ISSN 004-8917

THE AUSTRALIAN COMPUTER JOURNAL

VOLUME 19, NUMBER 2, MAY 1987

CONTENTS

SURVEY PAPER

49-55 Information Systems Planning: A Manifesto for Australian-Based Research
R.D. GALLIERS

CASE STUDY

56-62 The Transport Layer in a Microcomputer Network
M.R. FRY

SELECTED PAPERS FROM THE TENTH AUSTRALIAN COMPUTER SCIENCE CONFERENCE

63 Guest Editors' Introduction

64-68 A Linear Algorithm for Data Compression
R.P. BRENT

69-75 From a NIAM Conceptual Schema into the Optimal SQL Relational Database Schema
C.M.R. LEUNG and G.M. NIJSSEN

76-83 A Graphics Oriented Deductive Planning System
R.B. STANTON and H.G. MACKENZIE

84-91 Some Experiments in Decision Tree Induction
G.J. WILLIAMS

92-98 Reproducible Tests in CSP
C.J. FIDGE

99-104 Specifying the Static Semantics of Block Structure Languages
R. DUKE and D. JOHNSTON, and G.A. ROSE

SPECIAL FEATURES

68 Appointment of a New Editor

68 Call for Papers — ACSC-11

105-111 Book Reviews

112 The Journal's Referees

Published for Australian Computer Society Incorporated
Registered by Australia Post, Publication No. NBG 1124

Reproducible Tests in CSP

C. J. Fidge†

The difficulty of testing and debugging CSP programs is compounded by their inherent non-determinism. We illustrate a simple approach to recording and replaying traces of the control paths followed as a means of controlling this unpredictability. Where these traces would become unmanageably long, a variant of Bougé's repeated snapshot algorithm is used to put an upper bound on trace length. Some knowledge of CSP is assumed.

Keywords and Phrases: **CSP, non-determinism, debugging**
CR Categories: **D.1.3, D.2.5**

1. INTRODUCTION

As noted by Francis and Mathieson (1986), 'one of the greatest difficulties with concurrent programming concerns the lack of readily available experiential knowledge'. Faced with a similar problem we have implemented a small CSP-based language in LISP and have used it to gain some practical experience with writing and debugging CSP programs. Most of the classical features are available, including assignment, alternative, repetitive and skip commands (Hoare, 1978). Output guards (Buckley and Silberschatz, 1983) and the abort command (Hoare, 1979) are also supported. Nested processes and process arrays are not available.

Although irritatingly slow, this system has given us some experience with the practical problems encountered when testing non-deterministic, concurrent software. A number of simple debugging tools are available that allow the user to observe the execution of a selected process, or all inter-process communication associated with it. However after a program has misbehaved in some way, fruitless hours may be spent trying to reproduce the same situation with the debugging tools switched on! As Baiardi, de Francesco and Vaglini (1986) point out, 'the behaviour of a parallel program is not generally reproducible'. Also it would be very useful during debugging to study the same computation from a number of different viewpoints.

Most of the more sophisticated debugging systems for concurrent software appearing in the literature (e.g. see Gait, 1985; and Baiardi *et al.*, 1986) suffer from the same problem, relying on the debugging tools being appropriately configured if an error occurs. Experience with telecommunications systems suggests that infrequent synchronisation errors in large parallel programs can best be caught by allowing debugging tools to be left switched on all the time (Gupta and Seviora, 1984).

The solution presented here is to record a trace, or history, of a CSP program as it executes. This trace can then be used as a script to replay the test as often as necessary, allowing a given test to be studied over and over again.

2. BASIC MODEL

Our aim is to obtain, while a CSP program executes, a minimal trace of information that will allow us to reproduce this computation at a later time. Minimality is important to reduce the impact of the so-called 'probe effect' (Gait, 1986) where adding debugging code to a concurrent program changes the frequency of synchronisation errors in *incorrect* programs. In practice this phenomenon is quite noticeable and overcoming it has been the cause of a significant effort in the work of Baiardi *et al.* (1986). Arguably this 'effect' is not a legitimate concern since CSP makes no commitments regarding fairness (Hoare, 1978); adding trace statements can in no way affect the semantics of a CSP program.

There are two forms of non-determinism that affect selection of control paths in a CSP process, *external* and *internal* (Olderog and Hoare, 1986). Informally, external non-determinism is that influenced from outside the current process by the timing considerations of concurrency, e.g.

```
[A?x → ...
□B?y → ...]
```

while internal non-determinism involves the arbitrary choice of open boolean guards, e.g.

```
[true → ...
□true → ...]
```


†*Department of Computer Science, Australian National University, G.P.O. Box 4, Canberra, ACT, 2601.* *This paper was presented at the tenth Australian Computer Science Conference at Deakin University in Geelong, Victoria in February, 1987. Manuscript received February, 1987.*

In either case note that the control path selected is dictated entirely by which guards *succeed*. This applies to both alternative and repetitive commands, and since these are the only branching constructs in CSP, we can derive the following fundamental theorem:

> A partially-ordered history of successful guards completely characterises the control paths followed by a CSP program.

Therefore, to record a computation, we merely need to keep a separate trace for each process of all successful guards (assuming each guard in a process is uniquely identified). Replaying this computation is achieved by only allowing a guard to succeed if it is next in the trace. Note that the *total* ordering of events throughout the program will not necessarily be the same each time the computation is replayed; however this makes no difference to the result of the computation and is, in fact, undetectable except to an oracle with a global clock that can observe events occuring in all processes.

Thus recording, or storing, a trace can be achieved with the following rule:

> **Rule S1**: Each time a process *p* successfully executes a guard, a unique value identifying that guard is appended to the trace for *p*.

To implement rule S1 we add a new process ANALYSER to our basic program (adopting the terminology of Bougé (1985), the original program represents a *basic* computation, while the program modified for tracing is *marked*). The analyser process must have access to some permanent storage medium (typically files); for each basic process it creates a new trace, and accepts values from the process, appending them to the trace, and finally closes the trace when the basic process terminates. Each guard in the basic program is followed by an output to the analyser, i.e.

```
basic guard → ANALYSER!g;
                basic command list
```

where **g** is a unique value for each guard in the process. Although we will express our algorithms at the source level throughout this paper, this should not be taken to suggest that it would be practical, or desirable, to do so in practice. We assume that the CSP implementation is modified to achieve this effect. Also see section 3.2.1.

Each guard is identified by a unique number. Executing the marked program generates a separate trace for each process, each containing the guard identifiers of all successful guards. For example, for two processes A and B:

```
A: 1 1 1 1 1 ...
B: 1 2 3 4 3 5 3 5 ...
```

Replaying this computation simply involves reading the stored trace for each process, and only allowing a guard to succeed if it is next in the trace:

> **Rule R1**: A guard may only succeed if it is the next guard in the trace for this process. After a guard succeeds the trace is advanced.

For example:

```
...
LOAD?guard_id;
[guard_id = g ∧ basic guard →
        LOAD?guard_id;
        basic command list
        ...
```

where LOAD is a process that opens the trace for each basic process and sends each value in the trace. The special value 0 is sent when the end of the trace is reached.

In effect this removes all non-determinism from a CSP program, forcing it to follow the same control paths each time it is executed.

3. PRACTICAL CONSIDERATIONS

Here we consider two significant practical problems that become an issue when we step outside the bounds of 'pure' CSP.

3.1 Additional Non-determinism

In practice, there is another source of non-determinism that may affect a program, namely communication with the external environment. Although not included in CSP as defined by Hoare, or even in a 'practical' CSP-based language, occam† (INMOS, 1984), this feature must be implemented in practice. In our CSP system we have two special built-in processes, one which prints to the terminal, and another which reads from the keyboard. Similarly, Roper and Barter (1981) provide text file input and output as part of their CSP implementation.

If a program is replayed and provided with different external inputs, it generally fails (typically with a deadlock, or an abort due to all guards in an alternative construct failing) because it is attempting to follow control paths that are no longer valid. Therefore, in practice, it is important to note that a test can only be reproduced if the external environment behaves as it did when the trace was stored. This can easily be done by recording the stream of external input values (external outputs need not be traced).

Similarly, using a random number generator inside the CSP program, for example to generate test data,

```
SORTER!random*4
```

makes the program unrepeatable due to the additional non-determinism not controlled by the trace script.

3.2 Incomplete Traces

Another practical problem that must be considered is abnormal termination of a program being traced, by some external source. Often we want to 'kill' a program prematurely, either because it is deadlocked, caught in an infinite loop, or simply because we have already observed some event of interest. This may introduce difficulties when the replayed system

† occam is a trademark of the Inmos Group of Companies.

reaches the end of the trace. The situation varies depending on the CSP implementation.

3.2.1 Global termination

A CSP implementation using interleaving semantics running on a mono-processor makes it possible to kill *all* processes at the same global time. In this case the final global state will always be consistent only if reporting to the special analyser process is an atomic event that occurs at the same time a guard succeeds. If the tracing system is implemented at the source code level as above we may run into trouble if a process is 'slow' in reporting to the analyser. Consider the following code fragment:

```
A :: ...
    B?y;
    [y=50 → ANALYSER!1;⊗ ...
    □y=100 → ANALYSER!2; ...]
B :: ...
    [A!50 → ⊗ANALYSER!5; ...
    □A!100 → ANALYSER!6; ...]
```

If the program is killed when the processes are at the points marked ⊗ and we attempt to replay the program and continue past the end of the traces it is possible for A!100 to match with B?y (since there is no evidence in the recorded trace to the contrary) thus putting process A into an inconsistent state in which the trace specifies that it must select guard one, but local variable y forces it to follow guard two. For this reason we will in future assume that messages sent to the analyser form an integral part of the evaluation of a guard.

With this restriction it is possible to proceed beyond the end of traces without difficulty. Thus rule R1 can be changed to:

> **Rule R1'**: A guard may only succeed if it is the next guard in the trace or the end of the trace has been reached.

and the extra boolean condition added to all guards in the marked system is extended to

$$\text{guard\_id} = \mathbf{g} \vee \text{guard\_id} = 0$$

When replayed the program will continue past the end of the trace and continue *non-deterministic* execution unhindered. We have found it useful to get the LOAD process to print a warning message to the terminal when the end of the trace is passed so that the user realises that the program is no longer under any external control.

3.2.2 Incremental termination

A truly parallel, multiprocessor-based CSP implementation presents additional difficulties since processes cannot all be terminated at the same time. The programmer must incrementally terminate the separate processes one at a time. The distributed termination convention may result in erroneous control paths being followed and added to the trace while the various processes are being terminated. For example, if a process is waiting at the repetitive command in the following

```
    ...
*[B?x → ANALYSER!1];
[y≤5 → ANALYSER!2; ...
    ...
```

and process B is killed abnormally, this process may add '2' to the trace before it is also terminated. When replayed the process will now exit the repetitive command prematurely while B is still active. Note that this assumes that the 'kill' is done by some special CSP feature that allows the user to gracefully stop a process, and that the CSP implementation will inform other processes that B has terminated. The problem vanishes if the CSP implementation does not treat an external kill as 'termination'.

The solution is to ensure that the ANALYSER process is killed before any basic processes, thus preventing any incorrect values from being added to the trace.

4. RESTRICTING TRACE LENGTH

So far traces have been allowed to grow without bound. Obviously this is unacceptable for programs with many iterations, or for embedded programs that are expected to run permanently. Simply storing traces in some sort of circular buffer so that only the most recent values are kept is not an adequate solution because to restart processes at some point other than the beginning requires the state of the process, i.e. the value of the program counter (PC) and local variables at the time this segment of the trace started. We must ensure that these local process states represent a consistent 'slice' of the global program state.

The solution, therefore, is to periodically save the global state of the program and recommence tracing - the segment of the trace recorded before this global save can then be discarded to reclaim space. Replaying can begin at the oldest global state recorded. As we shall see, the global state is recorded one process at a time. Since a process may be killed from outside at any time, at least one complete state must be kept in reserve - saving two complete global states is preferable since it avoids the risk of having only a very short trace left if the program terminates soon after a save. Thus the traces for three processes might appear as follows

```
A: (A state) 1 2 1
   (A state) 2 1 2 1
   (A state) 2 1
B: (B state) 1 2 3 4 3 4 5
   (B state) 6 6 6
   (B state) 6
C: (C state)
   (C state) 1 1
```

at an arbitrary time during execution, where each trace consists of a number of *segments*, each starting

with a local state, followed by guard identifiers as before.

Notice that the length of trace segments varies. Whichever process fills its buffer first will initiate the global state save. In this example, when the third local state of process C has been saved, the first set of trace segments can be discarded. If the program is killed before C has a chance to save its state, the program may still be replayed because a complete global state is still available. Thus the maximum possible length of the trace for each process (in this scheme of keeping two states in reserve) is three times the maximum length of the trace segments for the process (this constant may be different for each process).

Replaying the program is achieved by loading the first state into each process, and then following the trace as before. Any subsequent states encountered are ignored.

4.1 Approach

Obtaining a global state in a CSP program is a classical problem that has received much attention, primarily due to its application to the distributed termination problem. One solution, the 'snapshot' algorithm, involves a process sending a *wave* of *markers* throughout the program, causing each process to suspend its basic computation and report its internal state to a special analyser process. Recently Bougé (1985) presented an improved version of this algorithm that allows for *repeated* snapshots of the global state (as a means of detecting a stable state, i.e. deadlock or termination).

Faced with a similar problem, we will adapt Bougé's algorithm to our needs. However, in contrast to Bougé, we will assume that an explicit analyser process is implemented. This avoids the need for the communications graph of the program to be strongly connected and reduces the number of marker messages that must be sent, albeit at the cost of supporting an additional process.

4.2 Normal Form

For convenience a CSP program may be expressed in a 'distributed normal form' in which each process consists of an initialisation section and a single large repetitive statement with *all* i/o events appearing in the guards (Apt, 1985). For our purposes we need an extended version of this definition in which all i/o events *and* all basic guards appear in the outer loop. This allows us to ensure that no basic guard or i/o statement can succeed without a check for trace buffer overflow. Also it gives each process the opportunity to receive marker messages at any time it could execute a basic i/o statement, thus avoiding deadlock in the case where the process we are attempting to communicate with is trying to send a marker rather than a basic message.

An arbitrary CSP process can be routinely converted into this normal form, typically by adding a variable that acts like an explicit program counter and protects each guard, only allowing it to succeed if the program counter in the original version of the program would be at the equivalent point in the code. Thus each process looks like

```
A :: initialisation code
     *[basic guard → ...
      □basic i/o event → ...
      ...
      ]
```

This process can be made semantically equivalent to the original version (although significantly less efficient due to the time spent in evaluating so many guards at each iteration), and can be made to terminate correctly (although distributed termination of nested repetitive statements in the original process can be complicated to represent, requiring the addition of explicit termination signals).

4.3 Storing a Trace

This section outlines the main rules our algorithm must implement. Adapting Bougé's terminology, a process is *white* if it is executing its basic computation, and *red* if it is involved in a marker wave.

Rule S1: Each time a process p successfully executes a guard, a unique value identifying that guard is appended to the trace for p.

Rule S2: Each process records how many guards have been successfully executed since the last state save.

Rule S3: When the number of guards successfully executed reaches a maximum bound for a process p, p must suspend its basic computation (i.e. turn red) and store its current state.

Rule S4: A red process may turn white only after exchanging a marker message with *all* of its *neighbours* (i.e. each process it communicates with in the basic computation).

Rule S5: A white process must stay alive and be ready to receive a marker message from any of its neighbours at any time (and hence turn red), even after its basic computation has finished.

Rule S6: Each process will save one local state during each wave. Any subsequent states may not be stored until the previous wave is complete.

Rule S7: A process may only terminate when all other processes have ended their basic computation.

Rule S8: The oldest trace segments may be discarded, while always maintaining at least the most recent complete global state intact.

Note that S3 allows any process to initiate a wave. It is therefore possible to have two or more waves spreading from independent points in the program at the same time; however, since messages merely need to be *exchanged* while red (direction is unimportant), there is no conflict when the waves meet, and they

fuse into one conglomerate state save. Rule S6 prevents an uncontrollable explosion of waves from being initiated.

Also note that a process must still be prepared to participate in a wave even after its basic computation has ended, otherwise a process that terminates long before the others would prevent a wave from being completed. Thus all processes must stay 'alive' until they have *all* completed their basic computation, after which no more waves can be initiated and it is safe to terminate the entire program.

4.4 Replaying a Trace

The rules for replaying a trace are comparatively simple.

Rule R1': A guard may only succeed if it is the next guard in the trace or the end of the trace has been reached. Any subsequent states in the trace may be ignored.

Rule R2: Each process loads its initial state from the start of the trace.

4.5 Implementation

Here we present the changes that must be made to a program to implement these rules for storing and replaying traces with an upper bound on trace size.

4.5.1 Storing a trace

Given a program in the extended normal form, we firstly consider the basic problem of recording the trace and obtaining consistent global states, ignoring termination:

a. add the following statements to the end of the initialisation part of each process

```
max := 100; s := 0;
ANALYSER!state;
```

Here **state** contains the current values of all basic local variables as well as the program counter. The variable *s* is used to count the number of guards successfully executed between saves (rule S2) and *max* represents the maximum size of a trace segment for this process (arbitrarily set to 100 here).

b. add to each basic guard the extra boolean condition

```
s < max
```

to prevent any attempt to execute a basic guard while the trace buffer is full (S3),

c. add to the start of each basic guarded command list

```
ANALYSER!g; s := s + 1;
```

to record the successful execution of a basic guard (S1),

d. add the following guarded command to the outer loop

```
s≥max → ANALYSER!state; s := 0;
    x_done := false; ...
*[¬x_done; x!save → x_done := true
 □¬x_done; x?save → x_done := true
    .
    . (for each neighbour x)
    .
]
```

to allow this process to spontaneously initiate a wave when its trace buffer is full (S3). Note that the process may not return to the main loop (i.e. go white) until it has exchanged a save marker with each process it communicates with (S4). Symmetric message passing is used so that it does not matter which direction the message goes. In either case we know that process 'x' is also red, or is going to be turned red by this message. Deadlock is avoided by not imposing any order on the exchange of markers with neighbouring processes.

e. for each process add the following guard to the outer loop for each neighbour x to allow the process to respond to a marker from another process (S5)

```
x?save → ANALYSER!state; s := 0;
    y_done := false; ...
    *[¬y_done; y!save → y_done := true
     □¬y_done; y?save → y_done := true
        .
        . (for each neighbour y
        .     except x)
        .
    ]
```

In this case x is known to be red and the process merely needs to ensure that all *other* processes it communicates with also turn (or are) red.

Unfortunately the above changes will prevent a marked program from terminating properly if the basic program used distributed termination to stop, since they create extra interdependencies among the processes. For example, if process A uses distributed termination to exit when process B terminates, the marked program will not terminate properly since B may now accept a marker message from A (i.e. A?**save** → ...) which will prevent B from terminating while A is still alive - the processes are deadlocked once the basic computation has ended.

Also rules S7 and S5 must be considered further as they force marked processes to stay active after their basic computation has terminated. The solution is to add explicit termination signals (similar to Apt and Francez, 1984) that allow marked processes to tell each other, and the analyser, when their *basic* computation has finished.

f. add to the initialisation part for each basic process

```
x_alive := true;
```

for all neighbours x,

g. add the following guarded command to the outer loop of each basic process

x_alive; x?**terminated** → x_alive := **false**

for all neighbours x. This allows a process to explicitly know whether its neighbours are still in their basic computation. This information is redundant and may not be received for processes that do not rely on distributed termination,

h. to explicitly exit the basic computation, and leave the main loop, add a guard which represents the negation of *all* basic guards. For example, with the following basic guards

```
*[b1 → ...
 □b2; x?y → ...
 □z!y → ... ]
```

the guard generated is

¬b1 ∧ (¬b2 ∨ ¬x_alive) ∧ ¬z_alive

which represents the state when all basic guards have failed, i.e. the basic computation is terminated. This guard is used to protect the following guarded command list

```
→ ANALYSER!terminated;
  *[x?save →
      ANALYSER!state;
      y_done := false; ...
      *[¬y_done; y!save →
          y_done := true
       □¬y_done; y?save →
          y_done := true
          . (for each
          . neighbour y
          . except x)
       ]
    .
    . (for each neighbour x)
    .
  □x!terminated → skip
    .
    . (for each neighbour x)
    .
  □ANALYSER?stop → abort]
```

which represents the behaviour of the process after the basic computation has terminated. After the process has informed the analyser of its demise it must still be ready to participate in a wave (rule S5). It must also be prepared to inform other processes that it has terminated if they need to know this. Finally rule S7 is implemented by getting the analyser to send a 'stop' signal to all processes, only when they have all ended their basic computation. For simplicity we have used an abort command although the same effect could be achieved with a boolean variable. Note that in this condition a process can respond to, but not initiate, a wave.

Finally, the analyser process must support the following functions:

a. It must be ready to accept a unique guard identifier from a process at any time, which is then appended to the current trace segment for this process.

b. It must be ready to accept a local state from any process at any time. This new state is used to start a new trace segment for this process, then to check how many *complete* global states now exist, and if this is over some pre-defined constant (typically 2) the oldest trace segments must be discarded, and space reclaimed (S8). A second state will not be accepted from this process until the current wave is complete (S6).

c. It must be prepared to accept a 'terminated' signal from each process. When all processes have sent a terminated signal to the analyser, the analyser must send a 'stop' signal to each basic process and itself terminate (S7).

For small examples this tracing code clearly outweighs the original program! Obviously preprocessing a program to add the tracing code is not a practical approach. These functions should be built-in to the CSP implementation.

4.5.2 Replaying a trace

To implement the 'replaying version' of the program:

a. replace the initialisation part of each process with

LOAD?**state**;
LOAD?guard_id;

where **state** contains initial values for all local variables and the PC (rule R2), and 'guard_id' contains the value of the first guard in the trace (if any),

b. add the following condition to each basic guard (R1′)

(guard_id = g ∨ guard_id = 0)

c. add

LOAD?guard_id

to the start of each basic guarded command list to advance the trace whenever a guard succeeds,

d. add the special LOAD process which concatenates each of the stored trace segments for each process, sends the first state to each process, and then sends each guard identifier in the trace to each process requesting one. The special value 0 is always sent after the end of the trace has been reached. Any further states encountered are skipped.

Termination is not a problem here. The replayed system terminates normally at the same time the original basic computation did, and the LOAD process terminates when all basic processes do.

5. CONCLUSION

We have presented a method of storing and replaying traces of control paths for non-deterministic, concurrent CSP programs as a means of controlling non-determinism during testing. This work is similar in aim to that of Smith (1985), who uses traces of inter-process activity as a script for future tests.

Some practical problems, outside the domain of 'pure' CSP, but still important in actual use of such a scheme, are also discussed. Bougé's repeated snapshot algorithm has been adapted to solve the problem of putting an upper bound on trace length.

At the time of writing the unbounded length tracing mechanism has been used for some time in our local CSP simulator. The bounded length version has been tested at the source code level by manually rewriting CSP programs with the above extensions, but remains to be incorporated into the CSP implementation.

Finally, we note that a significant optimisation is possible by identifying alternative and repetitive commands whose guards are mutually exclusive so that the choice is always deterministic; these guards need not be traced.

Acknowledgements

I wish to thank all those who patiently sat through a draft presentation of this paper and provided feedback, especially R. Stanton, M. Newey and C. Johnson. Special thanks to R. Edmondson for prompting me to put pen to paper in the first place.

References

APT, K. R. (1985): 'Correctness Proofs of Distributed Termination Algorithms', in *Logics and Models of Concurrent Systems*, K. R. Apt (ed.), Springer-Verlag.

APT, K. R. and FRANCEZ, N. (1984): Modelling the Distributed Termination Convention of CSP, *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 3, pp. 370-379.

BAIARDI, F., DE FRANCESCO, N. and VAGLINI, G. (1986): Development of a Debugger for a Concurrent Language, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pp. 547-553.

BOUGE, L. (1985): 'Repeated Synchronous Snapshots and their Implementation in CSP', in *Automata, Languages and Programming, 12th Colloquium*, Lecture Notes in Computer Science, Vol. 194, W. Brauer (ed.), Springer-Verlag.

BUCKLEY, G. N. and SILBERSCHATZ, A. (1983): An Effective Implementation of the Generalised Input-Output Construct of CSP, *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, pp. 223-235.

FRANCIS, R. and MATHIESON, I. (1986): 'Language Tools for Exploring Concurrency', in *Proc. Ninth Australian Computer Science Conference*, Australian National University, Canberra.

GAIT, J. (1985): A Debugger for Concurrent Programs, *Software — Practice and Experience*, Vol. 15, No. 6, pp. 539-554.

GAIT, J. (1986): A Probe Effect in Concurrent Programs, *Software — Practice and Experience*, Vol. 16, No. 3, pp. 225-233.

GUPTA, N. K. and SEVIORA, R. E. (1984): 'An Expert System Approach to Real Time System Debugging', in *Proceedings of the First Conference on Artificial Intelligence Applications*, Denver, IEEE.

HOARE, C. A. R. (1978): Communicating Sequential Processes, *Commun. ACM*, Vol. 21, No. 8, pp. 666-677.

HOARE, C. A. R. (1979): *A Model for Communicating Sequential Processes*, The University of Wollongong, Department of Computing Science, Preprint 80-1.

INMOS LIMITED (1984): *occam Programming Manual*, Prentice-Hall.

OLDEROG, E-R. and HOARE, C. A. R. (1986): Specification-Oriented Semantics for Communicating Processes, *Acta Informatica*, Vol. 23, pp. 9-66.

ROPER, T. J. and BARTER, C. J. (1981): A Communicating Sequential Process Language and Implementation, *Software — Practice and Experience*, Vol. 11, pp. 1215-1234.

SMITH, E. T. (1985): A Debugger for Message-Based Processes, *Software — Practice and Experience*, Vol. 15, No. 11, pp. 1073-1086.

Biographical Note

Colin Fidge received his Masters Degree in Computer Science from the Royal Melbourne Institute of Technology in 1984. From 1983 to 1984, Mr. Fidge worked for the Telecom Australia Research Laboratories, Software Engineering Research Section. He is presently working towards a Ph.D. degree at the Australian National University, researching debugging and testing environments for concurrent languages.