition is composed of a guard expression and an action block which is a sequence of Pascal and Estelle statements. The guard expression states a condition using values of local variables or the type and field values of the first message of some port queue. An evaluation of this condition must be done in order to determine whether the transition is firable in the current state of the automaton. Processes evolve asynchronously, each of them repeating continually the three following steps. First a process evaluates all the guards of its transitions. Then the process selects one of the transitions which has a guard evaluating to *true*. Then it executes actions associated with the selected transition. If its guard includes a message reception statement from some port then the first message in this port queue is consumed.

4.2 EREBUS

The EREBUS project is aimed at providing a set of tools to compile, run, analyze and debug Estelle programs on different distributed-memory parallel machines [4].

Executing an Estelle program

The language accepted by the compiler in the EREBUS environment is a subset of Estelle, differing by the absence of dynamic reconfiguration facility (*i.e.* the set of processes is static, as is the set of interprocess communication channels) and by the fact that using shared variables is forbidden. Moreover, the programmer gives informations about the mapping of processes on physical sites. The user of the EREBUS system is able to submit an Estelle program for execution on different computers with centralized or distributed architecture (for example a raw Sun workstation or the Intel iPSC hypercube) and the program is compiled automatically into machine language for the desired host computer. The compiler translates each transition of each Estelle process into a function G, which implements the guard expression, and a procedure A, which implements associated actions. During execution of the program, schedulers (one for each processor) execute the following algorithm :

1. Take into account incoming messages and put them in the queue corresponding to the entry port.
2. For each process P located on this physical site, evaluate every function G_1^P,...,G_n^P (assuming that process P has n transitions).
3. For each process P, execute one procedure from the set $\{A_k^P \mid G_k^P$ evaluates to *true* $\}$

This algorithm is repeated forever; from the language point of view, processes never terminate.

Non-determinism of Estelle program

As previously announced, a major point in the EREBUS project is design and implementation of tools which could help the programmer to analyze behaviors of distributed programs during execution. In order to design a coherent group of tools, every tool must be defined with regards to a single model of distributed computation. An execution of an Estelle program can be characterized by a partially ordered set of local states. The local states of an Estelle process are the local states it attains after each execution of an action block. The order relation between states is similar to the "happened before" relation defined by Lamport upon events [8]. Such an order is described in more details in [5].

Estelle programs are inherently non-deterministic because the fired transition is chosen at random among all fireable transitions. Furthermore, when a program is

executed on a distributed memory machine, the number of transitions detected as fireable during evaluation of guards can depend on the underlying hardware. Let us consider a transition whose guard contains a clause *when*. This clause specifies that the transition is not fireable if a message is not ready to be consumed. Because of the non-determinism of the transmission delay, this condition can be either false or true depending on the moment at which the evaluation of the guard occurs. So, given the same input, a program may show different but nevertheless correct behaviors. With the model used, a formal and unambiguous definition of equivalence between executions of a same program can be expressed: two executions are equivalent if they produce the same partially ordered set of local states.

As mentioned in the previous section, analysis of a behavior cannot always be done during a single execution. Also, the design of a replay mechanism is mandatory in the development of analyzing tools. Due to a replay mechanism, a programmer can reproduce at will executions equivalent to an initial execution (characterized by an interesting behavior one wants to study).

4.3 The basic replay mechanism EREBUS v1

Logging of informations

During the initial execution, information about the partially ordered set of local states is saved. This information, recorded in a file, is then used to force subsequent executions of the same program (called replays) to be equivalent to the initial execution.

The reexecution facility relies on some important properties of EREBUS programs, namely the followings facts:

- there is at most one message reception for each transition,
- a reception specification of a transition indicates one sender only,
- the communication is FIFO and point to point,
- the topology of the interprocess channel network is static.

Due to these features, logging identities of the transitions fired by a process during the initial execution is sufficient to replay the same sequence of transitions. An Estelle program is a specification of a closed system (without outside interactions). Yet, in order for a process to obtain the same sequence of local states, each value returned by internal non-deterministic sources of value such as wall clocks and random generators are also logged during the initial execution.

Assuming that every transition of a process has a unique identity number within the scope of this process, the amount of data one has to log during the initial execution is very small. Each time a scheduler fires a transition, it appends the transition identity to the log file of the corresponding process. Every call to a wall clock or random number service is trapped and the value given to process is logged in the same way.

The probe effect

If information gathered by a scheduler is stored in the local memory of the processor executing this scheduler, the probe effects induced by the gathering could seem limited; but this is not the case for two reasons. First, in order to detect the end of the computation, a termination detection algorithm must be executed by the set of schedulers. Second, copying information on disk must be done more or less frequently depending on both local memory space and amount of information saved by a scheduler. Dumping a local log buffer into disk storage on the machine front-end is a time-consuming operation which furthermore increases the network load. To limit such a copy out (which invariably perturbs the computation), a data compression algorithm is used. Moreover, in order to execute the replay on any computer, the data saved is encoded in a machine independent way before compression. The number of extra statements executed to transform the representation of a value is variable but always small. The extra activity is also not only concentrated in a few places but spread out all over the computation. Because of this spreading, the probe effect is reduced because executing a sequence of only a few statements does not always introduce perturbations in the computation. For example, if a scheduler saved the identity of a transition after its execution, this additional activity has no impact on the computation providing that all the processes managed by this scheduler are passive at this time (i.e. they are waiting for a message to proceed with their execution). As we had to balance the costs with the benefits, we chose a fast and simple compression algorithm [7].

Experimental results

Three examples were used to experiment with the compression algorithm: a simple token ring protocol, a pipelined Erathostenes sieve and a model of the X25 protocol. Experimental results on space saved by compressing the log buffers are displayed in Figure 2.

Estelle specification	Maximum number of defined transitions per process	Number of transitions fired (total)	Number of bits needed to log a transition execution.(after compression)	Space saved by compression (on the average)
TOKEN RING	2	4000	1	0 %
PRIME NUMBER	4	500	1.71	14.3 %
X25 PROTOCOL	155	488219	4.58	42.7 %

Figure 2: Space saved by log buffer compression

Execution speed impacts on the replay logging mechanism were also studied. Figure 3 shows the relative speed of execution of an Erathostenes sieve when the logging system is off (Plot #1), when the logging system is on (Plot #2) and when the program is replayed from log data (Plot #3). Two features of these plots are noteworthy. First, the extra cost of logging rapidly decreases as the number of processors increases (as in that case the number of schedulers grows up while the number of processes per scheduler decreases). Second, a replay runs faster than the original execution: during the reproduced execution, transition guard evaluations are reduced to the minimum. Only the guard of the transition which must be fired (as indicated by the log) is evaluated.

Remaining problems

During the initial execution, the slightest extra activity can perturb the computation. Therefore, nothing can be done during this first execution in order to start or even prepare the analysis of the computation. So as no intermediate global states are computed and stored during the initial execution, setting the program into some particular state during the first replay requires a lot of time: although the user may be interested in the last few seconds of an execution, everything has to be reexecuted, from the known initial state up to the point of interest. The simple solution which

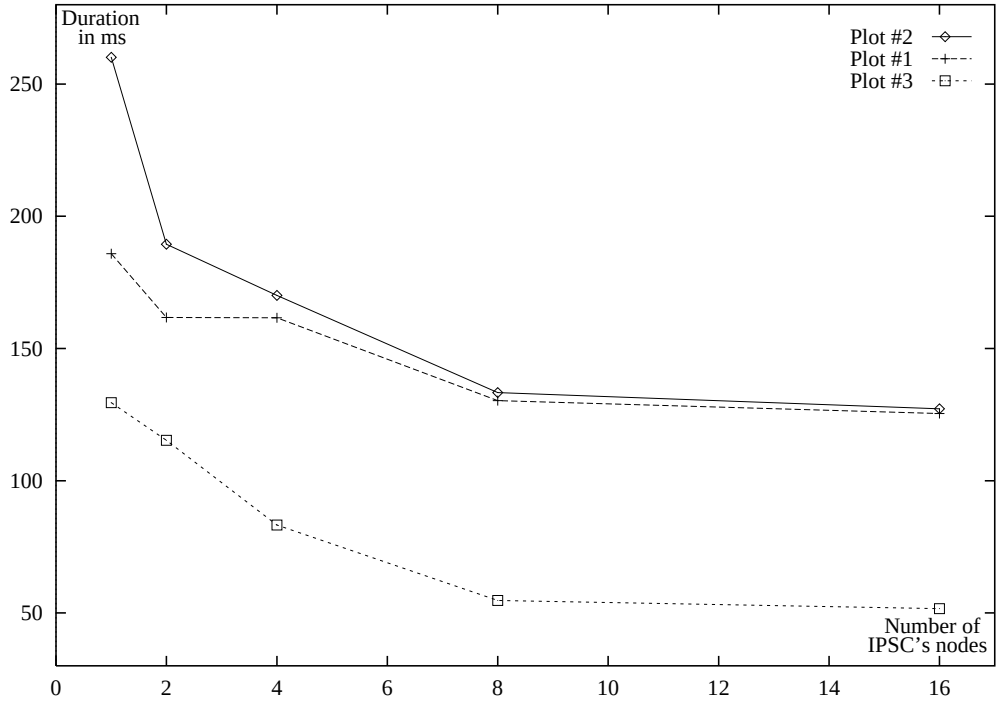

Figure 3: Efficiency measurements

consists in using checkpoints as shortcuts to the goal state can't be used during the very first replay, since no checkpointing can be done during the first execution without heavy side effects.

5 Integrating *on-the-fly replay* into EREBUS

5.1 Integration of the *on-the-fly* paradigm

A second replay mechanism has been designed to decrease the probe effects during the initial execution. This new system uses the *on-the-fly replay* presented in Section 3. It is even possible to periodically checkpoint the initial computation without sides

effects, hence enabling fast reexecutions (by going directly to an interesting global state that has been checkpointed during the *on-the-fly replay*).

Changes of representation performed for saved data and the termination detection are no more executed by the schedulers but by its twins. The extra activity of a scheduler is limited to the transmission of values to twin processes. Different solutions can be adopted. A message can be sent to a twin each time a new value must be saved. Or values can be logged into a buffer whose contents will be transferred from time to time. In all cases, the part of the network used to execute applications is not saturated by these control messages.

Twin processes are responsible for doing a part of the work realized by the scheduler in the previous version of the replay mechanism. They must detect the termination of the application and log in a machine independent way values needed for next replays. Here use of a compression algorithm loses its interest as durations of the executions are no longer dependent on the time spent by a twin process to save data into a file.

During the first replay performed by twins, the state of each process can be occasionally saved as a checkpoint. The problem described in the previous section is also resolved. Analyzing activities such as detection of predicates [5] can be executed by twins. This possibility is very useful when one tries to debug a program. The user who suspects an error in his program, can specify a propertie which characterized a failure [5]. The user executes the program until a failure is detected by twin processes during an execution. As information about last execution is saved, different replays can be executed in order to detect associated faults and to correct corresponding errors.

5.2 Performance measures

Experiments have been done to study the impact of the new *on-the-fly* technique. In order to ensure that execution times remain constant for a given set of experimental parameters values, a deterministic application has been chosen: application processes participate in a mutual exclusion algorithm, order for requests being fixed to ensure a deterministic behavior. Each application process runs on a distinct processor, with a total of eight processes. As each scheduler manages only one process, the first replay mechanism is tested in a favourable case. Three different configurations have been defined:

1. mutual exclusion application alone (no replay software involved)
   (Plot #1)

2. mutual exclusion application with the standard log technique (Plot #2)
3. mutual exclusion application with *on-the-fly* log technique (Plot #3)

For each configuration different executions have been performed with different transitions set sizes. The transitions set size of a process is easily modified by duplicating transition declarations in its source text. The only effect of such duplications is increasing the overhead of transition guards evaluation; as the aim of *on-the-fly* technique is reducing this overhead, changing the transition set size of processes shows how the overhead may depend on the process size.

Figure 4 shows that the scheduler overhead is constant with respect to the transition set size when the *on-the-fly* technique is used (this overhead is given by the difference between Plot #3 and Plot #1). On the contrary, the standard technique overhead increases with the transition set size. Finally the bigger the number of guards to be evaluated the better the *on-the-fly replay* (as show by the difference between Plot #3 and Plot #2 that grows with the number of transitions).

6 Conclusion

This paper has introduced a new practical paradigm, called *on-the-fly replay*, and its use to lower probe effects in distributed debugging. Experimental results shows the interest of such an approach: not only probe effects are made minimal but additionally logging of informations and checkpoint computations can be done by a twin execution at the same time of the first execution. The distributed debugging facility using this paradigm is currently working.

Other researchs on the use of this paradigm are under investigation, especially for the computation of snapshots and detection of stable properties such as termination and deadlock. This paradigm seems to be a viable practical alternative to the superimposition one [1].

References

[1] K. M. Chandy and J. Misra. *Parallel program design : a foundation.* Addison-Wesley, 1988. 516 p.

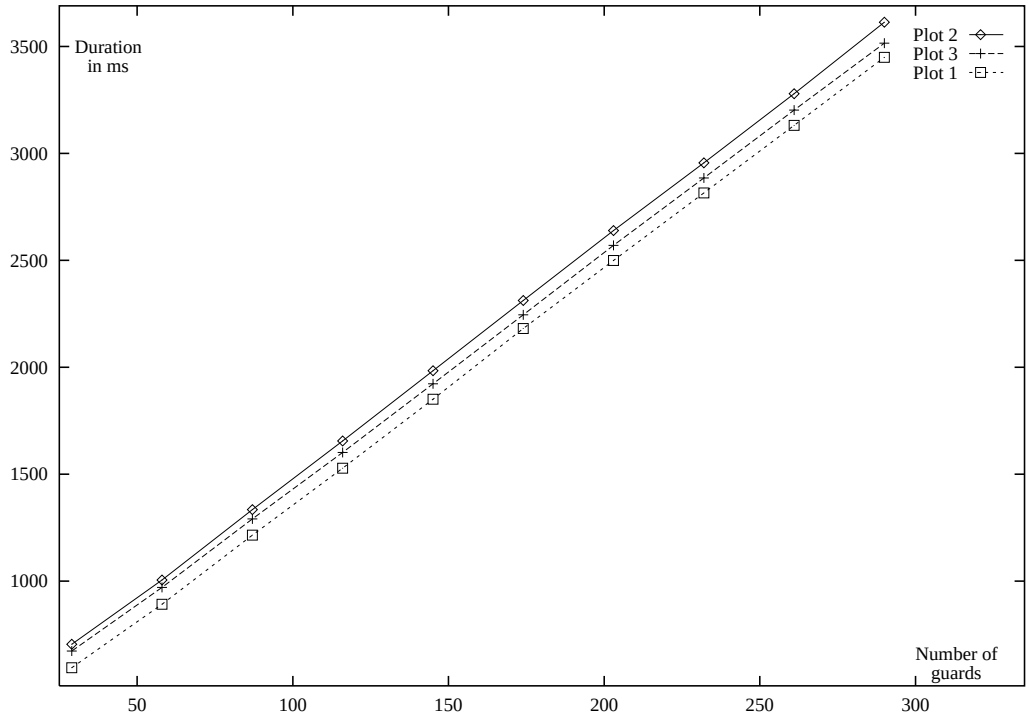

Figure 4: Efficiency measurements

[2] R. Cooper and K. Marzullo. *Consistent detection of global predicates.* In Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pages 163–173, Santa Cruz, California, May 1991.

[3] Vijay K. Garg and Brian Waldecker. *Detection of Unstable Predicates in Distributed Programs.* In *Proc. of the 12*th *conf. Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, Lecture Notes in Computer Science 652, Springer-Verlag, New Delhi, India, December 1992.

[4] M. Hurfin, N. Plouzeau, and M. Raynal. *A debugging tool for Estelle distributed programs.* Journal of Computer Communications, May 1993.

[5] M. Hurfin, N. Plouzeau, and M. Raynal. *Detecting Atomic Sequences of Predicates in Distributed Computations.* In Proc. ACM Conference on Parallel and

Distributed Debugging, San Diego, May 1993.

[6] ISO 9074. *Estelle: a Formal Description Technique based on an Extended State Transition Model.* ISO TC97/SC21/WG6.1, 1989.

[7] D. Jones. *Application of splay trees to data compression.* Communications of the ACM, 31(8):996, aug 1988.

[8] L. Lamport. *Time, clocks and the ordering of events in a distributed system.* Communications of the ACM, 21(7):558–565, July 1978.

[9] T. Leblanc and J. Mellor-Crummey. *Debugging parallel programs with instant replay.* IEEE Transactions on Computers, C-36(4):471–482, April 1987.

[10] C.E. McDowell and D.P. Helmbold. *Debugging concurrent programs.* ACM Computing survey, 21(4):593–622, 1989.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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