J M Bishop
10/9/91.

South African Computer Journal
Number 5
September 1991

Suid-Afrikaanse Rekenaar-tydskrif
Nommer 5
September 1991

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal

An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif

'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083

Production Editor
Prof G de V Smit
Department of Computer Science
University of Cape Town
Rondebosch 7700
Email: gds@cs.uct.ac.za

Editorial Board

Professor Gerhard Barth
Director: German AI Research Institute
Postfach 2080
D-6750 Kaiserslautern
West Germany

Professor Judy Bishop
Department of Computer Science
University of the Witwatersrand
Private Bag 3
WITS 2050

Professor Donald Cowan
Department of Computing and Communications
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Professor Jürg Gutknecht
Institut für Computersysteme
ETH
CH-8092 Zürich
Switzerland

Professor Pieter Kritzinger
Department of Computer Science
University of Cape Town
Rondebosch 7700

Professor F H Lochovsky
Computer Systems Research Institute
University of Toronto
Sanford Fleming Building
10 King's College Road
Toronto, Ontario M5S 1A4
Canada

Professor Stephen R Schach
Computer Science Department
Vanderbilt University
Box 70, Station B
Nashville, Tennessee 37235
USA

Professor Basie von Solms
Departement Rekenaarwetenskap
Rand Afrikaanse Universiteit
P.O. Box 524
Auckland Park 0010

Subscriptions

	Annual	Single copy
Southern Africa:	R32-00	R12-00
Elsewhere:	$32-00	$12-00

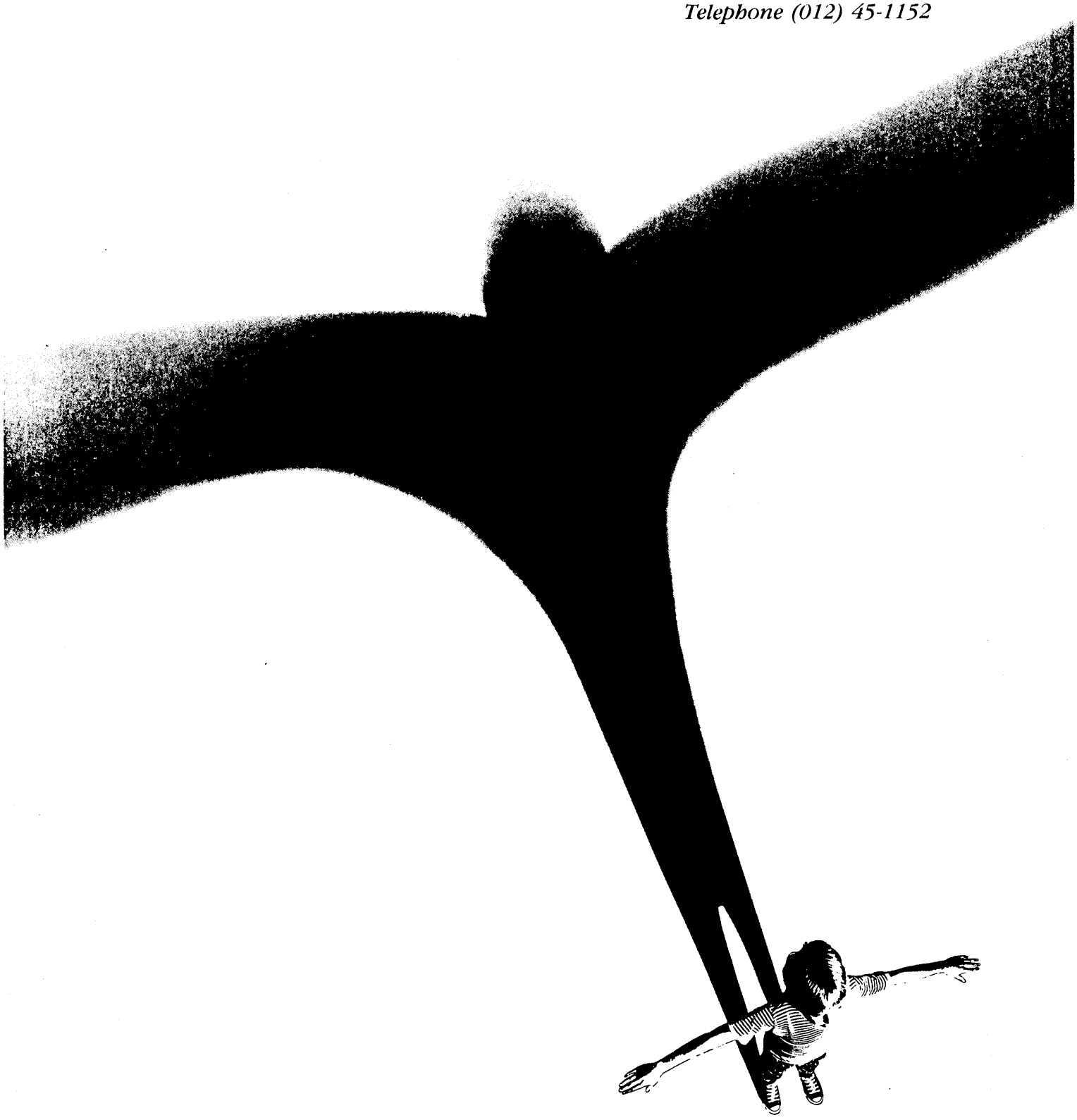
to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: (012) 420-2843 Fax: (int)+(012)+43-6454
Sigma Press

Regardless of the size of your computer system it is a strategic tool. We are committed to helping you use it most effectively. All it needs is a little imagination.

UNIDATA

Telephone (012) 45-1152



THERE'S A GIANT IN EVERY ONE OF US

YOUNG & RUBICAM 108113

UNIVERSITY OF PRETORIA
COMPUTER SCIENCE
MSc. BURSARIES
FOR STUDENTS FROM AFRICA

The Department of Computer Science at Pretoria University is inviting applications for two annually awarded bursaries for MSc study. These bursaries are aimed at outstanding students who have a Computer Science Honours degree (or equivalent qualification). As far as possible, one such bursary will be awarded to a South African student, while the other will be awarded to a student from another African state. The hope is to thereby promote interaction between South African computer scientists and computer scientists elsewhere in Africa.

The bursaries are funded by the Department's contract research work and will amount to a minimum of R20000 per year. They are intended to cover costs for tuition, housing, transport, books, and an adequate allowance for other living expenses. Successful candidates will be required to work for 20 hours per week as part of the Computer Science Department's contract research team. The work assigned to each candidate will be aimed at providing him with practical experience in his field of interest.

Study fields for which bursaries will be considered include:

- Computer Networking
- Distributed Systems
- Software Engineering
- Artificial Intelligence
- Intelligent Computer Aided Instruction
- Machine Aided Translation

Students should aim to complete the dissertation that is required for an MSc within two years of study (i.e. one year of full time study). The dissertation should preferably be written in English but it may also be written in Afrikaans.

The continuation of the bursary and adjustments to the allowance will be decided upon annually and will depend on progress. It is a condition of the bursary that, after completing his/her studies, the student should return to his/her country of origin for the same number of years for which the bursary was held.

Applicants may request application forms from:

MSc. Bursaries
The Computer Science Department
University of Pretoria
Hillcrest
Pretoria
0083
South Africa

Editor's Notes

It is with sincere gratitude that SACJ takes leave of Dr Peter Lay who, until recently, was the assistant editor dealing with Information Systems. He has left academia for what sounds like a more gentle lifestyle. (He has gone farming!) Under Peter's stewardship the number of high-quality IS papers in SACJ grew steadily. In general, IS papers tend to be accessible and relevant to a wide spectrum of computer professionals, and the quality of IS papers that have been appearing in SACJ has significantly contributed to the increased interest being shown in the journal by the local computer industry. If this growth in interest is to be sustained, it is urgent and important to find a suitable replacement assistant editor. The ideal candidate should not only be respected as an academic by his peers, but should also be disposed to enthusiastically promote SACJ in the private sector. Since a shortlist of candidates is currently being compiled, I would like issue a general appeal for names that might be included on it. Please contact me urgently if you would like to be considered for the job, or if you would like to nominate someone that you consider to be particularly suitable.

My three year term of office as editor expires in October. I have always considered it a great privilege to hold this position, and as a result, I felt honoured when the SAICS executive committee requested that I stay on for a further term. Nevertheless, I initially declined the request on the grounds that the time-demands of the job were significantly eroding my ability to fulfil other duties. Particularly demanding has been the task of seeing to the typesetting of the various contributions - either by doing it myself, or by ensuring that it is adequately done by someone else. Recently, however, Prof G de V Smit (Riël Smit) at UCT has offered to assume the role of production editor. This generous offer so much changes the complexion of what is being asked of me that I am now both willing and honoured to continue as editor for another term. I am very grateful to Riël for his offer and I look forward to working with him. In future, authors whose papers have been accepted for publication will be asked to liaise directly with him regarding the precise form in which the final contribution should be submitted.

The next issue of SACJ will consist largely of a selection of papers that were presented at the 6th South African Computer symposium. The selection will be based on comments from the referees who, at the time, were asked to adjudicate the papers in terms of their appropriateness for both the conference as well as for SACJ publication. Papers which, in the opinion of one or more referees, required major revision will have to be resubmitted to SACJ for refereeing purposes. Authors will soon be contact in this regard.

At the time of writing, the updated list of "approved" publications for the first half of 1991 had not yet been released by the relevant authorities. For the sake of past, present and future contributors I sincerely hope that SACJ will be on the list when it eventually comes out. However, I have become increasingly aware that there is a real danger of laying too much store on papers published in so-called approved journals as a basis for evaluating and rewarding research. I hope to expand more fully on this theme in a future edition of SACJ. Keep watching this space!

Derrick Kourie
Editor

This SACJ issue is sponsored by
a generous donation from
UNIDATA

A Linda Solution to the Evolving Philosophers Problem

S.E. Hazelhurst

Department of Computer Science, University of the Witwatersrand, Johannesburg, Private Bag 3, 2050 Wits

Abstract

The dining philosophers problem and the evolving philosophers problems are abstractions of resource sharing problems in parallel and distributed systems. A Linda solution to the dining problem has already been shown; this solution is not fair, and it couples the processes in the system together. The solution proposed here remedies some of the defects of this solution, and extends it to deal with the evolving problem. By comparing these solutions, and by comparing the proposed solution to the evolving problem with a solution to the problem in another language, the strengths of Linda are found, and areas for research identified.

Keywords: *change management, Linda, evolving philosophers problem, dining philosophers problem*

CR Categories: *D.1.3, D.4.1, D.4.7*

Received January 1990, Accepted January 1991

1. Introduction

The dining philosophers problem [Chandy and Misra 1984], one of the classic problems in computer science, is often used to illustrate the elegance or power of a programming paradigm (for example see [Carriero and Gelernter 1989; Ringwood 1988]). Its importance comes from the fact that the dining philosophers problem and its variants can be seen as abstractions of many real problems in distributed and parallel programming. Kramer *et al.* [1988] introduced the evolving philosophers problem – a variant – to explore the power of their particular language in coping with change management.

Carriero and Gelernter present a Linda solution to the dining problem. Their method has two drawbacks. First, it is unfair, some philosophers may starve. Second, the forks are labelled by numbers. This is unfortunate in that it restricts the generality of their solution, and would also make coping with change management difficult. It is the objective of this paper to remedy these defects, and to propose a solution to the extended problem of the evolving philosophers.

This paper is organised as follows. The rest of this section describes the dining philosophers problem (dining problem) and the evolving philosophers problem (evolving problem). Section 2 gives a brief overview of Linda, and discusses the original Linda solution to the dining problem. Section 3 presents a proposed Linda solution to the dining problem (the proof of correctness is found in the appendix). Then, a solution to the extended problem is presented in section 4 (with proof of correctness also in the appendix). Section 5 compares the various Linda solutions, and the solution to the evolving problem proposed here with another solution to the evolving problem [Kramer *et al.* 1989]. This comparison allows a critique of Linda.

In the dining problem, a number of philosophers sit around a table. Each philosopher alternates between states of thinking, being hungry, and eating. Each philosopher shares a fork with a neighbour to her left, and a neighbour to her right. In order to eat, a philosopher must have two forks. Clearly, this is an abstraction of a system where a number of objects share resources, and classic problems of starvation and deadlock can occur.

The evolving problem is an extension of the problem. Here a philosopher may move from one place in the table to another. A philosopher may leave the table, and new philosophers may join the table. Thus we introduce to the problem the question of change management.

2. A brief overview of Linda

2.1 The Linda paradigm

Linda [Gelernter 1985; Ahuja *et al.* 1986] is a parallel programming paradigm. Processes in a system communicate with each other through *tuple space* (and only communicate with each other through tuple space). Tuple space is a global associative memory. While the implementation of the tuple space may be distributed, to the processes in the system the tuple space is a shared memory, thus any process in the system may communicate with any other process via tuple space no matter their physical locations.

Six Linda primitives which can be added to any computer language to make that language a Linda language. Using these primitives, a process can communicate with tuple space. The objects placed in tuple space are called tuples; these tuples may either be data (passive), or processes (active). A short explanation of the primitives is given below. There are a number of

fuller references [Ahuja *et al.* 1986; Carriero 1987; Carriero and Gelernter 1989; Gelernter 1988]:

- *out(t)*: the tuple *t* is placed into tuple space. *out* is non-blocking.
- *in(t)*: the process executing the primitive attempts to retrieve a tuple from tuple space. A tuple is retrieved from tuple space if the actual tuple matches the tuple template specified by the *in*. For example, the tuple `("A",3,4)` could be matched by the templates `("A",3,4)`, `("A",3,INTEGER y)`, `("A", INTEGER x, INTEGER y)`. In this example, *x* and *y* would be given the values 3 and 4. *in* is blocking, the process executing it waits until the match succeeds. A tuple matching an *in* request is removed from tuple space.
- *read(t)* is the same as *in* except that a tuple which is matched is not removed from the tuple space.
- *inp(t)* and *readp(t)* are the same as *in* and *read*, but do not block.
- *eval(proc)* places an active tuple into tuple space. This is a process which executes, and it can, in turn, place passive or active tuples into tuple space.

2.2 The dining philosophers problem: a simple Linda solution

Carriero and Gelernter [1989, p. 452] present the solution shown in figure 1.

```
phil(i)
    int i;
{while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i+1)%Num);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1)%Num);
    out("room ticket");
    }
}
```

Figure 1. Simple Linda solution

Tuple space is initialised by placing one fork (fork = chopstick) for each philosopher (there are *Num* philosophers in all), and *Num*–1 *roomtickets* in tuple space. The latter initialisation prevents deadlock, by ensuring there is at least one more fork than philosophers trying to eat.

The solution presented below differs in three major respects from this solution:

- the method of deadlock detection: the simple solution above uses *roomtickets* to prevent deadlock. Although a completely satisfactory solution here, it may not generalise since essentially it is a method which requires global knowledge. The method below solves the problem by each philosopher knowing only its own state, and some knowledge of its previous actions;
- the naming of the forks: the forks are labelled by numbers (which are the identities of the philosophers). By binding the name of the fork to the identities of the philosophers, and also giving them identities which are numbers, the generality of the solution is limited (more on this later); the method proposed below does not have this restriction;
- starvation: the simple solution does not guarantee that a philosopher will not starve; the method proposed below avoids starvation. While it is true that if the underlying Linda implementation is fair, then the above solution is also be fair, Linda semantics do not require a fair implementation (indeed, the implementation of Linda which the proposed solution below was tested is not fair).

3. A new Linda solution to the dining problem

The solution to the evolving problem is shown in figure 2 (the initialisation code) and figure 3 (the philosopher's code). If no reconfiguring occurs then this is a fair, non-deterministic solution to the dining problem. This section gives an informal description of the algorithm. Proof of correctness is deferred to the appendix. Table 1 provides a list of the tuples which are used by the program segments and a brief description of what they are used for; referring to the table may make the code easier to follow.

Table 1

Tuple Name	*Function*
booked	used by a philosopher to ask her neighbour for use of the fork
fork	a shared fork — the second field of the tuple is the name of the fork
info	provides a philosopher with the names of her forks, and tells her whether she has them to start with
instruction	gives a specified philosopher an instruction
leave	tells the specified philosopher that she should leave the table
newfork	when a philosopher has been given the 'newfork' instruction, this tuple is placed in tuple space to inform her of the name of the new fork
passive	placed by a philosopher into tuple space to indicate that she is passive and is waiting to receive an instruction
ready	placed by a philosopher into tuple space to indicate that she is ready to run again
reconfigure	used by the system to tell a philosopher to reconfigure
restart	informs a philosopher that she should start running again

First we assume an appropriate data structure maintaining a list of philosophers—keeping the current configuration. It is assumed here that there is a linked, circular list of type *philRecord* which has all the philosophers' details. Note this is a generalisation of having an array of philosophers represented by their indices in the array. Each philosopher knows her own name (which is a logical name, independent of physical location), and the names of the forks that she uses. In addition, the system knows who the philosophers' neighbours are.

The reasons for this initialisation are explained in the appendix. The first FOR loop places in tuple space alternately the philosophers whose neighbours have their forks, and those philosophers who actually have their forks. The second loop places in tuple space all philosophers who have one of their shared forks free.

This is a restriction on the generality of the solution in that it is assumed that the philosophers must start off in this way. This is not overly restrictive since it is only an initial ordering which does not confer lasting special privileges or disadvantages to philosophers with two or no forks. Other initialisations may well be possible: this was chosen to aid the proof of correctness.

The data structure for the philosophers' code is explained below:

- *first* and *second*: these variables indicate the order in which the forks are to be retrieved from the tuple space. Note that their value is *leftpartner* or *rightpartner*. This indicates the logical relationship between the neighbours only, and does not specify the identities of the neighbours.
- *forks*[*leftpartner*] contains the name of the fork which the philosopher shares with her left neighbour, *forks*[*rightpartner*] the name of the fork shared with the right neighbour.

Note that the information which a philosopher needs is: her identity, the identity of the forks she is using, and the fact that she is sharing them with a left partner and a right partner; she does *not* know the neighbours' identities.

When the program starts, the main program will create as many philosophers as necessary. Each philosopher is given (in a tuple) her identity, the names of her forks and told whether she initially has two forks or zero forks.

Each philosopher then repeatedly goes through the following phases: thinking, getting two forks, and eating. How long a philosopher eats or thinks for is non-deterministic (but finite). Since getting forks may rely on how long neighbouring philosophers eat or think for, getting two forks may also take a non-deterministic (but finite) time.

```
 partnerType   = [leftpartner..rightpartner];
 forkArray     = ARRAY partnerType OF resourceType;
detailType    = RECORD
                  name       : identities;
                  forks      : forkArray;
                END;
philPtr        = POINTER TO philRecord;

philRecord    = RECORD
   info        : detailType;
   left,
   right       : philPtr;
END;

thisphil := firstphil;
(* assume numphil philosophers, j with 2 forks, j with none *)
FOR k:=1 TO j*2-1 BY 2 DO
   eval(Philosopher);
   out("info", thisphil^.info, TRUE);
   thisphil := thisphil^.right;
   eval(Philosopher);
   out("info", thisphil^.info, FALSE);
   thisphil := thisphil^.right;
END;
FOR k:=j*2+1 TO numphils DO
   eval(Philosopher);
   WITH thisphil^ DO
      out("info", info, FALSE);
      out("fork", info.forks[leftpartner]);
   END;
   thisphil := thisphil^.right;
END;
```

Figure 2. Initialisation code: The solution was tested on a Modula-2 Linda version. Extracts of the code are shown..

Let us first examine the behaviour of a philosopher who has both forks. The boolean variable *haveforks* indicates whether a philosopher does have both forks.

Once the philosopher has finished thinking, she becomes hungry and wants to eat. To eat she must obtain both forks. This is trivial since *haveforks* is true. She can then eat for however long she wants to (as long as it is for a finite time). Once she has finished eating, she checks tuple space to see whether either of her partners wants to use one of the forks. If either does, then she gives both forks up. An important thing to note is that she records whose request caused her to give up the forks (of course both partners could have requested the forks, but it is the request which caused the action which is important).

Now, consider the behaviour of a philosopher who does not have the forks. When she has finished thinking, she tries to get the forks. As *haveforks* is false, this is not trivial. For each fork she goes through the following steps.

- first she checks to see whether any other philosopher has booked the fork. This occurs when, after giving up the fork to a neighbour who has requested it, the neighbour responds so slowly that she manages to finish thinking and become hungry again before the neighbour has actually retrieved the fork. To prevent a hungry, fast-thinking philosopher from giving up her forks and then snatching them back before one of her neighbours can get it (because of the non-determinism of the `in` statement), this check must be put in, although it will not be needed very often.
- having ensured that no-one else is trying to get the fork, she makes an attempt to retrieve the fork from the tuple space.
- should the fork not be in tuple space, she then places a request for the fork in tuple space, and waits for the neighbour to give it up.

The order in which a hungry philosopher without forks tries to retrieve forks is very important. A naïve approach would be for each philosopher to grab her left (or right) fork first each time. This could easily lead to a situation of deadlock if these requests all succeed, since the next step would be to retrieve the other fork which her neighbour has. This is a classic deadlock situation.

Consider how the danger arises. Suppose that philosopher *A* has two neighbours *Z* to her left, and *B* to her right. Look what could happen if *A* has two forks, and *B* requests one of them. Then, *A* would give up both forks. If *A* is a quick-thinker, she might become hungry again before *Z* becomes hungry and take their shared fork. So, *A* could retrieve the fork shared between *Z* and *A*, and then try to take the fork shared between *A* and *B*. But, there is no guarantee that *B* is also not waiting for a fork from *her* right and so on. Then, when *Z* attempts to retrieve her right fork, she will find that *A* has it, and won't give it up because she is waiting for the fork from *B* which will never come.

By making *A* first get the fork from the neighbour which took it from her last, the algorithm shown avoids the deadlock. *A*'s other neighbour cannot be starved in the same way. Thus the key issue here is to be able to alternate the order in which the forks are retrieved.

Note also that the algorithm is non-deterministic. When *A* finishes eating, she keeps the forks if there isn't a request from one of her neighbours. This means that slow thinkers will not hold everyone else up.

On the other hand, the solution is fair in that no philosopher will be starved. That is, once a philosopher finishes thinking and becomes hungry, she will be able to eat in a finite time. This also ensures that deadlock cannot occur.

A difficulty with the algorithm though is that slow-thinkers may hold up neighbouring philosophers. This is not an insurmountable problem. The advantage of not giving up the forks when finishing eating is that some unnecessary communication is avoided — only when necessary will a fork be given up. The approach of Chandy and Misra [1984] is essentially the same, except that in their solution a request interrupts a philosopher, whereas here the request is mediated through tuple space.

The protocol outlined below can be used to avoid the problem. This protocol has been implemented and tested. When a philosopher is hungry, she attempts to pick up her forks, first the left and then the right. She checks to see whether there is a *booked* tuple to prevent deadlock problems. If she gets them she can eat. Otherwise she must use the *GetFork* procedures to get them. If she manages to pick up the left fork but not the right fork, then she *out*s the left fork. In this case she will do a *GetFork* on her right fork before trying to get the left fork, otherwise she will get her left fork before the right one. This procedure avoids any deadlock or starvation.

The problem with this is that it is more complex, and there is more communication. Which is the better approach clearly depends on the assumptions made including the expense of communication, and length and variance of the thinking times of the philosophers. Two other approaches to solving this problem are outlined in the last part of the appendix.

```
PROCEDURE GetFork(id:identities; fork:resourceType; haveforks: BOOLEAN);
  BEGIN
    IF NOT haveforks THEN
       WHILE rdp("booked", fork) DO;
       IF NOT inp("fork", fork)  THEN
         out("booked", fork);
         in("fork", fork);
         in("booked", fork);
       END;
    END;
  END GetFork;

PROCEDURE Philosopher;
  VAR  first, second :  partnerType;
       myinfo        :  detailType;
       haveforks,halt:  BOOLEAN;
  BEGIN
      in("info", VAR myinfo, VAR haveforks);
     WITH myinfo DO
        first := leftpartner;  second := rightpartner;
       LOOP
          thinking(name);
          CheckReconfigureSystem(myinfo, halt, haveforks);
          IF halt THEN EXIT; END;
          GetFork(name,forks[first], haveforks);
          GetFork(name,forks[second], haveforks);
          haveforks := TRUE;
          eating(name);
          IF rdp("booked", forks[leftpartner]) THEN
            first := leftpartner;     second := rightpartner;
            out("fork", forks[leftpartner]);
            out("fork", forks[rightpartner]);
           haveforks := FALSE;
          ELSIF rdp("booked", forks[rightpartner]) THEN
              first := rightpartner;     second := leftpartner;
              out("fork", forks[leftpartner]);
              out("fork", forks[rightpartner]);
             haveforks := FALSE;
        END;
       END; (*loop*)
     END; (*with*)
   END Philosopher;
```

Figure 3. New Linda solution

4. The evolving problem

Kramer *et al.* [1988] describe three types of changes. A philosopher can leave the table, a new philosopher can join the table, or a philosopher can move from one part of the table to another. The solution (the code for which is shown as figure 4) also deals with a fourth change – one of the forks which a philosopher uses being changed. (I argue that it deals with a useful abstraction, that of a process changing the resource that it uses).

A key point to change management is that the specification of changes (which is specified at the system level) should not require a knowledge of the state of each process in the system. A process receives an instruction of what changes are required; the process must decide how this is to be implemented depending on its state. It is also important that changes leave the system in a consistent state [Kramer *et al.* 1989].

An informal description of the code is given below. Proof of correctness is shown in the second part of the appendix. Figure 5 shows the code which the system uses to make changes, and figure 4 shows the code which is added to each philosopher.

The first step for all changes is to instruct the philosophers concerned to reconfigure. Before issuing further instructions, a message must be received from each philosopher concerned with the change saying that they are all in passive states. What this means is that they do not hold resources which may create deadlocks, and also that when they restart the system will be able to attain a consistent state.

Once acknowledgements have been received, the appropriate instructions can be issued. For renaming a fork, it means instructing the two philosophers that share a fork that the name of the fork has been changed. Adding a new philosopher means *eval*ing a new philosopher with the appropriate names of forks, and informing the neighbouring philosophers that the name

of one of their forks has changed. To delete a philosopher, the *leave* instruction is issued, and informing the neighbouring philosophers of the change of names of forks. In all cases, the system would have to update the data structure (not shown here). A null *startagain* instruction is used for philosophers who just need to be made passive while changes are being made to their neighbours.

The one case not dealt with is the case of a philosopher moving from one part of the table to another. This can be done by a deletion and an insertion (although optimisations may be possible). The code in figure 5 must be added to allow each philosopher to react to system reconfiguration instructions.

The first thing which a philosopher does when informed that she must reconfigure is to check whether she has her two forks or not. If she does, she must place them into tuple space. This prevents deadlocks from occurring. She can then acknowledge the reconfiguration command by placing the *passive* tuple into tuple space. The next step is to see what type of reconfiguration is required.

The simple case is if the philosopher is being told to leave the table. In this case she simply removes her left fork (to ensure that the number of philosophers and forks is the same) and terminates.

The more complicated case is if the name of one of the philosopher's forks is changing. In this case she removes from tuple space a tuple saying whether it is the fork which is shared with her left or right partner that is changing, and the name of the new fork. If it is her left fork being changed, then she removes her old fork from the tuple space and replaces it with the new fork (thus philosophers have a special responsibility for their left forks). She is also informed whether she should take possession of the two forks or not. In most cases she does not take possession of the forks, and must contend for her forks. She updates her data structures, and indicates that she is ready to be active again, and waits for the system to tell her that she can restart.

```
PROCEDURE CheckReconfigureSystem(VAR myinfo:detailType;
                                 VAR halt, haveforks:BOOLEAN);
    VAR  instruction : instructions;
    BEGIN
       halt := FALSE;
       WITH myinfo DO
          IF NOT inp("reconfigure", name) THEN RETURN; END;
          IF haveforks THEN
             out("fork", forks[leftpartner]);
             out("fork", forks[rightpartner]);
             haveforks := FALSE;
          END;
          out("passive", name);
          in("instruction", name, VAR instruction);
          CASE instruction OF
              startagain : ;
             |leave   :
                 in("fork", forks[leftpartner]);
                 halt := TRUE;
             |newfork :
                 in("newfork",name,VAR partner,VAR forkID,VAR haveforks);
                 IF partner = leftpartner THEN
                    in("fork", forks[partner]);
                    out("fork", forkID);
                 END;
                 IF haveforks THEN
                    in("fork", forks[leftpartner]);
                    out("fork", forks[rightpartner]);
                 END;
                 forks[partner] := forkID;
          END;
          out("ready", name); in("restart", name);
       END;
    END CheckReconfigureSystem;
```

Figure 4. Reconfiguration code

To remove a philosopher—

```
(* thePhil -- the philosopher to be removed;      *
 * phileft  - the philosopher to her left;        *
 * philright- the philosopher to thePhil's right *)
WITH thePhil^ DO WITH thePhil^.info DO
 out("reconfigure", name);      out("reconfigure", phileft.name);
 out("reconfigure", philright.name);
 in("passive", name);     in("passive", phileft.name);
 in("passive", philright.name);
 sharedfork := forks[rightpartner];
 out("instruction", phileft.name, newfork);
 IF left^.left = right THEN  (* only two philosophers left *)
    out("newfork",phileft.name, rightpartner,sharedfork, TRUE);
 ELSE
    out("newfork",phileft.name, rightpartner, sharedfork, FALSE);
 END;
 out("instruction", name, leave);
 out("instruction", philright.name,startagain);
 (* update data structures -- not shown *)
 in("ready", name);
 in("ready", philright.name); in("ready", phileft.name);
 out("restart", name);
 out("restart", phileft.name);out("restart", philright.name);
```

To add a philosopher to the right of thePhil

```
(* Insert 'newID' between 'thisID' and 'oldright'        *
* thisID-- thePhil's name                                *
* newLfork- the name of the new phil's fork              *)
out("reconfigure",thisID);   out("reconfigure",oldright);
in("passive", thisID);       in("passive", oldright);
newRfork := thePhil^.info.forks[rightpartner];
NEW(newphil);
WITH newphil^ DO WITH newphil^.info DO
   name := newID;
   forks[leftpartner] := newLfork;
   forks[rightpartner]:= newRfork;
   eval(Philosopher);
   out("info", newphil^.info, FALSE);
   (* pointer operations to update data structure -- not shown *)
END; END;
out("instruction", thisID, newfork);
out("newfork", thisID, rightpartner, newLfork, FALSE);
out("instruction", oldright, startagain);
in("ready", oldright); out("restart", oldright);
in("ready", thisID);    out("restart", thisID);
out("fork", newLfork);
```

To change the name of a fork

```
myname := thisPhil^.info.name;
leftname := thisPhil^.left^.info.name;
WITH thisPhil^ DO
   out("reconfigure", myname);   out("reconfigure", leftname);
   in("passive", myname);        in("passive",  leftname);
   out("instruction", myname, newfork);
   out("newfork", myname, leftpartner, newID, FALSE);
   out("instruction", leftname, newfork);
   out("newfork", leftname, rightpartner, newID, FALSE);
   info.forks[leftpartner] := newID;
   left^.info.forks[rightpartner] := newID;
   in("ready", myname);      in("ready", leftname);
   out("restart", myname);   out("restart", leftname);
```

Figure 5. System code

5. Conclusion

This section of the paper discusses the proposed solution by comparing it with the simple Linda solution and the Conic solution [Kramer *et al.* 1989]. This illustrates the strengths and weaknesses of Linda.

A full comparison with the Conic solution is not possible, since the code for the Conic solution has not been published. The importance of the comparison is that Linda shares a key feature with Conic, and the solution proposed here has its origins in the Conic solution. Importantly too is that it was designed to address some of the criteria for change management.

5.1 Solutions to the dining problem

The solution proposed here is an advance over that of Carriero and Gelernter in that it is fair (avoids starvation), and that the names of the forks are completely decoupled from that of the philosophers. Each philosopher knows the names of the forks that it uses, and the fact that it shares the fork with a left neighbour and a right neighbour, but does not know the identity of those neighbours. Another improvement is (arguably) that the mechanism which is used to avoid deadlock does not require knowledge of global state. Rather each process (philosopher) knows only its own state, and has some limited information about its previous actions. It does have a limitation that slow-thinking philosophers could hold up their neighbours. However, this can be avoided at the cost of extra communication.

5.2 Reconfiguration

Essentially, the evolving problem describes change management: where the different processes which make up a system change their configuration (which may reflect changes in either the physical or logical make up of the system). Change management is difficult as it requires correct cooperation between the different components of the system. An important question then is how the system can be structured so as to reduce the complexity of change management.

Linda and Conic share the property of decoupling (since any process could take a tuple). In the Conic system [Magee *et al.* 1989; Sloman and Kramer 1987], the modules communicate with each other via named ports. Each module communicates with its port. It does not know which module is on the other side of the port. At the system level, the configuration of the modules is specified by stating which ports are linked to which others.

In Linda there is a higher degree of decoupling. Processes communicate through tuple space only. In this example, the philosopher processes have to retrieve named resources from tuple space. It is not necessary for them to have knowledge of which other processes share those resources.

This decoupling simplifies the reconfiguration. The system is protected from the workings of individual processes, and the processes are protected from the workings of the system and other processes.

The key step in reconfiguring both in Conic and in the Linda solution proposed here is to bring those processes affected by change into a quiescent state, where they cannot cause deadlock. Changes can then be made, and the system brought into a consistent state.

The reconfiguration proposed here largely meets the following objectives of change management suggested by Kramer *et al.* [1989]:

- changes must be specified at the system level, and thereby be independent of the state of the processes or the ways they are structured;
- what changes are necessary are specified at the system level, how the changes are to be made are the responsibility of the processes — the concerns of the two levels are clearly separated;
- changes must leave the system in a consistent state;
- limiting the effects of changing: only processes directly affected by the changes (and occasionally one of their neighbours) need be stopped while changes take place; other processes can continue normally (this is an important quality in real-time distributed systems).

One extra advantage that Linda has over Conic due to the extra level of physical decoupling, is that the solution of the evolving problem is simplified in distributed systems. Moving a philosopher from one machine to another does not need reconfiguration because of the physical transparency which Linda has. However it is not clear how machine dependencies are specified — this is discussed in the next sub-section.

The Linda solution presented above is not as good as the Conic solution in two respects. The first is that Conic has developed a specialised syntax to perform change management. For example, making a component passive requires one instruction in Conic and two in the solution presented here. Although not a problem in principle, it makes change management slightly more error prone. The second problem is that the solution places greater responsibility on the system for change management. While this makes change management more flexible, it also means that the change management protocol is more sensitive to the communication protocol between the different components. In view of the difficulty of change management, this may be a significant limitation of the Linda solution proposed here.

5.3 Problems with Linda

This work in this paper identifies two problems with Linda caused by the fact that the semantics of Linda have not properly been formalised.

The first problem is that the proof of correctness is not easy. The proof presented in the appendix is informal and contains an unfortunately high level of anthropomorphisms. A more formal proof (without having the proper calculus for specifying the behaviour of the Linda primitives) would have been at least twice

the length and difficulty. This is an aspect which needs further work (and is getting it).

The second problem relates to how machine dependencies are specified, and concern what the semantics of *eval* are. As Carriero [1987] says: "There are significant unanswered questions about *eval*...important questions arise over dynamic process creation and management".

One issue in particular arose in the development of the solution. What is the scope of the code which is *eval*ed? For example, can a philosopher see the data structure of the main program? A more complicated case arises when a philosopher is given a parameter. What are the semantics of the parameter passing? If the philosopher changes the formal parameter, does the actual parameter get updated too? The answers to these questions are not easy, and seem to be done by implementation *fiat* rather than any underlying philosophy. For more discussion on this and other problems with the semantics of *eval*, see Leichter's PhD dissertation [Leichter 1989].

Further, there seems to be no way of specifying machine dependencies. In distributed and real-time systems, this will probably be necessary.

5.4 Summary

This paper has explained what the evolving problem is. It then briefly described the Linda primitives, and a simple solution to the dining problem. A better solution was presented, as well as a solution to the evolving problems. The comparison between the two Linda solutions, and a discussion of a Conic solution were used to critique Linda.

6. Appendix: Correctness proofs

Proof of dining problem

The proof of correctness for the algorithm, disregarding the reconfiguring, follows from the fact that when a philosopher becomes hungry, she will be able to retrieve the two forks from tuple space in a finite amount of time. This is proven by induction: if we have a ring of n philosophers, then in finite time, if philosopher A becomes hungry, she will be able to gain both her forks and eat.

- The basis step is with two philosophers. The algorithm works here, as when a philosopher becomes hungry, if she does not already have both her forks, she can place a booking tuple into tuple space. Eventually her partner will finish eating, examine tuple space for a booking tuple and give up both forks.
- Induction hypothesis: with n philosophers, $A...Y$, then if A becomes hungry she will in finite time eat.
- The induction step is to show that if there are $n+1$ philosophers $A...Z$, then if A becomes hungry, she will eat in finite time. (Notation: the fork shared between philosophers C and D is referred to as *cd*.)

1. Now suppose A does not have her forks, and suppose that her first fork is *az* (the case of the first fork being *ab* is symmetric). This implies that when A last had two forks she gave them up because of a request which emanated from Z. Now, if Z has two forks, Z will finish eating in a finite time and so give up both forks, which will mean that A will get the *az* fork. There are two reasons why Z could give up the forks. The one is because she responds to a request from Y, and the other is that she could respond to a request from A (of course, both could have asked for the forks, but it is the request which Z responds to which matters).
2. If Z responds to a request from A, then she will wait for A to release the fork *az* before attempting to get the *yz* fork. Thus the *yz* fork will be available for Y at least until A has eaten. And Z has no forks. Thus, the question whether A can eat in the ring of philosophers $A,...,Y, Z$ reduces to the question whether A can eat in the ring of philosophers $A,...,Y$: the answer to which we know is "yes" from the induction hypothesis.

3a. The other possibility is that Z responds to a request from Y when giving up the fork. Note that this means that Z cannot obtain the *yz* fork until after Y has eaten, and thus will not attempt to take the *az* fork until after that (so Z does not have the *az* fork). Y will be able to eat at some stage, since if she couldn't it would mean that we would have a chain where X has the *xy* fork and was waiting for a fork from $W,...,B$ has the *bc* fork and is waiting for a fork from A, A has the *ab* fork and is waiting for a fork from Z. But this can't happen since we know that Z doesn't have the *az* fork.

3b. By a similar argument to paragraph (2) above, if Y gives up her forks (after eating) in response to a request from Z then A will be able to eat since we now have to consider the question whether A can eat in the ring of philosophers A to X which we know from the induction hypothesis.

3c. If Y gives up her fork to a request from X, then by repeating steps (3), we will finally get the answer. The last time it will be possible to repeat the step is if B gives up her forks in response to a request from A. But we know that A already has the *az* fork, she will now be able to get the *ab* fork and eat.

- We have shown that if the system is not in a deadlock siutation, then starvation cannot occur. The final thing to be shown is that tuple space is initialised correctly (not initially in deadlock). Suppose there are to be n philosophers in the tuple space, with identities x_1 to x_n. Initially let there be j philosophers with two forks (i.e. with *haveforks* set true), $1 \leq j \leq \lfloor n/2 \rfloor$. Let these philosophers be philosophers $x_1, x_3,..., x_{2j-1}$ (so when their code is `eval`ed, they have *haveforks* set to true). Then philosophers $x_2,..., x_{2j}$ be initialised so that they have *haveforks* set to false when they are `eval`ed. For philosophers x_{2j+1} ... x_n `eval` their

code so that they have *haveforks* set to true. Place in tuple space the identity tuples *forks* so that each philosopher knows the identities of her forks. For the philosophers with one fork only place in tuple space the tuple representing the fork which the philosopher shares with her left partner (i.e. if $k > n-2j$, perform an `out(fork)`, where *fork* has the same value as the *k*-th philosopher's *forks*[*leftpartner*]). In this initial state, there is clearly no deadlock since a number of the philosophers can eat immediately.

Evolving problem: proof of solution

The proof for this relies on the behaviour of the *CheckReconfigureSystem* procedure which handles change management. Those philosophers affected by the change will have given up their forks to the tuple space. Their state — to whom they last gave up their forks — remains the same.

All the other philosophers at the table will continue to be able to eat (since while the changing philosophers are passive their forks are available to any active philosopher). Note (see figure 3) that a philosopher cannot distinguish between a neighbour thinking (albeit for a long time) and changing.

Thus when the passive (and/or new) philosophers are restarted, the system is not in a deadlock situation. Thus, the proof above will hold. The exception to this is the case where a table of three philosophers is reduced to a table of two philosophers. To perform the removal of one of the philosophers, all three must be suspended. Thus the rest of the table is in deadlock, and so the proof fails. But, the removing code deals with this as a special case: when only two philosophers are left (when one philosopher's right partner is the same as her left partner), then one of the philosophers is told to pick up both forks. Thus when the philosophers restart, one will have both forks and the other none. The table is not in deadlock, so the proof holds.

Slow-thinkers holding-up the table

The code as presented does have the problem that a slow-thinking philosopher might hold up the entire table. Two more possible solutions are presented below. (Call the part of the code where the philosopher checks to see whether to give up her fork the *checking code*.)

Ideally, we would like the checking code to be executed at any time a request for a fork comes. This code could in fact be invoked at any time from the philosopher finishing eating to the philosopher finishing thinking. Better would be to have the checking code as an interrupt handler which can be invoked during certain sections of the loop. Thus, if an interrupt facility existed, this could be used. This solution is reasonably elegant, but it deviates from the principle that all communication takes place via tuple space. Conceptually it is very simple.

A second possibility, is to *eval* an active tuple with the checking code. When the philosopher wants to start eating again, an *in* is performed on this active tuple (using the ideas on the treatment of active tuples proposed recently by Gelernter [1989]). The difficulty with this is that the active tuple will need direct access to the philosopher's data structures, or there will have to be increased communication through tuple space.

References

[1] S Ahuja,N Carriero and D Gelernter,[1986], Linda and Friends, *IEEE Computer*, **19**(8), August, 26–34.

[2] N Carriero, [1987], Implementation of Tuple Space Machines, *Research Report YALEU/DCS/RR-567*, Computer Science Department, Yale University.

[3] N Carriero and D Gelernter, [1989], Linda in Context, *Comm. of the ACM*, **32**(4), April, 444–458.

[4] KM Chandy and J Misra, [1984], The Drinking Philosophers Problem, *ACM Trans. Prog. Lang. Syst.*, **6**(4), October, 632–646.

[5] D Gelernter, [1985], Generative Communication in Linda, *ACM ACM Trans. Prog. Lang. Syst.*, **7**(1), January, 80–112.

[6] D Gelernter, [1988], Getting the Job Done, *Byte*, **13**(12), November, 301–308.

[7] D Gelernter, [1989],"Multiple tuple spaces in Linda", *PARLE '89: Parallel Architectures and Languages Europe Volume II (Lecture Notes in Computer Science 366)* E. Odijk *et al.* Eds. Springer-Verlag, Berlin, 20–27.

[8] J Kramer and J Magee, [1988], A Model for Change Management, *Proceedings of the IEEE Workshop on Future Trends in Distributed Computing Systems in the 1990s* (Hong Kong, September). IEEE, Washington D.C.

[9] J Kramer, J Magee and A Young, [1989], A Refined Model of Change Management in Distributed Systems, *Proceedings of the 3rd Workshop on Large Grain Parallelism*, ACM, New York.

[10] JS Leichter, [1989], Shared Tuple Memories, Shared Memories, Buses and LANs — Linda Implementations Across the Spectrum of Connectivity, *Technical Report YALEU/DCS/TR-714*, Yale University, New Haven.

[11] J Magee, J Kramer, and M Sloman, [1989], Constructing Distributed Systems in Conic, *IEEE Trans. Software Engineering*, **15**(6), June, 663–675.

[12] GA Ringwood, [1988], Parlog86 and the dining logicians, *Comm. ACM*, **31**(1), January, 10–25.

[13] M Sloman and J Kramer, [1987], *Distributed Systems and Computer Networks*,Prentice-Hall International, London.

Acknowledgements

I thank Conrad Mueller and Sheila Rock who read previous drafts of the paper, and the referees for their very helpful and insightful comments and advice.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted **in triplicate** to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines.

- Use double-space typing on one side only of A4 paper, and provide wide margins.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.
- References should be listed at the end of the text **in alphabetic order** of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:

  [1] E Ashcroft and Z Manna, [1972], The translation of 'GOTO' programs to 'WHILE' programs, *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255.

  [2] C Bohm and G Jacopini, [1966], Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM*, **9**, 366-371.

  [3] S Ginsburg, [1966], *Mathematical theory of context free languages*, McGraw Hill, New York.

Manuscripts *accepted* for publication should comply with the above guidelines, and may be provided in one of the following formats:

- in a **typed form** (i.e. suitable for scanning);
- as an **ASCII file** on diskette; or
- as a **WordPerfect**, $\TeX$ or $\LaTeX$ or file; or
- in **camera-ready** format.

A page specification is available on request from the editor, for authors wishing to provide camera-ready copies. A styles file is available from the editor for Wordperfect, $\TeX$ or $\LaTeX$ documents.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

South African Computer Journal

Number 5, September 1991
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 5, September 1991
ISSN 1015-7999

Contents