SAFETY OF COMPUTER CONTROL SYSTEMS 1985

(SAFECOMP' 85)

Achieving Safe Real Time Computer Systems

Edited by

W. J. QUIRK

IFAC

International Federation of Automatic Control

SAFETY OF COMPUTER CONTROL SYSTEMS 1985
Achieving Safe Real Time Computer Systems

SAFETY OF COMPUTER CONTROL SYSTEMS 1985 (SAFECOMP '85) Achieving Safe Real Time Computer Systems

Proceedings of the Fourth IFAC Workshop
Como, Italy, 1–3 October 1985

Edited by

W. J. QUIRK

Computer Science & Systems Division,
Atomic Energy Research Establishment, Harwell, U.K.

Published for the

INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL

by

PERGAMON PRESS

OXFORD · NEW YORK · TORONTO · SYDNEY · FRANKFURT

U.K.	Pergamon Press Ltd., Headington Hill Hall, Oxford OX3 0BW, England
U.S.A.	Pergamon Press Inc., Maxwell House, Fairview Park, Elmsford, New York 10523, U.S.A.
CANADA	Pergamon Press Canada Ltd., Suite 104, 150 Consumers Road, Willowdale, Ontario M2J 1P9, Canada
AUSTRALIA	Pergamon Press (Aust.) Pty. Ltd., P.O. Box 544, Potts Point, N.S.W. 2011, Australia
FEDERAL REPUBLIC OF GERMANY	Pergamon Press GmbH, Hammerweg 6, D-6242 Kronberg-Taunus, Federal Republic of Germany

Copyright © 1985 IFAC

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means: electronic, electrostatic, magnetic tape, mechanical, photocopying, recording or otherwise, without permission in writing from the copyright holders.

First edition 1985

Library of Congress Cataloging in Publication Data
IFAC Workshop (4th : 1985 : Como, Italy)
Safety of computer control systems 1985
(SAFECOMP '85)
1. Automatic control — Reliability — Congresses.
2. Computers — Reliability — Congresses. 3. Industrial safety — Congresses.
I. Quirk, William J. II. International Federation of Automatic Control. III. Title
TJ212.2.I338 1985 629.8'95 85–19088

British Library Cataloguing in Publication Data
SAFECOMP '85 (*Conference : Como*)
Safety of Computer Control Systems 1985 : (SAFECOMP '85) : achieving safe real time computer systems : proceedings of the fourth IFAC workshop, Como, Italy, 1–3 October 1985.
1. Control theory — Data processing I. Title II. Quirk, W. J. III. International Federation of Automatic Control IV. Series
629.8'312 QA402.3
ISBN 0–08–032570–X

These proceedings were reproduced by means of the photo-offset process using the manuscripts supplied by the authors of the different papers. The manuscripts have been typed using different typewriters and typefaces. The lay-out, figures and tables of some papers did not agree completely with the standard requirements; consequently the reproduction does not display complete uniformity. To ensure rapid publication this discrepancy could not be changed; nor could the English be checked completely. Therefore, the readers are asked to excuse any deficiencies of this publication which may be due to the above mentioned reasons.

The Editor

Printed in Great Britain by A. Wheaton & Co. Ltd., Exeter

FOURTH IFAC WORKSHOP ON SAFETY OF COMPUTER CONTROL SYSTEMS (SAFECOMP '85)
Achieving Safe Real Time Computer Systems

Organized by
The Associazione Nazionale Italiana per l'Automazione (ANIPLA)

Sponsored by
The International Federation of Automatic Control (IFAC) through
Comitato Nazionale per la Ricerca e per lo Sviluppo dell'Energia Nucleare e delle Energie Alternative (ENEA)
Raggruppamento Ansaldo S.p.A.
Centro Studi ed Applicazioni in Tecnologie Avanzate (CSATA)

International Programme Committee
E. de Agostino *(Chairman)* (Italy)
T. Anderson (U.K.)
S. Bologna (Italy)
Don Bristol (U.S.A.)
P. Ciompi (Italy)
G. Dahll (Norway)
B. K. Daniels (U.K.)
J. Dobbins (U.S.A.)
W. Ehrenberger (Germany)
H. Frey (Switzerland)
R. Genser (Austria)
E. Johnson (U.K.)
Th. Lalive d'Epinay (Switzerland)
R. Lauber (Germany)
A. Mariotto (Italy)
V. Massari (Italy)
P. G. Mirandola (Italy)
W. J. Quirk (U.K.)
J. M. Rata (France)
I. C. Smith (U.K.)
B. J. Sterner (Sweden)
U. Voges (Germany)
R. W. Yunker (U.S.A.)
R. Zoppoli (Italy)

National Organizing Committee
S. Bologna (ENEA) *(Chairman)*
S. Anderloni (ENEL)
E. de Agostino (ENEA)

此为试读,需要完整PDF请访问: www.ertongbook.com

PREFACE

Computers continue to be used in more and more areas. Once the province of high technology industry, they now pervade the electronics controlling everything from space vehicles to domestic appliances. Further, they are expected not only to control and monitor, but also to provide speedy and accurate information concerning the state of their environment; not only at the superficial level of their raw input data, but increasingly in terms of the likely fundamental causes of that data. Many research projects are already underway on the use of artificial intelligence to aid operator in plant control rooms. The robot, not so long ago firmly in the world of science fiction, is now an industrial reality. No doubt it will soon be a domestic reality too. Not only must such a device act safely despite its programmer, it must also act safely despite the baby, the dog and the precious antiques.

Since the first SAFECOMP, held in 1979 in Stuttgart FDR, much has been learned concerning the successful implementation of computer systems where safety is a primary concern. The operational benefits to be gained from using computers, in terms of the enhanced control and safety capabilities which can be implemented, are readily acknowledged. Subsequent SAFECOMPs in 1982 at West Lafayette USA and in 1983 at Cambridge UK have reported the continuing progress in achieving and demonstrating these benefits.

The initiative and impetus for these events continues to be TC 7, the 'System Safety and Security' technical committee of the European Workshop on Industrial Computer Systems. TC 7 is a body of experts concerned with all aspects of safety and security arising from the use of computers in potentially hazardous situations. It addresses the problems of protecting human wellbeing, the environment and the plant itself against hazards arising from failures in computer control or safety systems however these may occur. The objectives of TC 7 include the determination and dissemination of procedures to construct, document, test and verify the safe performance of such systems. The 'Call for Papers' for the present SAFECOMP'85 reaffirmed the continuing interest and activity in this area: many more replies were received than could possibly be accommodated in a workshop. Contributions were proffered from 15 different nations, of which 14 are represented in this volume.

These papers cover a wide range of topics; both hardware and software receive attention, as do theoretical and practical aspects both experimental and real life. The systems of interest range from direct process control through robotics to operator assistance, with safety aspects being central in each case. Construction techniques, including diversity, are balanced against reliability assessment techniques.

The programme committee wish to record their thanks to the sponsoring organisations: the International Federation of Automatic Control (IFAC), the Associazione Nazionale Italiana per l'Automazione (ANIPLA), the Comitato Nazionale per la Ricerca e per lo Sviluppo dell'Energia Nucleare e delle Energie Alternative (ENEA), the Raggruppamento Ansaldo S.p.A., and the Centro Studi ed Applicazioni in Tecnologie Avanzate (CSATA); also to the National Organising Committee and ENEA for their administrative efforts, to TC 7, particularly to the past and present chairmen W.D. Ehrenberger and J.-M.A. Rata whose enthusiasm has kept the committee united through financially troubled times, and to the Safety and Reliability Society of Great Britain for their supportive efforts. The editor is grateful for the assistance of the staff of the IFAC Publisher Pergamon Press and of Mrs. W.A. James of Harwell in the preparation of these proceedings. It is hoped that this event continues the fine tradition of previous SAFECOMPs and will lead to many more in the future.

W.J. Quirk
AERE Harwell

CONTENTS

ORIGINAL PAGE IS
OF POOR QUALITY

Copyright © IFAC SAFECOMP '85
Como, Italy, 1985

STRUCTURING PROCESSES AS A SEQUENCE OF NESTED ATOMIC ACTIONS

F. Baiardi and M. Vanneschi

Dipartimento di Informatica, Universita di Pisa, Corso Italia, 40, 56100 Pisa, Italy

Abstract. Implementation of atomic actions by means of concurrent programming constructs is discussed. It is shown that several trade-offs between performance and reliability may be obtained when an atomic action is defined through the composition of constructs and not as an elementary one.

Several alternative implementations are then discussed with reference to the ECSP concurrent language.

Emphasis is placed on process structuring, parallel activation and termination.

Keywords. Atomic action; Concurrent programming; Distributed system; Reliability; Performance.

INTRODUCTION

It is by now widely recognized that the notion of distributed atomic action is essential for the development of reliable distributed software (Liskov, 83; Randell, 78; Lomet, 77).

A computation can be easily structured into a sequence of steps, each implemented by an atomic action. Because of the "all or nothing" property of atomic actions, a failed step does not modify its input data and hence can be easily recovered. Furthermore, the ability of nesting atomic actions supports high granularity for both error detection and recovery.

As a counterpart of these advantages, usually atomic actions constrain the degree of concurrency and of asynchronicity of a computation since, as an example, the components of the action are forced to enter and to leave the action simultaneously (Gray, 78). This condition is not necessary for an action to be atomic, but is imposed to guarantee atomicity independently from the semantics of the computation of each component (Jensen, 83). Several advantages, in terms of efficiency, are possible if the previous constraint is relaxed provided that it is allowed by the action semantics. This can be done when the notion of atomic action is not primitive in the adopted programming language. Instead, atomic actions should be defined in terms of the composition of constructs for error recovery and synchronization. In this way the degree of synchronization and the amount of data to be saved may depend upon the particular computation that is made atomic.

In this paper we describe the implementation of distributed atomic actions in terms of the constructs of ECSP, a message passing concurrent language defined by a set of extensions to the CSP model (Hoare, 82). The extensions allow the programmer to define, among others, a continuation on the occurrence of a command failure. ECSP has been designed to be able to support fault-tolerance policies based upon cooperation among autonomous partners with the same authority. Process cooperation is based not only upon communication but also upon process structuring and process termination handling. Termination handling is the fundamental mechanism for error detection and confinement, as well as the basis for forward and backward error recovery. Forward recovery is possible by establishing alternative communications with processes that are functionally equivalent to those that are supposed to be faulty. Backward recovery is instead based upon the ability of preventing the update on the state of a process when one of its commands fails.

ECSP is briefly reviewed in sect. 2, where we also discuss the failure model we assume, as well as some assumptions on ECSP implementation in a distributed system. The implementation of atomic actions in ECSP is discussed in sect. 3. In sect. 4 we show some solutions to a problem of stream manipulation. Each solution is based upon the notion of atomic action, but it offers a different trade-off between efficiency and reliability.

COOPERATION AND ERROR RECOVERY IN ECSP

We give here a short introduction to the main constructs of ECSP (Baiardi, 81, 84a, 84b). The sequential part of the language, which is Pascal-like, and some concurrent constructs not relevant here will not be described. Furthermore, some familiarity with the CSP model is assumed.

Communication and nondeterminism Control

Processes of an ECSP program can exchange values through the execution of input/output (i/o) commands. The result of the joint execution of an output command A!c(expr) and of an input command B?c(y) in processes B and A respectively, is the assignment to y of the value of expr. The communication has place through typed channels, each one individuated by the triple (source process, destination process, message type). The message type is a pair (constructor, type of value) where the constructor is an identifier that may be absent. As an example, the two previous i/o commands exploit a channel (A, B, (c, type of y)).

In ECSP communications can be asynchronous and each channel contains a constant amount of buffers implementing a FIFO queue. A synchronous communication with rendez-vous may be obtained as a particular case by not declaring any buffers for a given channel.

ECSP channels are not shared objects, instead they are considered as data structures local to the receiver process. As an example, the declarations of the amount of buffers appear in the receiver process only. Because of the absence of shared objects, we can avoid the introduction of protection mechanisms distinct from those for communications (Baiardi, 84c).

Both dynamic and static channels may be defined. A dynamic channel is exploited when the partner name in an i/o command is given by the value of a processname variable. The range of values of a processname X declared in a process P is given by the names of processes that can be referred by P and the undefined value Ø. The following operations are defined on X (they can be executed only by P):

a) connect(X,name), connect(X,Y): the constant value "name" or the value of the processname Y is assigned to X. All the commands using X refer now to a process distinct from the one referred by the previous value of X;
b) detach(X): the undefined value Ø is assigned to X. All the i/o commands using X do no longer refer to any process;
c) is(X) : this is a boolean functions that returns true only when the value of X is different from Ø;
d) eq(X, name), eq(X,Y) : this is a boolean function that returns true when the value of X is "name" or is equal to that of Y.

Dynamic channels are the main ECSP mechanism to implement protection and reconfiguration of communication channels. They support the implementation of policies based upon the "minimum privilege" principle (Denning,76), since a process can create a dynamic channnel to communicate with another one and destroy the channel as soon as it decides that the communication is terminated.

Taking into account dynamic channnels, we can define the termination conditions of an i/o command as follows:

a) with successs: for an output command this means that the message value has been either copied into a channel buffer or assigned to a variable in the receiver. In the case of an input command this means that a value has been assigned to the target variable;
b) with failure because of the partner termination;
c) with failure because the (dynamic) channel has been disconnected: this failure signals that the partner has uptated (or has not yet updated) the value of a processname variable and hence no communication is possible on the channel individuated by the command;
d) with failure because of the unability to communicate with the node where the partner is allocated; this outcome models decisions of the diagnostic procedures in the communication protocol of the language run time support.

The failure of an i/o command may be handled by the onfail clause

```
   i/o command  onfail
               terminated : CL1
             disconnected : CL2
                   failed : CL3
```

After a failure, the appropriate command list, if present, is executed and then the execution goes on as if the command were successful; an unhandled failure results in the process termination with failure.

To control and express nondeterminism, ECSP adopts the alternative and repetitive commands with input guards of CSP. Each guard may be associated an integer variable to express the priority of the corresponding alternative. This allows the programmer to express directly scheduling policies and real-time decisions.

The termination conditions of guarded commands are similar to those of CSP. The failure of an alternative command may be handled by the onfail clause.

Process Structuring

ECSP supports nesting of parallel commands. Therefore each program has a hierarchical-parallel structure to support the implementation of the required degree of modularity and parallelism. As an example, a process P can activate processes P1 and P2 by the parallel command

```
   out(X,Y,Z)
    [P1 :: in(X,Z); ...; out(T)
    ||P2 :: in(X,Y); ...; out(Z)]
   in(T,Z);
```

This causes the simultaneous activation of P1 and P2, and suspends the execution of P until the command is terminated, i.e. until both P1 and P2 are terminated. A process terminate by the execution of the command terminate(succ) or terminate(fail). A parallel command terminates successfully only when all the activated processes terminate successfully.

The nested structure of processes induces a visibility rule for process names. Let PR(Pi) be the set of process that can be referred to by Pi. If Pi has been activated by process P through a command including processes P1, ..., Pn then PR(Pi)=PR(P)U(P1, ...,Pi-1, Pi+1, ..., Pn). In the previous example, PR(P1)=PR(P)U(P2), PR(P2)=PR(P)U(P1). This implies that a process Q that can refer to P cannot refer to P1 or to P2, while P1 and P2 inherit from P the knowledge of the name Q. Hence P1 and P2 can send/receive messages to/from Q, while Q cannot detect whether the partner of a communication is P or P1 or P2. Implementation of nested parallel commands is discussed in (Baiardi, 84a, b).

Communication among P and P1 and P2 is implemented through import/export (I/E) lists. The clause out(list of variables) in P specifies the export list, i.e. the variables whose values are to be transmitted to P1 and P2. The clause in(list of variables) in P specifies the variables that will be assigned the values received from P1 and P2. P1 and P2 include, in turns, an import and an export list to specify the variables that receive a value from, and those whose values are to be transmitted to, P. The value of a variable in an export list is assigned to the variable with the same name in the corresponding import list. To avoid ambiguities, the export list of P1 and P2 have to be disjoint. In the given example, on the execution of the parallel command, the values of X, Z are assigned to variables in P1 while the values of X, Y are assigned to variables in P2. When the parallel command ends, the values of T in P1 and of Z in P2 are transmitted to P and assigned to the corresponding variables.

According to the ECSP semantics, the assignements to variables in the input list of P are executed only if the parallel commands terminates successfully. This is the ECSP mechanism for backward recovery policies as well as the basis for the definition of atomic actions.

The failure of a parallel command may be handled by the onfail clause, where a distinct recovery action may be executed for each distinct subset of failed processes.

Failure Model and Language Implementation

In the following we assume that any fault in the system corresponds to an inconsistent behaviour of a set of processes. By inconsistent behaviour we mean, for example, a process unexpected termination or an erroneous communication attempt.

The language run time support should include mechanisms that transform an error into an inconsistent behaviour. As an example, the support transforms the crash of a processor into the termination with failure of all the the processes (Pi) allocated to the processor. In turn, this induces the termination with failure (kind b) of any i/O command referring to a Pi. When, instead, a Pi is executing a parallel command, its anomalous termination has to be masked until the parallel command terminates. Hence a proper implementation has to be adopted to prevent the crash from affecting the processes of the parallel command not allocated to the crashed processor.

Another essential hypothesis is that, even when physical communication media are unreliable, the diagnostic mechanisms at the communication level are reliable. This corresponds to say that the termination conditions of an i/o command are uniquely distinguishable. The mechanisms to grant the non ambiguity condition about the kinds of a command termination are implemented:

a) in a way that strongly depends upon language constructs for the cases a),b),c) of sect. 2.1;
b) by implementing mechanisms as diagnostic procedures in the run time support for the case d).

IMPLEMENTATION OF ATOMIC ACTIONS IN ECSP

The basic mechanism to implement atomic actions is the parallel command with I/E lists. In this section we introduce some sufficient conditions to guarantee that a set of parallel commands is an atomic action.

Let us consider, at first, a single command

```
out(LO)
[ P1:: in(LO1); ... ; out(LI1)
||        .
          .
          .
||Pn:: in(LOn); ... ; out(LIn)]
in(LI) onfail ....
```

If the following conditions are verified, the command is an atomic action:

c1) each Pi can communicate only with processes belonging to the command;
c2) the output of the command are the values to be assigned to LI and they depend only upon the values transmitted by LO.

The two conditions guarantee, respectively, idempotency and restartability of the command. Condition c1 can be relaxed: consider the case when a Pj invokes a server process Q to execute a function f on a value X. If X is not stored into Q's state after computing f(X), the communication between Pj and Q does not violate atomicity. Condition c2 guarantees that the outputs of the action can be made persistent, e.g. stored on an external device, only after the termination of the action.

It is worth discussing now some noticeable characteristics of the implementation of atomic action in terms of ECSP parallel commands.

Consider the case when a Pj fails. After this failure, it is known that the work executed by the other processes is useless, since the command will fail anyway and hence no results will be transmitted to the process executing the parallel command. Thus, in this case, all the other processes of the parallel command could be aborted. In ECSP, instead, the other processes terminate only when they detect that the failure of Pj prevent them from accomplishing their tasks. This choice preserves the autonomy of each process whose behaviour cannot be affected by other processes. In particular, a process cannot be forced either to terminate or to take part in an interaction.

The abandonement of the autonomy principle, though it could increase the efficiency of some computations, may introduce "privileged" system components that could strongly reduce the overall reliability and, furthermore, would make it very difficult to express forward recovery policies. Anyway, since a process is informed of the failure of a Pj through the failure of its i/o commands referring to Pj, it is very simple to stop a computation by propagating process termination.

Consider now a set S of parallel commands (PC1, ..., PCn), where each PCi activates processes (Pij, $1 \leq j \leq mi$) that receive values in LOi and returns values in LIi. S is an atomic action if we can guarantee that:

c3) a Pij activated by PCi can communicate only with a process Pkl activated by PCk;
c4) communications between processes not belonging to the same parallel command exploit dynamic channels that are connected before executing the parallel command and detached when a failure occurs;
c5) all the parallel commands end in the same way.

Condition c3 corresponds to c1 since it guarantees that no intermediate result is visible outside the action. Condition c4 is instead related

此为试读,需要完整PDF请访问: www.ertongbook.com

to the scope rule for process names. Since two processes Pij and Pkl activated by distinct parallel commands of S cannot refer to each other directly, an anomalous termination of Pij cannot be detected by Pkl. When the communication is implemented by dynamic channels, these channels can be detached by the process activating Pij and thus the i/o commands of Pkl will return a failure of kind "disconnected channel". Furthermore, since the state of a dynamic channel is reinitialized when the channel is detached or connected to a new partner, c4 guarantees idempotency with respect to communications.

By structuring an atomic action into several parallel commands it is possible to increase the amount of concurrency among components.

As illustrated in the following, conditions c3 and c4 can also be relaxed depending upon the semantics of the action.

ALTERNATIVE IMPLEMENTATIONS OF ATOMIC ACTIONS

To show alternative implementations of atomic actions we will refer to the following problem: given two data streams, S1 and S2, produced from data structures IS1 and IS2 respectively, compute the stream S3 defined as follows:

$S3j=g(h(S1j),S2j)$ if $j \leq L1$, $j \leq L2$; $S3j=h(S2j)$ if $L1 \leq j \leq L2$;
$S3J=k(S1j)$ if $L2 \leq j \leq L1$
where Li is the length of Si and Sij is the j-th element of Si.

The solutions we present are all structured as follows. The program is partitioned into two sets of processes: processes in the first one receive IS1 and IS2 from processes in the other and return to them the whole stream S3. The second partition will be modelled by a single process called Other: this is not a limitation since parallelism inner to Other may be expressed by nested processes.

Let us consider a first simple solution that does not take into account reliability or robustness.

```
    Prog::[P1||P2||P3||P4||Other]
```

P1 and P2 receive IS1 and IS2 from Other, and produce S1 and S2, respectively. As soon as an element of the stream is produced, P1 sends it to P3 and P2 to P4. P1 and P2 end when the last element of the corresponding stream has been produced.

The programs of P3 and P4 are:

```
P3::<declarations>
    begin
       *[P1?new ⟶ P4!h(new)];
       terminate(succ);
    end
P4::<declarations>
    begin
    *[P2?x1  ⟶ rec:=false;
       * [ not(rec), P3?x2 ⟶ rec:=true;
    R1                        append(S3,g(x2,x1))
       ]   ;
          if not(rec) then begin
                              append(S3, k(x1));
                         *[ P2?x1 ⟶ append(S3, k(x1))]
                                    end
     □ P3?x2   ⟶ rec:=false;
          * [ not(rec), P2?x1 ⟶ rec:=true;
     R2                         append(S3, g(x2,x1))
          ];
          if not(rec) then begin
                              append(S3, x2);
                     *[ P3?x2 ⟶ append(S3, x2) ]
                              end
     ];
       terminate(succ)
     end
```

Process P3 applies function h to all elements of S1. The termination of P1 will produce the failure of the guard and the successfull termination of P3. P4 waits for a pair of elements, one from P2 and the other from P3. When both producers end, the repetitive command ends successfully. If one stream is longer than the other, then either R1 or R2 will end because of the failure of the input guard. In this case, P4 will wait only for elements of the longer stream. Notice that P3 and P4 assume that the communication from the producer fails iff the stream is terminated.

To transform the computation of P1-P4 into an atomic action, we can nest the processes into another one that receives IS1 and IS2 and returns S3. the corresponding program is :

```
  Program::[ Rstream||Other]

  Rstream::<declarations>
           begin
             Other?IS1;
             Other?IS2;
             done:=true;
             nret:=0;
             repeat
                out(IS1,IS2)
                 [P1||P2||P3||P4 ]
                in(S3) onfail begin
                                nret:=nret+1;
                                done:=false
                              end;
            until done or (nret>max);
            if done then Other!S3 else ...;
            terminate(succ)
          end
```

In this solution, P1 and P2 receive their inputs by I/E lists and P4 returns S3 to Rstream in the same way. If any Pi fails, this will prevent the assignement to S3 and the parallel command will be executed again. If we do not modify the program of P4, it will not detect the anomalous termination of P2 or P3 and thus it will append to S3 the elements received from the other producer. This can be avoided by forcing P3 and P2 to communicate explicitly their termination and by replacing the repetitive command in P4 by the following repeat command:

```
  repeat
    if not(end1) then [ P2?x1 → skip
                      □ P2?end() → end1:=true
                      ] ;
    if not(end2) then [P3?x2 → skip
                      □P3?end() → end2:=true
                      ] ;
    case end1, end2 of
      begin
        false, false : append(S3, g(x1,x2))
        true,  false : append(S3, x2)
        false, true  : append(S3, k(x1))
        true,  true  : skip
      end;
   until end1 and end2
```

If neither a value nor the message end() is received from a producer, then P4 will fail immediately, thus inducing the termination of the other producer. Notice that this solution is much more synchronous that the previous one since both IS1 and IS2 have to be received before the parallel command is executed. Other synchronization are introduced to stop the computation on the failure of a producer.

Let us consider now a solution where the atomic action is implemented by a pair of parallel commands.

```
Program::[Rstream1 || Rstream2 || Other ]

 Rstream1:: ...
      Other?IS1;
      repeat
        connect(sink, Rstream2);
        Rstream2?ready();
        out(IS1,sink)
        [P1||P3] onfail begin
                     (P1, P3) :<processsor recovery>
                        (P3)  : done:=false;
                                nret:=nret+1;
                                detach(sink);
                                Rstream2?detect();
                           end;
      until done or (nret>max);
      ...
 Rstream2 :: ...
       Other?IS2
       repeat
         connect(source,Rstream1);
         Rstream2!ready();
         out(IS2,source);
         [P2||P4] in(S3) onfail begin
                    (P2, P4)  :<processor recovery>
                       (P4)   : done:=false;
                                nret:=nret+1;
                                detach(source);
                                Rstream1!detect();
                                  end
      until done or (nret>max);
```

The dynamic channnels corresponding to the processname variables sink and source implement the communications among processes of the two parallel commands. To assure that these channels have been properly connected, Rstream1 and Rstream2 are synchronized by the message ready().

In the programs of Rstream1 and Rstream2, we have assumed that a failure of P3 or of P4 is due to a line fault. In this case the recovery action consists in a new execution of the parallel command in both processes. As an example, after the failure of P3, Rstream1 will detach the processname variable sink thus inducing a failure in P4 and the termination of the parallel command in Rstream2. Then Rstream1 and Rstream2 will exchange the message detect() and connect sink and source before executing again the parallel command.

It is possible that P3 (P4) is affected by a fault after having sent (received) the last message to (from) P4 (P3). In this case the disconnection of the dynamic channel has no effect on the partner: this is a consequence of the "uncertainity principle" (Gray, 78) and it will be considered as a "disaster" similar to that induced by a crash during the "commit" phase of a distributed atomic action.

The programs of Rstream1 and Rstream2 assume that the failure of the process working on the input sequence is due to a processor crash. An appropriate recovery action may be, in this case, the execution of a parallel command activating processes with the same input/output behaviour of the failed ones. As an example, in Rstream1

```
<processor recovery>= detach(sink);
                      Rstream2?detect();
                      connect(sink, Rstream2);
                      Rstream2?ready();
                      out(IS1, sink)
                      [P1'||P3']
```

where Pi' may be a copy of Pi allocated to a distinct processor.

The programs of P1, P3, P4 in this solution may be the following:

```
 P1:: in(IS1)
      begin
        while IS1≠0 do
            begin
              produce new;
              P3!new onfail terminate(succ);
            end;
        P3!end()
      end
 P3:: in(sink)
       begin
       [P1?el → sink!h(el)
      □P1?end() → sink!end()
       ] ;
       terminate(succ)
       end;
 P4:: in(source)
      begin
       end1:=end2:=false;
        repeat
            ...
        until end1 and end2;
       terminate(succ);
      end
```

The program of P2 is similar to that of P1. Notice that the failure of the output command is transformed into the termination with success of P1 or P2.

The implementation of the distributed atomic action by a pair of parallel commands supports a larger degree of concurrency. As a matter of fact, we have that:

i) IS1 and IS2 are received concurrently by distinct processes;

ii) as soon as S1 has been produced, Rstream1 may continue its execution without waiting for the termination of the parallel command in Rstream2.

This increase in concurrency is paid in terms of reliability: as previously discussed, a fault in P2 or P4 cannot be recovered when the parallel command of Rstream1 is already terminated.

Another trade-off between performance and reliability can be achieved if we require that the two parallel commands are initiated asynchronously, i. e. the message ready is eliminated. In this case, the input command of P4 and the output command in P3, using respectively sink and source, could fail since the other process has not yet connected its processname variable. Since in ECSP this kind of failure can be distinguished, the process can recover the failure and attempt later the communication.

This behaviour can be described, in the case of P3, by the following program:

```
P3:: in(sink)
        end:=false;
        repeat
        [ P1?el ⟶ if empty(S') then x:=el
                                  else append(S', el);
                  done:=true;
                  sink!el onfail
                           disconnected : done:=false
        □ P1?end() ⟶ end:=true
        ];
        until end or done;
        if end then begin
              Timer!delta;
          *[ not(done), Timer?delta ⟶
                  sink!x onfail
                          disconnected : done:=false
          ] ;
             Timer!stop();
                   end;
       while not(empty(S')) do
              begin
                x:=first(S');
                sink!x;
              end;
       if not(end) then ⟨previous program for P4⟩;
```

We have assumed that S1 is not empty. Timer is a process that, after receiving a time interval delta, sends a signal each delta units of time, until it receives a message stop(). The loss of reliability is due to the fact that P3 can detect that the dynamic channel has been disconnected only after it has succeded in sending at least one message. When instead, P4 fails before receiving the first message, P3 will assume that the channel has still to be connected and hence will go on attempting the communication.

Other trade-offs are possible, but it is important to notice that in the same program we can have several distinct implementations of atomic actions, even nested one into the other. Furthermore, the ability of nesting parallel commands, and hence atomic actions, makes it possible to recover some faults not detected by inner actions.

CONCLUSION

We have shown how it is possible to define nested atomic actions using a small set of programming constructs. These constructs support several trade-offs between reliability and efficiency, thus allowing to exploit information about the semantics of the action to be implemented.

As far as concern the implementation of these constructs, it can be shown that no efficiency is lost by obtaining atomic actions through the composition of constructs (Baiardi, 81, 84b).

REFERENCES

Baiardi, F. and others, (1981). Mechanisms for a robust multiprocessing environment in the MuTEAM kernel. Proc. of 11th Fault Tolerant Computing Symp., Portland,June 1981, pp.20-24.

Baiardi, F., Ricci, L. and Vanneschi, M. (1984a). Static checking of interprocess communications in ECSP. ACM SIGPLAN Symp. on Compiler Construction, Montreal, June 1984.

Baiardi, F. and others, (1984b). Distibuted implementation of nested communicating processes and termination. 13th Int. Conf. on Parallel Processing, Aug. 1984.

Baiardi, F. and others, (1984c). Structuring processes for a cooperative approach to fault tolerant distributed software. Proc. of 4th Symp. on Reliability in Distributed Software and Database Systems, Silver Spring, Oct. 1984, pp. 218-231.

Denning, P.J. (1976). Fault tolerant operating systems. ACM Computing Surveys; 8,4, pp.359-389Gray, J.N. (1978). Notes on database operating systems. in Operating systems - An advanced course, Lect. Notes in Comp. Science, Springer & Verlang, Berlin.

Hoare, C.A.R. (1982). A calculus of total correctness for communicating processes. Science of Computer Programming; 1,1, pp.49-72.

Jensen, E.D., and others, (1983). Distributed cooperating processes and transactions. ACM SIGCOMM Symposium, Oct. 1983, pp.98-105.

Liskov, B. and Scheifler, R. (1983). Guardians and actions: linguistic support for robust, distributed programming. ACM TOPLAS; 5, 3, pp. 381-404

Lomet, D.B. (1977). Process structuring, synchronization and recovery using atomic transactions. ACM SIGPLAN Notices, 12, 3.

Randell, B. Lee, P.A. and Treleaven, P.C. (1978). Reliability issues in computer system design. ACM Computing Surveys; 10, 2, pp. 123-165.

Copyright © IFAC SAFECOMP '85
Como, Italy, 1985

A DYNAMIC SYSTEM ARCHITECTURE FOR SAFETY RELATED SYSTEMS

W. J. Quirk

Atomic Energy Research Establishment, Harwell, Oxfordshire, UK

Abstract. Safety related applications usually demand the provision of redundant resources within the system and some method of reconfiguration when a failure is detected. One problem with such an approach is that it has proved to be very difficult to design, implement and test adequately this reconfigurability. A dynamic system architecture is described which obviates some of these difficulties. This architecture takes advantage of the fact that the processing associated with any set of inputs and at any instant of time is of finite duration. By arranging for sufficient parallel redundancy to be available so that the system is not compromised by a single process instance failure, system error recovery becomes almost trivial. There is no need to recover the single instance failure (because of the available redundancy) and future processing will be initiated by normal process initiation. Little error-recovery specific procedure is necessary other than producing an effective fail-stop processor. The efficient implementation of such a system depends crucially on a number of issues. These include a novel, fully distributed scheduling procedure and the topology and functionality of the underlying communication system. The implications of such an architecture for overall system safety include the effects and benefits of software diversity and the possibility of producing systems which are to some extent proof against their own design errors.

Keywords. Architecture, diversity, fault tolerance.

INTRODUCTION

Safety related applications usually demand the provision of redundant resources within the system and some method of reconfiguration when a failure is detected. However, one major problem with such an approach is that it has proved to be very difficult to design, implement and test adequately this reconfigurability. Much useful work has been done on this problem, but most approaches rely on the assumption that the software is fully functional when an error occurs /LaSh82/, /Lomb84/ and that all errors are due to the hardware. Published figures reveal that a sizeable proportion of complete system failures are brought about by a failure to reconfigure correctly while attempting to recover from what was intended to be a recoverable error /Toy78/. The assumption that the software is totally error free seems hardly justified in most cases and the safety implications of such failures can be extremely serious.

This paper describes the basis of a dynamic system architecture which obviates some of these difficulties. The architecture takes advantage of the fact that the processing associated with any set of inputs and at any instant of time is of finite duration. The fact that the processing procedure may be repeated a potentially unlimited number of times does not mean that a process implementing this procedure need have a potentially unlimited life. Rather, new processes can be initiated when required and can terminate when they have completed a particular processing sequence.

By arranging for sufficient parallel redundancy to be available so that the system is not compromised by a single process instance failure, system error recovery becomes almost trivial. There is no need to recover the single instance failure (because of the available redundancy) and future processing will be continued by normal process initiation. No error-recovery specific procedure is necessary other than producing an effective fail-stop processor, a topic already addressed by a number of workers /CrFu78/, /ScSc83/, /Kenw84/, /Schn84/.

The efficient implementation of such a system depends crucially on a number of issues. These include a novel, fully distributed scheduling procedure and the topology and functionality of the underlying communication system. Such an architecture has several interesting implications for overall system safety. Of particular interest are the effects and benefits of software diversity to overcome problems of common-mode failure and the possibility of producing systems which are to some extent proof against their own design errors.

FUNCTIONAL REDUNDANCY

In a previous paper /GiQu79/, the basis of functional redundancy was described. The provision of redundant input data is of course a standard requirement for safety systems and the provision of redundant data processing, usually by identical replication, is also well established. But within single computing systems, the provision of such redundancy needs careful design. Otherwise, reliability 'bottlenecks' may occur, where the benefits of redundancy in one part of the system are lost because of potential common mode failures in another. The aim of functional redundancy is to achieve a uniform level of coverage throughout the system, taking into account any inbuilt redundancy available due to replication of input sensors or output actuators available. The approach taken is to trace the data flow through the system and to ensure that independent processing paths are followed so that a failure on one path does not affect other redundant data. Not only must there be separate physical data paths through the system, but, in order to obviate common mode failures, separate software must be utilised on the different data paths.

This previous paper also observed that redundancy in the input data was provided so that failures in the input sensors did not compromise the integrity of the system. This implies that the system can tolerate a limited amount of faulty input data and, in particular, a number of transient input errors. But there is no difference between a faulty input and a failure within the system along the path followed by that input, providing the failure is isolated to just that path. Thus, with the provision of suitable internal redundancy, it is possible for the system to be essentially unaffected by isolated transient failures. Indeed, if the normal output range of a process is augmented with a recognisable 'failed' value, then subsequent processes using that process output can take appropriate safe action. This can be achieved even without the augmentation, but it is easier to implement and demonstrate if it is done. With proper design, system safety can be maintained through single isolated failures.

The problem then is to design the system so that all failures are are transient and isolated. Clearly no system can be made proof against total failure; the architecture proposed below will be demonstrated to be proof against single internal failures which are sufficiently separated in time. The precise definition of this sufficient separation will be derived later.

The first step in rendering all failures transient is to note that there is no requirement to dedicate physical resources to functional units on a long-term or near-permanent basis. The 'standard' architecture for repetitive functionality has the form shown in figure 1. The WAIT interval is often omitted, with the program cycling as often as possible. However, the system requirement, properly assessed, will yield the required repeat interval. This fragment is then equivalent to that shown in figure 2.

```
REPEAT FOREVER
   [
   WAIT REQUIRED TIME INTERVAL
   READ INPUTS
   DO CALCULATION
   WRITE OUTPUTS
   ]
```

Fig 1. Static Scheduling Fragment.

```
WAIT REQUIRED TIME INTERVAL
RESCHEDULE NEW COPY OF SELF
READ INPUTS
DO CALCULATION
WRITE OUTPUTS
EXIT
```

Fig 2. Dynamic Scheduling Fragment.

Practically all the system components can be implemented in this second fashion. System 'long term' memory can be implemented as overlapping finite duration 'data server' processes. Only the system peripherals cannot be made dynamic, but these must be replicated in any case to provide a suitable level of fault tolerance and safety. Thus implementing the system in this manner hinges on the rescheduling procedure and the rest of this paper considers only the messages associated with this procedure.

In order to render the former fragment fault tolerant, some other part of the system needs to oversee its performance and, when a failure is detected, attempt to allocate the fragment to an alternative resource. This is precisely the operation in the second line of the latter fragment. The difference is that this operation is used only rarely in the former architecture (and then only at a time of stress for the system), whilst it is regularly used in the latter. Thus the procedure is extremely thoroughly tested and great confidence placed in it.

One should also notice that the characteristics of failures change subtly as systems are made more and more reliable and fault tolerant. Because there are less failures observed, they tend to be more complex than the 'simple', 'common' failures observed in less reliable systems. The difference between transient hardware failures and software failures also becomes blurred. The software is likely to function properly for all but a very small range of rarely-occuring special circumstances. Consequently, a failure on one cycle is unlikely to imply a failure in a subsequent cycle (providing that no code or data has been corrupted by the failure). This is just the same as the observed effects of a hardware transient; after a suitable reset, the device will resume normal operational characteristics. Thus, the latter architecture is inherently more resilient to these failures. It is this observation coupled with the confidence in the scheduling procedure which justify the assumption that failures will appear transient and isolated in time.

DYNAMIC SCHEDULING & FAULT TOLERANCE

The fundamental scheduling design is shown in figure 3. This mirrors the fragment in figure 2. However, some extra messages have to be added. First, the initial action of the process is to signal its grandparent. Second, after completing its normal processing, it waits another scheduling interval, and listens for the signal from its grandchild. In normal circumstances, this final signal arives as expected and the grandparent exits. If it fails to arrive, it could mean a number of possible errors have occured: that the child failed before scheduling the grandchild, that the grandchild failed before signaling the grandparent, or that the message was sent but not delivered. The corrective action in all cases is for the grandparent to attempt to schedule a grandchild directly. However, if the error was delivery failure, this would lead to two copies of the grandchild being activated. To prevent this, the initial action of each process is not to signal the grandparent first, but rather to try to signal any copy of itself. It expects no reply from its own generation, because there should not be such another copy. In order to prevent two copies killing each other, a priority scheme has to be included. This too can be made dynamic: by adding the process instance number to the physical processor identification and reducing the result modulo the total number of processors. This yields a number which is unique for any particular process instance. If processes only exits in response to a higher priority message from the same generation, then only the highest priority process will not be killed.

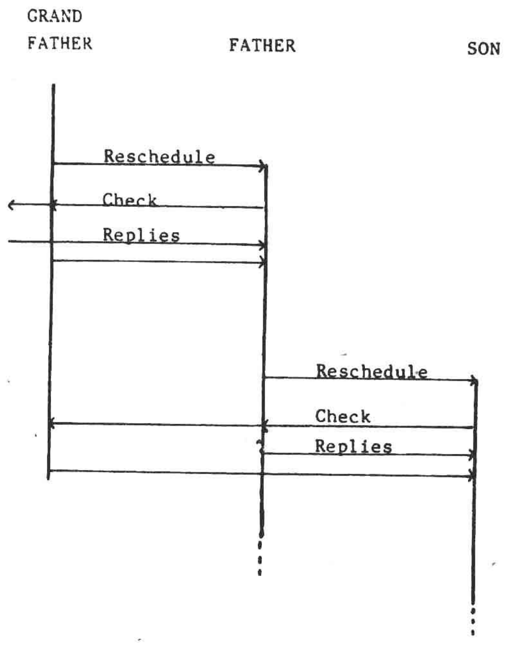

Fig 3. Principle Scheduling Transactions.

There are further failure possibilities in this system. Having scheduled its child, the grandparent may fail. If it merely dies quietly, there is no effect on the system. If it erroneously tries to schedule a grandchild when one is already successfully scheduled, then the mechanism described above will resolve the situation. But in all message sending situations, there is the possibility, at least in principle, of the sender going into a tight loop sending repeat messages. This poses two different problems. One is to ensure that the communications subsystem does not saturate, locking out all other processes. The other is to ensure that the system can effectively ignore the erroneous messages. In the case of process initiation, it is required to allow one process to run, rather than for an infinite sequence of processes to be initiated and almost immediately die. Support for this is based on the priority structure already discussed. This determines an unique processor to continue from any pair trying to perform the same process. Once this one has been established, any other can be killed by the working one. In order to limit the initial number to two, some support from the communications subsystem is proposed, as described below. Indeed, if process creations are sequenced by the communications system, the priority system could be replaced by a simple 'first-started continues' approach. But sequencing is difficult to enforce if the communications subsystem is itself replicated.

The ability to kill a process might initially seem to be a great problem; the possibility of a rogue processor killing the whole system must be made negligible. However, the power to kill used here is very restricted. A process is killed only by asking to be killed, and it can check the validity of that reply. When a process requests to be killed, it broadcasts only its name, not its instance number. Each process of the same name can reply with a kill for their own instance number. Thus only one generation can be killed by any single process. It is worth noting that the system will still recover if two processes of the same generation survive, or if they both die. In the former case, only one of their offspring should survive after the next reschedule; in the latter the normal reschedule procedure will restart the next generation. This is because the architecture is, in fact, resilient to many, but not all, twin failures too. This fact can be used to enforce a harder regime; that a whole generation should die if two or more identical processes appear. By allowing a process to continue only if at least one of its grandfather or father reply to its 'check' broadcast, one can remove any possibility of the system being compromised by an infinite reschedule loop. Such a system is still proof against a single failure, but it becomes vulnerable to more twin failures than would otherwise be the case.

此为试读,需要完整PDF请访问: www.ertongbook.com