EREBUS: a debugger for asynchronous distributed computing systems

Michel Hurfin, Noël Plouzeau, Michel Raynal

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

1992

25ème anniversaire

N° 1613

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

EREBUS

A DEBUGGER FOR ASYNCHRONOUS DISTRIBUTED COMPUTING SYSTEMS

Michel HURFIN
Noël PLOUZEAU
Michel RAYNAL

Février 1992

RR-1613

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CéDEX
FRANCE
Téléphone : 99 84 71 00
Télex : UNIRISA 950 473F
Télécopie: 99 38 38 32

EREBUS

A Debugger for Asynchronous Distributed Computing Systems

EREBUS : Un Metteur au Point pour les Programmes Répartis Asynchrones

Michel Hurfin, Noël Plouzeau, Michel Raynal

Campus de Beaulieu -- 35042 RENNES CEDEX -- FRANCE

<Name>@irisa.fr

Programme 1, Projet ADP (Algorithmes Distribués et Protocoles)

Publication Interne n°630 - Janvier 1992 - 14 pages.

RESUME

Cet article aborde les problèmes liés au déverminage des programmes répartis s'exécutant sur des machines parallèles à mémoire distribuée où les processus ne communiquent que par envoi de messages. Les principales difficultés liées au déverminage de tels programmes sont exposées. Les principes pour concevoir et implémenter un dévermineur pour des programmes spécifiés dans le language Estelle sont présentés. Toutes les solutions décrites ont été implémentées dans un dévermineur distribué appelé EREBUS.

ABSTRACT

This paper addresses the problem of debugging distributed programs executing on distributed memory parallel computers with message-passing interprocess communication. The main issues of debugging such programs are exposed. Principles for designing and implementing a debugger for programs specified in the Estelle language are presented. All the described solutions have been implemented in a distributed debugger called EREBUS.

EREBUS
A debugger for asynchronous distributed computing systems*†

Michel HURFIN Noël PLOUZEAU
Michel RAYNAL

Team ADP - IRISA
Campus de Beaulieu - 35042 RENNES CEDEX - FRANCE
<*Name*>@irisa.fr

Abstract

This paper addresses the problem of debugging distributed programs executing on distributed memory parallel computers with message-passing interprocess communication. The main issues of debugging such programs are exposed. Principles for designing and implementing a debugger for programs specified in the Estelle language are presented. All the described solutions have been implemented in a distributed debugger called EREBUS.

1 Introduction

Designing environments and tools for building distributed applications is one of the major current research challenges in distributed systems. At the present time, effective tools able to help the programmers in distributed software development are uncommon. The lack of efficient tools is painful especially as programming applications for distributed memory parallel computers is difficult.

An efficient debugger must take into account problems peculiar to distributed systems. Also many aspects of distributed debugging will differ from sequential debugging ones.

*This work has been partly funded by the french national project C^3 on concurrency and distribution.

†This report is an article presented at the 3rd IEEE Worshop on Future Trends in Distributed Computing Systems - Taipei, Taiwan, 13-16 April 1992.

Even the circumstances in which a debugger is employed are not similar. Generally, a programmer has recourse to a sequential debugger in order to locate and correct an error. The use of a distributed debugger could be more widespread. All the observing facilities offered by a distributed debugger are useful to analyze the behavior of a distributed algorithm, even if this one is already correct in regard to the specifications. Having an accurate view of an algorithm's behavior is very useful in order to improve its performances. In connection with this, tools for measuring quantitative aspects (number of exchanged messages) and qualitative aspects of the computation (parallelism and concurrency) can be usefully associated to debuggers [14].

The current paper is divided in two parts. In the first part, we expose some problems we are faced to when designing a debugger for asynchronous distributed computing systems. In a second part, the design and the implementation of a debugger for programs specified in the ISO-normalized Estelle language are presented.

2 Debugging a distributed program

2.1 Granularity

First of all, the designer of a debugger must define a level of observation and control. This decision influences all other choices. By defining a granularity, the designer of a debugger intends to identify instants of the execution where facts can be observed. As a result, a distinction has to be made between observable actions and undetectable ones. Thanks to a fine grain (machine instructions) few phenomenons escape to the observation. Yet the huge amount of information overwhelms the observer. In addition, the debugging activity depends also on the hardware. Debugging at the level language by choosing a coarse grain of observation and control resolves these two problems. But then many events become unnoticed and some aspects of the computation cannot be discussed by the debugger. So, if one tries to debug at the level language a correct program compiled by an erroneous compiler, the events which generate a failure will probably be invisible. Defining a granularity is tantamount to the selection of a subset of detectable errors. In order to take into account the synchronization among processes, a coarse grain is sufficient and well suited.

2.2 Design of a model

All the facilities offered to the programmer must be cohesive. To reach this goal, a model of distributed computation is needed. This model must be consistent in regard to the chosen granularity. Only observable events must be mentioned in this abstract representation of the computation. All the developed tools have to comply to this model and are relevant as long as the distributed program satisfies the properties given in the model. Of course the model must take into account the relative time relationships between events. The conventional representation of the behavior of a distributed application by the means of three kinds of

action (sending events, receiving events, internal events) can be adopted on condition that all the occurrences of this actions are observable [11].

2.3 Reproducibility

Distributed program, when executed on distributed memory machines with message-passing interprocess communication, can exhibit non-deterministic behaviors because of the non-determinism of the message transmission delays and also because some programming languages offer non-deterministic constructs, e.g. CSP, Ada and Estelle. Given the same input, the program may show different but nevertheless correct behaviors. In order to detect an error, several successive executions of the same program are generally necessary. For obvious reasons, each execution must be similar [12, 13]. As it is impossible to reproduce a computation identically in all details, an equivalence between two behaviors of a same program must be defined. A formal and unambiguous definition of equivalence between executions has to be found in regard to the model.

2.4 Consistent global states

An effective debugger has to provide facilities for monitoring the evolution of every single process as well as the relations between processes, *i.e.* the evolution of processes states as a whole. As mentioned above, the state of a single process (a *local* state) is defined at some particular moments. The global state of a program being executed is made of the states of all processes and of all communication channels, whose state is the list of in transit messages. As every process evolves asynchronously with respect to the other processes, and as computing a snapshot of the global state is not an instantaneous operation, only some snapshots are consistent with respect to message transmission causality [4]. More precisely, presenting to the debugger user a snapshot where some message has been received but not sent should be forbidden, because the causality rule stating that a message reception occurs after the emission is not respected in this case.

During the debugging activity, halting or restoring the computation back into a consistent global state is a common request. Most of the time, the exact halting state is not defined by the request. One state must be chosen among all the possible consistent global states which respect the constraints imposed by the user's request. Instead of choosing at random a global state, it is more interesting to define particular states in regard to the user's request. The definition of minimal and maximal global state is a possible solution to this requirements. The concept of minimal state appeared in [5]. A formal description of these states can be found in [1]. If a user selects a global state by choosing the local state of some process, the relevant local states for the other processes are determined thanks to those formal definitions. In the case of the minimal state, all the processes are restored in a state such that the corresponding global state is the earliest consistent one. The definition of maximal state is dual to the minimal's one. If the halting condition identifies an error, the user will discover the error's origins thanks to the minimal state. The maximal state will be useful in order to

understand how this error could produce forthcoming faults.

3 Implementation of a debugger

Introducing the language The Erebus debugger controls the execution of distributed programs described with a subset of the ISO Estelle language [3], named the Echidna language [7]. In a few words, Estelle is a cross-breed between Pascal and communicating automata: a program is made of processes which use message passing communications. Messages travel along FIFO channels which ensure transmission without loss, duplication or alteration. Each Echidna process is an automaton, described as a set of transitions from one control state to another one. Each process also owns a set of Pascal local variables and a set of communication ports where messages received but not consumed are queued. Each transition has a guard expression, stating a condition upon the local variables' values or upon the type and field values of the first message of some port queue, and an action block formed by a Pascal instruction block. Processes evolve asynchronously by repeating the following actions: evaluate all transitions' guards, selecting one of the transitions which have a guard evaluating to *true* and then executing the Pascal action block of the selected transition. If its guard includes a message reception statement from some port then the first message in this port queue is consumed.

Echidna programs differ from Estelle ones 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.

Although this presentation is very short, most of the main features of the language are given. The semantics of an Echidna program is indeed simple and clear, thanks to the well-known communicating automata model; for instance, the state of a stopped program is a well-defined notion: the state is the cross-product of every process' state (control state, values of local variables and input queues) and of the sequence of messages under transmission on every communication channel.

Executing an Echidna program The user of the Erebus system is able to submit an Echidna program for execution on different computers with centralized or distributed architectures. Although the program is compiled into machine language for the desired host computer, the user never sees the machine language version. Hence, choosing an architecture for execution is a matter of speed and cost, because an Echidna program describes a parallel system whose meaning remains the same, whatever the computer executing it. Of course parallel machines are well suited to fast execution of these programs, but a common workstation is sufficient. This enables debugging strategies where a program is executed on a fast, distributed machine until some event occurs (breakpoint detection, assertion failure) and then the debugging task is performed on a personal workstation, freeing the parallel machine for use by other people. After some debugging is done, the user may go on with the fast execution mode on the parallel or distributed computer. Switching between low and high

speed execution modes can be achieved because the user operates at the source language level.

3.1 From source file to program execution

A user executes and debugs an Erebus source program by invoking a compiler from her user interface; the compiling and linking operations are then performed by the Echidna system [7]. The Echidna program is compiled into a C file, which is then compiled by a standard C compiler into an object file. This file is independent from the target architecture, but the final executable file is not, as it is made by linking the previous object file with a runtime defined and tuned for the target architecture. The runtime acts as a layer hiding peculiarities of the underlying operating system and hardware; namely, it performs two important tasks:

- it schedules the processes on one processor and fires transitions of these processes,
- it provides the processes with an interprocess communication facility complying with Estelle's semantics.

Distributed execution of an Erebus program In the following paragraphs we focus on the issues related to the execution of an Erebus program on a distributed memory parallel computer (DMPC for short). The first phase of execution starts by loading the same program on every processor of the DMPC; in other words, every processor's local memory contains a copy the code of all processes and a copy of the runtime. Every processor starts executing the runtime. Erebus processes which are going to execute on processor i are locally created by the runtime executing on i; the runtime also creates communication channels between each process on i and processes on the same or on other processors. Then every runtime starts scheduling transitions for the processes on its processor.

Transition scheduling The concept of transition and transition scheduling is an important one, regarding Estelle and Erebus program semantics. The Echidna compiler translates each transition of each Echidna process into a function G, which implements the guard expression, and a procedure A, which implements the action. During the execution of the program, the schedulers (one for each processor) execute the following algorithm for every process P:

1. Check for incoming messages, so that the queues which contain the message received but not consumed are up to date.
2. Evaluate every function $G_1, \ldots, G_n$ (assuming that process P has n transitions); some of the G_i may refer to the type of the first message in some queue.
3. Execute one procedure from the set $\{A_i | G_i$ evaluates to $true\}$.

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

One main issue about this algorithm is the non-determinism of the transition selection; this non-determinism is also a vital part of the Estelle language definition. Thus, the unavoidable non-determinism of distributed, message-passing set of processes is included in the language.

3.2 Granularity and model

The grain of observation and execution is the Estelle transition. When a transition is fired, the Pascal body is executed and its execution is atomic: all instructions in the body are executed and their executions cannot be individually observed.

The representation of a computation in terms of sending events and consuming events is consistent with this grain. The execution of a transition can be particularized into a sequence of events. At most one consuming event placed at the head of the sequence can be eventually followed by a succession of sending events. The special case of a transition where no message is consumed or sent can be considered thanks to the notion of internal event.

No reference is made to the reception of messages by a process. Only the consummation of a message is observable.

3.3 Implementing the reexecution facility

The reexecution facility implemented in the Erebus debugger aims at helping the user in the following way: a first execution of the program is performed, using a DMPC to achieve high speed. At some point during this first execution, the computation is stopped. Then the debugger enters the interactive mode, where the user has access to each state reached by the initial computation. Because of the parallelism and interleaving of events in the distributed initial execution, each of the states presented to the user is in fact a member of an equivalence class: two global states are equivalent if and only if each process executes the same events in the same order. All the causal relationships between events are preserved whereas temporal properties are not conserved. One of the state of the class was reached during the initial execution, but it is impossible to know which one.

Logging data to enable replays The reexecution facility relies on some important properties of Echidna or Erebus programs, namely the fact that there is at most one message reception for each transition, that a reception specification indicates one sender only, that the communication is FIFO and point-to-point and that the topology of the interprocess channel network is static. Thanks to these features, the amount of data one has to log to change a non-deterministic automaton into a deterministic one is very limited. Assuming that every transition of a process has a unique identity number within the scope of this process, logging identities of the transitions fired during the initial execution is sufficient to replay the same sequence of transition. More precisely, each time the scheduler fires a transition, it appends

the transition's identity to the log file. All schedulers behave so during the initial execution. Schedulers log into their own local log files, without mutual synchronization.

Guiding the scheduler The set of schedulers performs a replay by resetting all processes to their initial states and then by reading their log files from the beginning. Upon each cycle of transition's selection and firing, each scheduler is bound to follow the log file data to select the transition to fire. When the imposed transition is not fireable (*i.e. its guard evaluates to* false), the scheduler waits until it becomes fireable. This means that the transition is waiting for a message which was consumed by the selected transition during the initial execution but has not arrived at this point of the reexecution. No other information is needed to replay the behaviour of each process, as the local states are recomputed implicitly by the reexecution of each transition: neither the contents of the variables, nor the type and field values of received messages are stored.

For sake of simplicity, we omitted in the preceding paragraphs the problems related to non-deterministic sources of values, such as wall clocks and random number generators. The solution implemented in the Erebus debugger is similar to the scheduler log system. During the initial execution, every call to a wall clock or random number service is trapped by the runtime and the value given to the process is logged. During replay, these service calls are answered directly by the runtime, which reads the values from the service call log file. The sequence of wall clock times and random number values is then the same than the initial one.

The current implementation of Erebus stores the log files of each scheduler into the local memory of the processor executing this scheduler. As the amount of information may be important, data must be compressed. To this aim, a data compression algorithm is used [9]. Nevertheless the maximal duration of an execution is limited by the local memory space. In order to reproduced an execution on any computer, the saved data must be encoded independently from the machine.

During the gathering, the probe effect induced by the debugger interference must be limited. No extra synchronization must occur. Hence schedulers do not exchange messages with a file system, in order not to increase the network load.

3.4 Breakpoints and execution control

Local breakpoints Common sequential debugging techniques make use of manual or automatic setting of breakpoints (automatic setting allows step by step execution) with machine instruction or source line granularity. Describing halting conditions with predicates on the state is also possible. Such features need to be considered in a distributed context. Considering the cost of global state computation, we want to keep the possibility of defining breakpoints on any single process. Such breakpoints are named *local breakpoints*, as they do not mention other processes' states or events. One can specify them as predicates on the local state or as a trace of local events. More formally, a set of breakpoints made of event traces defines a language and well-known techniques of language specification and parsing

are available.

Global breakpoints Global breakpoints specify conditions over more than one process. A simple form of these is a disjunction of local breakpoints: the computation stops as soon as one local breakpoint condition is satisfied.

However, this form of breakpoint is not powerful enough to express frequent behaviors of a distributed computation: one is often interested in pin-pointing situations such as overtaking of specific messages (e.g. release indication received before request indication), etc [6]. Such breakpoints are difficult to define precisely and to monitor [15], because they involve computations on disseminated local states, and such computations are costly and also non-instantaneous. A lot of theoretical and practical work has still to be done to determine the costs and algorithmic complexities of distributed breakpoint evaluation, *i.e.* what kind of breakpoints are useful and reasonably cheap to monitor [5].

Halting or restoring the computation An implementation of a distributed breakpoint mechanism must provide mechanisms to reset the global state into the minimal or maximal state when an halting condition is satisfied. This problem is similar to transaction rollback mechanisms in databases [2] and to failure recovery techniques in distributed systems [16, 10, 8]. Common techniques rely on checkpointing. The local states are periodically recorded and messages received by the processes between checkpoints are logged. The storage space occupied by the saved data increases indefinitely and periodic discarding of old information must be made to avoid running out of storage. A protocol must allow sites in the system to agree on which obsolete information can be discarded [2].

Snapshots With the help of the log files, any process behavior occurring during the initial execution can be replayed, by performing a deterministic reexecution from the initial state. While this technique saves a lot of space, because no global state are logged, it is costly from the reexecution time point of view. Therefore, saving the global state of the program from time to time can be useful. Periodic checkpoints are used by the replay module to restore the program state; the other states are reached by deterministic reexecution, starting from the restored global state.

3.5 User interface

Until now, we have addressed only the runtime aspects of a distributed debugger. A man-machine interface has to be designed to control the distributed kernel and the facilities. The quality of the debugging tool depends from this graphical user interface.

4 Conclusion

In this paper a compiler for Estelle programs [7] is used and its associated debugger has been presented. The concepts exposed in this paper are fundamental for defining what a

distributed debugger is. Parts of the debugger are now implemented, namely the execution replay, other parts are under study. These studies concerned the definition of a breakpoint system and its implementation. More precisely, the time and message complexities of interesting breakpoint languages have to be studied in order to find a trade-off between the power of expression and the computation costs of these breakpoints.

Acknowledgments

Acknowledgments are due to Claude Jard and Jean-Marc Jézéquel who designed and implemented the Echidna tool on which our Erebus debugger is based. We would also like to thank Carlos Maziero for his fruitful discussions about distributed system kernels.

Références

[1] M. Adam, Hurfin M., N. Plouzeau, and M. Raynal. *Distributed Debugging Techniques.* Research report 1459, INRIA, 1991.

[2] P.A. Bernstein and N. Goodman. Concurrency control in distributed data base system. *Computing Surveys*, 13,2:185–221, June 1981.

[3] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.

[5] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *The* 10^{th} *international conference on distributed computing systems*, pages 134–141, 1990.

[6] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the Hawaii International Conference on System Sciences*, 1988.

[7] C. Jard and J.-M. Jézéquel. A multi-processor Estelle to *C* compiler to experiment distributed algorithms on parallel machines. In *Proc. of the* 9^{th} *IFIP International Workshop on Protocol Specification, Testing and Verification, University of Twente, The Netherlands*, North Holland, 1989.

[8] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of algorithms*, 11:462–491, 1990.

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

[10] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software engineering*, SE-13(1):23–31, January 1987.

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

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

[13] E. Leu, A. Schiper, and A. Zramdini. Efficient execution replay techniques for distributed memory architectures. In Arndt Bode, editor, *Proc. of the Second European Distributed Memory Computing Conference, Munich*, pages 315–324, April 1991.

[14] M. Raynal, M. Mizuno, and M.-L.. Neilsen. *A synchronization and concurrency measure for distributed computations.* research report 610, IRISA, October 1991. 20 pages.

[15] R. Schwarz and F. Mattern. *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.* Technical Report, University of Saarland, Germany, 1991. 36 pages.

[16] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on computer systems*, 3(3):204–226, August 1985.

LISTE DES PUBLICATIONS INTERNES IRISA 1992

PI 624 SIGNAL AS A MODEL FOR REAL-TIME AND HYBRID SYSTEMS
Albert BENVENISTE, Michel LE BORGNE, Paul LE GUERNIC
Janvier 1992, 22 pages.

PI 625 ON THE CENTRAL-LIMIT THEOREM FOR TRACKING ESTIMATORS WITH SMALL GAIN - INFINITE HORIZON CASE
Bernard DELYON, Anatoli JUDITSKY
Janvier 1992, 16 pages.

PI 626 A MONTE CARLO METHOD BASED ON ANTITHETIC VARIATES FOR NETWORK RELIABILITY COMPUTATIONS
Mohamed EL KHADIRI, Gerardo RUBINO
Janvier 1992, 28 pages.

PI 627 CONSTRAINED MULTISCALE MARKOV RANDOM FIELDS AND THE ANALYSIS OF VISUAL MOTION
Fabrice HEITZ, Patrick PEREZ, Patrick BOUTHEMY
Janvier 1992, 40 pages.

PI 628 ON ITERATIVE REFINEMENT FOR THE SPECTRAL DECOMPOSITION OF SYMMETRIC MATRICES
Alexander N. MALYSHEV
Janvier 1992, 26 pages.

PI 629 STRUCTURAL OPERATIONAL SPECIFICATIONS AND TRACE AUTOMATA
Eric BADOUEL, Philippe DARONDEAU
Janvier 1992, 36 pages.

PI 630 EREBUS, A DEBUGGER FOR ASYNCHRONOUS DISTRIBUTED COMPUTING SYSTEM
Michel HURFIN, Noël PLOUZEAU, Michel RAYNAL
Janvier 1992, 14 pages.

PI 631 PROTOCOLES SIMPLES POUR L'IMPLEMENTATION REPARTIE DES SEMAPHORES
Michel RAYNAL
Janvier 1992, 14 pages.

PI 632 L-STABLE PARALLEL ONE-BLOCK METHODS FOR ORDINARY DIFFERENTIAL EQUATIONS
Philippe CHARTIER, Bernard PHILIPPE
Janvier 1992, 28 pages.

PI 633 ON EFFICIENT CHARACTERIZING SOLUTIONS OF LINEAR DIOPHANTINE EQUATIONS AND ITS APPLICATION TO DATA DEPENDENCE ANALYSIS
Christine EISENBEIS, Olivier TEMAM, Harry WIJSHOFF
Janvier 1992, 22 pages.

PI 634 UN NOYAU DE SYSTEME REPARTI POUR LES APPLICATIONS GEREES PAR UN TEMPS VIRTUEL
Janvier 1992, 20 pages.

ISSN 0249-6399