A Tool for Proving Deadlock Freedom

J.M.R.Martin
Oxford University Computing Services,
13 Banbury Road,
Oxford 0X2 6NN, UK

S.A.Jassim
Department of Mathematics,
Statistics, and Computer Science,
University of Buckingham,
MK18 1EG, UK

Abstract. We describe a tool, programmed in Java, for the formal verification of the absence of deadlock and livelock in networks of CSP processes. The innovative techniques used scale well to very large networks, unlike the exhaustive state checking method employed by existing tools.

1 Introduction

As computer programs become increasingly vast and complex and are used for more and more safety-critical applications the use of formal mathematical methods in their development is becoming crucial. Lives may depend on it. However there are two important barriers to overcome. Firstly the large amount of work required in applying rigorous formal methods might seem infeasible. Secondly, computer programmers come from diverse backgrounds, and the level of mathematics involved will be off-putting to many, and also increase the chance of error. For these reasons various tools have been developed which automate the application of formal methods in particular circumstances.

A particularly serious problem of networks of parallel processes is deadlock – A state where each of the processes becomes permanently blocked. This paper describes the program Deadlock Checker, a tool which uses various innovative techniques to prove deadlock and livelock-freedom for networks of CSP[7] processes. It provides a vital safeguard against human error in the construction of parallel and distributed systems.

The FDR tool[6], of Formal Systems Europe, is often used to check for deadlock freedom. The method employed involves constructing a single transition system for the entire network and then checking each global state for deadlock. The main drawback to this approach is that the number of states of a system usually increases exponentially with the number of processes, and so this method is suitable only for analysis of rather small systems. Deadlock Checker operates by testing properties of individual CSP processes, or pairs of processes, within a network. This is done using a separate transition system for each process. The algorithms employed by Deadlock Checker, described below, scale efficiently to networks of arbitrary size. For instance, to check a network of *100* Dining Philosophers and Forks by exhaustive state analysis would require the construction of a transition system with $10^{\sim 50}$ states, whereas, using our methods, it is necessary only to construct a digraph with *800* vertices and then check it for circuits.

The price for this level of efficiency is incompleteness – there exist networks which are free of deadlock but that may not be proven so using this tool. This is not a high price as, despite this limitation, Deadlock Checker has no problem with proving deadlock-freedom for networks constructed using the following design rules: the *client-server* protocol[11], the *resource-allocation* protocol[15] and the *cyclic* protocol[10, 15, 18], and hybrid forms of these. The combination of these simple design rules and efficient machine verification would seem to be a powerful weapon against deadlock in large-scale process networks.

In section 2 we describe the basic operation of the Deadlock Checker program illustrated with the example of the Dining Philosophers network. In section 3 we explain the novel algorithm it uses for deadlock analysis. Section 4 covers several techniques for extending the power of this algorithm.

Table 1: Machine Readable CSP

Typeset CSP	ASCII CSP
$STOP$	`STOP`
$SKIP$	`SKIP`
$e \rightarrow P$	`e -> P`
$c!x \rightarrow P$	`c!x -> P`
$c?y \rightarrow P$	`c?y -> P`
$P \;[\![A \mid B]\!]\; Q$	`P [A||B] Q`
$P \mid\mid\mid Q$	`P ||| Q`
$P \sqcap Q$	`P |~| Q`
$P \Box Q$	`P [] Q`
$\Box_{i:A}\; P(i)$	`[] i:A @ P(i)`
$P \setminus A$	`P \ A`
$P \lhd (i = n) \rhd Q$	`if i == n then P else Q`

In section 5 we explain the algorithm used for livelock analysis, which is derived from a theorem of Roscoe. Section 6 comprises two interesting case studies. The first is a message routing system prototyped in CSP, verified by Deadlock Checker, and then implemented in occam. The second is a published algorithm which is shown to deadlock. Finally, in section 7, we discuss the current state of Deadlock Checker and how it may be extended in the future, possibly to analyse source code written in occam, Ada and Java.

2 Using Deadlock Checker

Programs to be analysed by Deadlock Checker are coded in the CSP language. These are compiled to networks of transition systems using a program which invokes the FDR tool. Interactive analysis may then proceed, using algorithms based on graph theoretical techniques.

Network Compilation

Deadlock Checker has a companion program called compile which is used to convert from a network program written in CSP (in the machine-readable format of Scattergood [16]) to a set of individual transition systems – one for each process in the network. These are then used by Deadlock Checker for performing various local checks in order to prove deadlock-freedom. Deadlock Checker is written entirely in Java. Program compile is implemented in ML on top of FDR version 1.4. which performs the actual compilation.

The main difference between machine readable CSP and the algebraic form is that, in the former, the *type* of communication channels has to be explicitly defined using a `pragma` statement. The representation of various CSP operators in ASCII format is given in table 1. Comment lines beginning with `--+` are used to specify to compile exactly which processes constitute the network to be analysed. There is no need to define the alphabets of these processes as the compiler calculates them automatically[1] (as being exactly those events that each process may ever perform).

Dijkstra's classic Dining Philosophers network may be defined as follows.

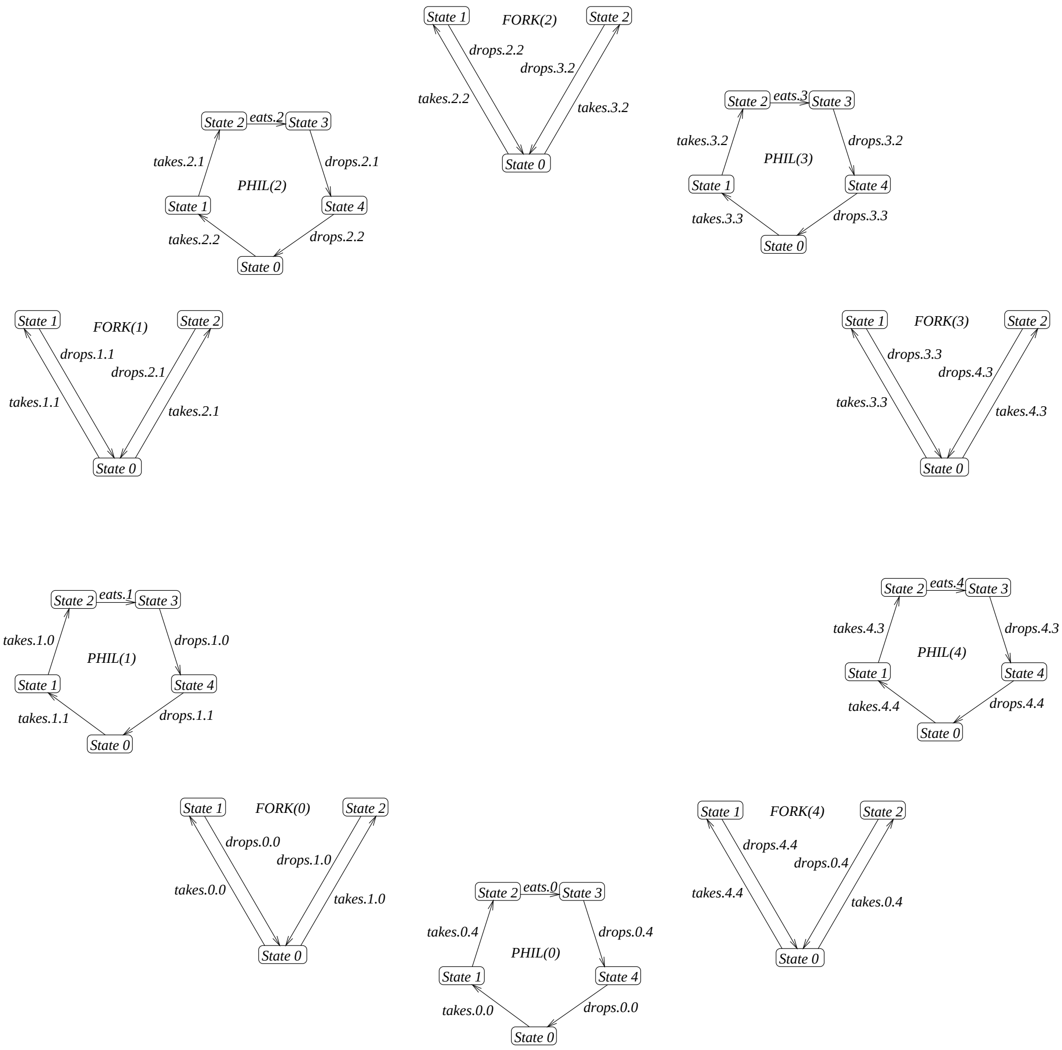
```
--- CSP process definitions

PHILNAMES = {0,1,2,3,4}
FORKNAMES = {0,1,2,3,4}
pragma channel eats:PHILNAMES
pragma channel takes,drops:PHILNAMES.FORKNAMES

PHIL(i) = takes.i.i -> takes.i.((i-1)%5) -> eats.i ->
          drops.i.((i-1)%5) -> drops.i.i -> PHIL(i)
```

[1]However, there are circumstances where one might wish to define the process alphabets explicitly, and this feature will be included in a future version of the program.

```
FORK(i) = takes.i.i -> drops.i.i -> FORK(i) []
          takes.((i+1)%5).i -> drops.((i+1)%5).i -> FORK(i)

--- Define network for Deadlock Checker

--+ PHIL(0),PHIL(1),PHIL(2),PHIL(3),PHIL(4)
--+ FORK(0),FORK(1),FORK(2),FORK(3),FORK(4)
```

This file, which is called `phils.csp` is processed by compile into a file `phils.net` containing a corresponding set of transition systems – one for each process in the network. Figure 1 illustrates the transition structures created from this network.

Figure 1: Transition Systems for Dining Philosophers



Network Analysis

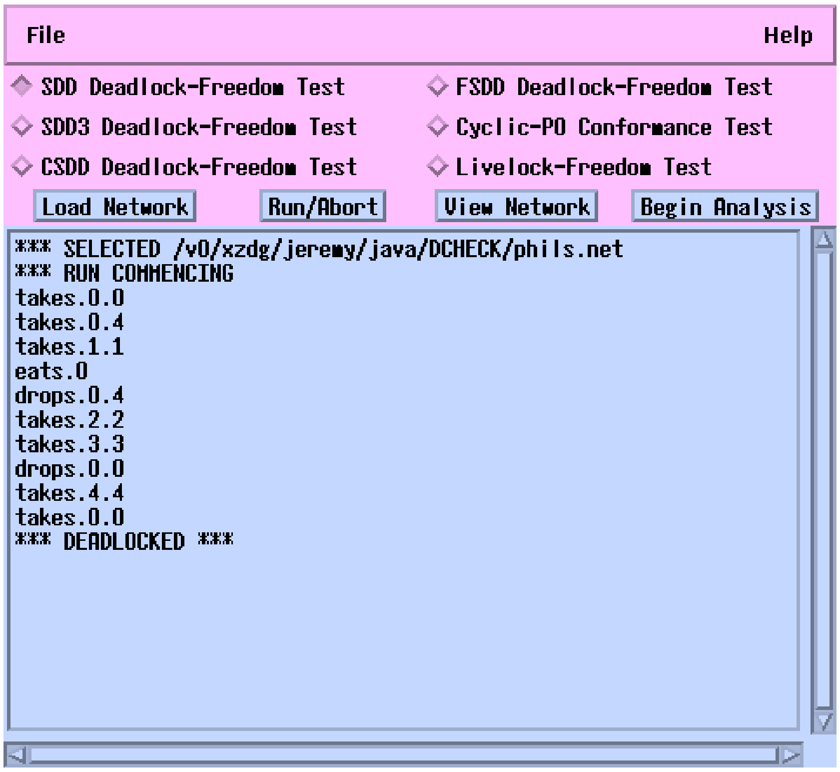
Following compilation the interactive analysis may proceed. We start up the main Deadlock Checker program (see figure 2) and press its Load Network button. A file browser is displayed which is used to load the compiled network. Two properties are immediately checked which are required for all the network analysis techniques.

Firstly the network must be *busy*. This means that each individual process is non-stopping – it would run forever if executed in isolation. This is checked by ensuring that every state of each transition system has at least one outgoing arc.

Secondly the network must be *triple-disjoint*. This means all communication is strictly between two processes. There are no events that are shared by three or more processes.

Pressing the Run button commences execution of the CSP network (see figure 2). Whenever deadlock ensues it is detected by the program. However this feature is intended only as a debugging aid – it can be used to show that a network deadlocks but never that it is deadlock-free.

Figure 2: Execution of the Dining Philosophers Network



The View Network button is another debugging feature. It enables us to explore the structure of the various processes comprising the network and also to visualise their intercommunication topology, as shown in figure 3.

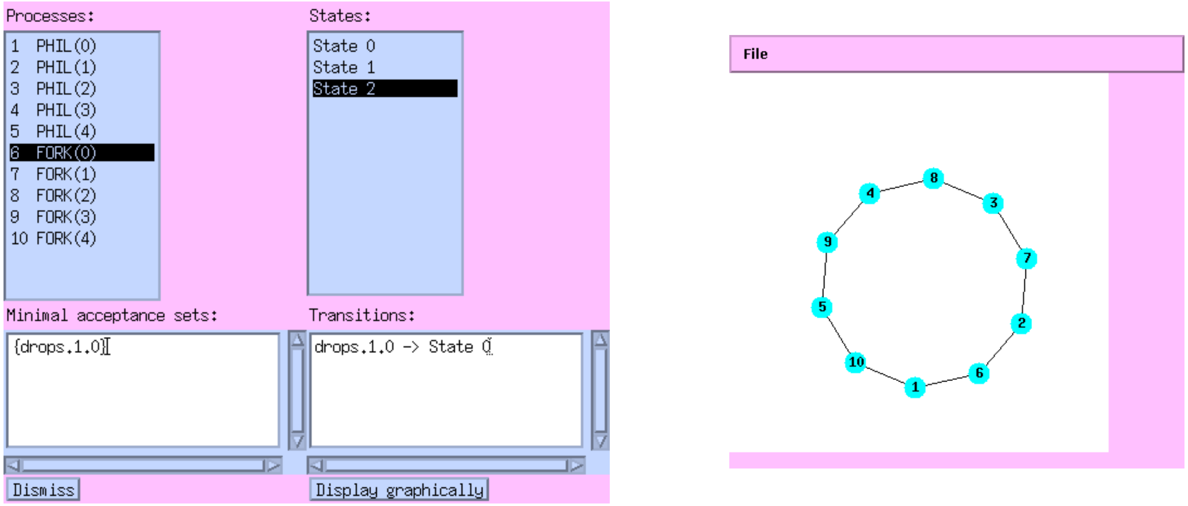
The Begin Analysis button is used to initiate each of the various network analysis options which are selected from the menu above it. Figure 4 shows what happens when the SDD test, which is described in the next section, is applied to the Dining Philosophers.

This network is not deadlock-free, and Deadlock Checker reveals the problem. A standard way to fix this deadlock is to make one of the philosophers change the order in which he picks up his forks. In that case Deadlock Checker has no problem in proving the network deadlock-free, using the SDD technique, which will now be explained.

3 The SDD Algorithm

A well-known characteristic of deadlock-states of busy, triple-disjoint networks, is that they involve a cycle of *ungranted requests*, i.e. a ring of processes each of which is blocking its predecessor and is blocked by its successor. (See, for example, [10].) This fact is the cornerstone of the SDD algorithm.

Figure 3: The Network Viewer



We shall attempt to show deadlock-freedom by proving the impossibility of existence of a cycle of ungranted requests.

Definition: Consider two process transition systems P and P' within a network $\langle P_1, .., P_N \rangle$. When process P is in state σ and process P' is in state σ' we say that P has an *ungranted request* to P' *if* P is ready to communicate with P' but P' is not ready to perform any event that P' is ready to perform, *and* neither P nor P' can perform any event other than to communicate with another process in the network. Ungranted requests may be thought of as the building blocks of deadlock.

A network's *state dependence digraph* (SDD) is defined as follows. It contains a vertex $P.\sigma$ for each state σ of each process P in the network.

It contains an arc $(P.\sigma, P'.\sigma')$ if, and only if, the following conditions apply

- Processes P and P' communicate with each other, i.e. $\alpha P \cap \alpha P' \neq \{\}$
- Within the subnetwork $\langle P, P' \rangle$ P can be in state σ at the same time as P' is in state σ'.
- When P is in state σ and P' is in state σ', P has an ungranted request to P'.

It is easy to see that if the network could ever exhibit a cycle of ungranted requests then this phenomenon would show up as a circuit in the SDD. Hence if the SDD has no circuits the network is deadlock-free. A formal proof of this result is found in [9].

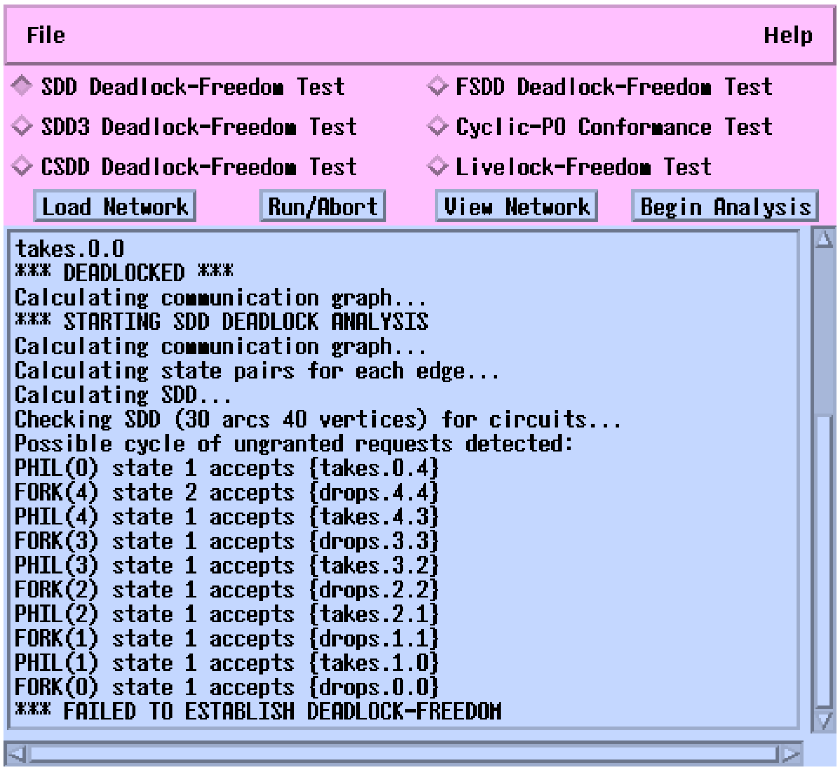
The basic algorithm for constructing the SDD of a network is as follows

1. Start with a network of transition systems $\langle P_1, P_2, ...P_N \rangle$. Form a digraph, *SDD*, the vertices of which are the states of the various processes P_i. Initially the digraph contains no arcs.
2. For each pair of communicating processes (P, P') we form the set $D(P, P')$ of all state pairs (σ, σ') that processes P and P' can be in simultaneously.
3. For each pair (σ, σ') in each $D(P, P')$ if P has an ungranted request to P', when in these respective states, add arc $(P.\sigma, P'.\sigma')$ to digraph *SDD*. And if P' has an ungranted request to P, add arc $(P'.\sigma', P.\sigma)$.

If the SDD is circuit-free then the network is stated to be deadlock-free, otherwise a potential cycle of ungranted requests is reported.

The SDD for our Dining Philosophers network is shown in figure 5. The single circuit that it contains corresponds to the single possible deadlock state. (This arises when each philosopher is holding exactly one fork.)

Figure 4: Deadlock Analysis of the Dining Philosophers



4 Eliminating Bogus Circuits from the SDD

The main advantage of our SDD test over the exhaustive global state analysis used by other tools is its efficiency. The algorithm has been shown (in [9]) to scale with near linear efficiency in the number of processes, for suitably well-constructed networks. However it is clear the property tested for by the SDD algorithm is somewhat stronger than deadlock-freedom. Firstly, if a circuit exists in the SDD it is not necessarily the case that the ungranted requests it represents can all occur at the same time in the network as a whole, and secondly, even if they can, a cycle of ungranted requests is not a *sufficient* condition for deadlock – it is merely a *necessary* condition. A network may exhibit a cycle of ungranted requests which is subsequently broken by the action of a process from outside the cycle.

Despite this the SDD algorithm may be used successfully to prove deadlock-freedom for a wide range of useful networks. In particular it can always prove deadlock-freedom for networks that obey the client-server protocol[11] or the resource-allocation protocol[15]. This is proven in [9].

But there are useful classes of network that cannot always be proven deadlock-free by this technique. For instance cyclic networks[4, 10, 15, 18] which are widely used for computationally intensive tasks.

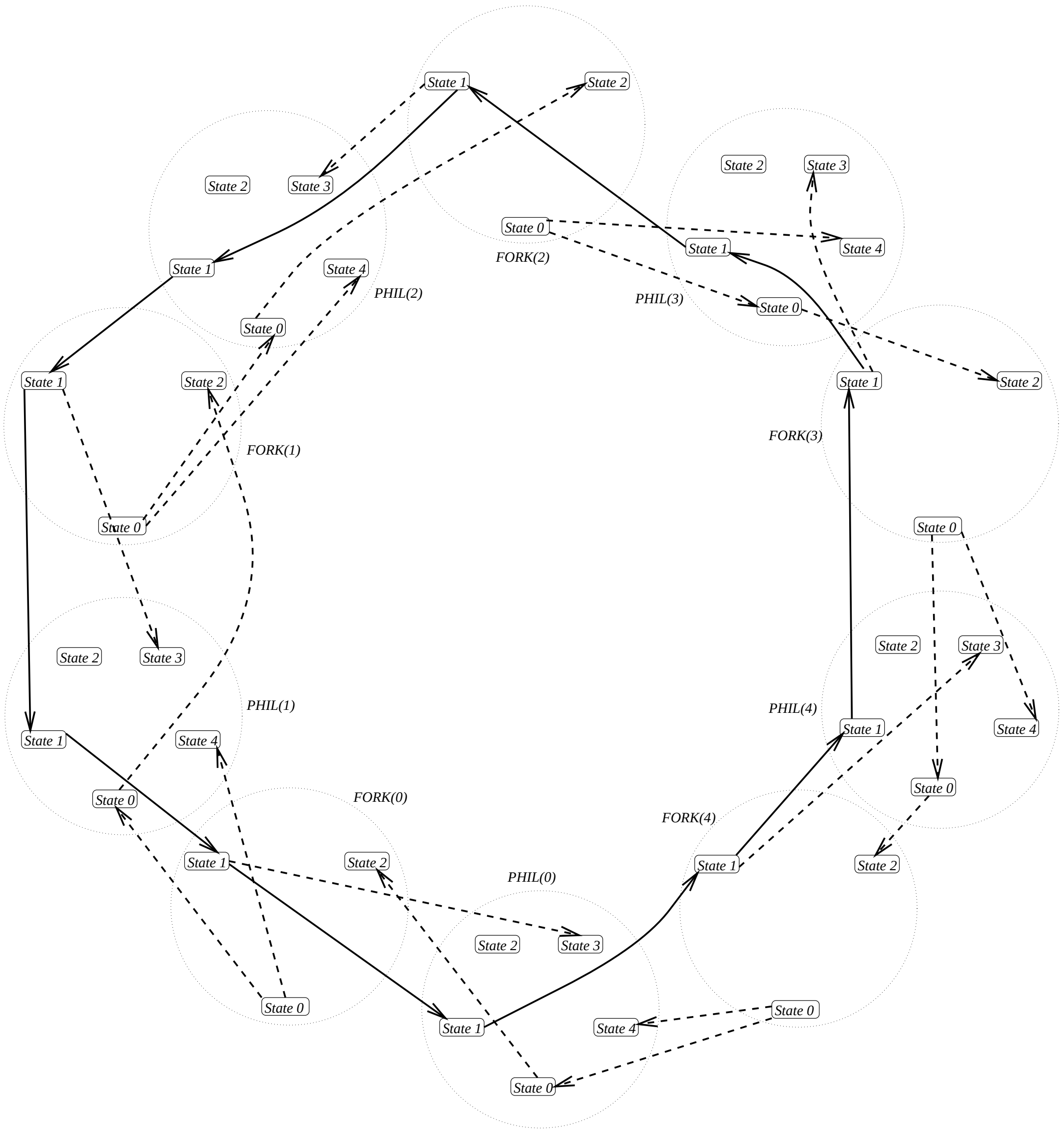
In this section we explore ways to increase the power of the SDD technique to address these shortcomings, either by adding extra information to the digraph or by performing radical surgery to it.

The Coloured State Dependence Digraph (CSDD)

We define the number of *iterations* of a process at a particular point in its execution (i.e. following a certain trace of events) as the number of times that it has returned to its initial state. We then colour arcs $(P.\sigma, P'.\sigma')$ in the SDD as follows

Red: P can only be in state σ at the same time as P' is in state σ' if both processes have performed the same number of iterations.

Figure 5: Construction of SDD for Dining Philosophers



Green: P can only be in state σ at the same time as P' is in state σ' if it has performed at least one more iteration.

Blue: Neither of the above properties is established.

A circuit containing some green and some red arcs but no blue ones cannot represent a real cycle of ungranted requests. We can prove that the ungranted requests that correspond to its arcs cannot all occur simultaneously by *reductio ad absurdum* reasoning about the number of iterations performed by each process in the circuit. However we cannot eliminate any circuit which *either* consists entirely of red arcs *or* contains at least one blue arc.

So we test for deadlock-freedom as follows. First we partition the vertices of SDD into *strongly connected components*, using an efficient algorithm from [5], and remove all arcs whose vertices lie in different components. This leaves us with exactly those arcs which lie on a circuit in the original SDD. Then if any blue arc remains there must be a circuit containing a blue arc and hence a possible deadlock. If no blue arc remains we remove the rest of the green arcs and check whether the remaining (all-red) digraph is circuit-free. If it is so the network is reported as being deadlock-free.

This particular colouring is sufficiently powerful for proving networks of cyclic processes deadlock-free[9].

The Flashing State Dependence Digraph (FSDD)

Alternatively we set arcs in the SDD to be *flashing* or *non-flashing* according to the following criterion

An arc $(P.\sigma, P'.\sigma')$ is set to be flashing only if it is known that whenever P is in state σ and P' is in state σ' then P' must have communicated with P at least once *and* more recently than with any other process.

A circuit consisting entirely of flashing arcs cannot represent a real cycle of ungranted requests for, if they did, we would know that each process in the circuit had communicated with its predecessor more recently than its successor, and following this argument around the circuit would lead to a contradiction. So we can state that a network is deadlock-free if all the arcs which lie on a circuit on its SDD are flashing. This graph property is tested in a similar manner to the coloured SDD check.

This extension to the SDD algorithm enables automatic proof of deadlock-freedom for the class of message routing networks introduced by A. W. Roscoe in [14].

Strong Conflict Freedom

It is sometimes useful to allow deadlock-free networks to exhibit cycles of requests of length 2. These are known as conflicts.

Definition: If when process P is in state σ and process P' is in state σ' there are ungranted requests *both* from P to P' and from P' to P we say that $(P.\sigma, P'.\sigma')$ is a conflict. In other words a *conflict* is a cycle of ungranted requests of length two. *Unless* P in state σ is also ready to communicate with some process other than P' *and* P' in state σ' is also ready to communicate with some process other than P we say that the conflict between P and P' is *strong.*

Definition: A network is strong-conflict-free if there is no strong conflict of any subnetwork $\langle P, P' \rangle$

Testing whether a network is strong-conflict-free may be done for a little extra effort during the construction of the SDD. It is shown in [1] that a deadlock state of any busy, triple-disjoint, strong-conflict-free network must have a cycle of ungranted requests of length at least three. This means that we can establish deadlock-freedom for such a network by showing that its SDD contains no circuits of length three or more. Circuits of length two are not a problem in this case.

Checking for the existence of a circuit in the SDD of length three or more is performed using the following algorithm devised by A. W. Roscoe. First partition the digraph into strongly connected components. Then for each component calculate the undirected simple graph which is induced. The original digraph contains no circuit of length three or more if, and only if, the induced simple graph of each strongly connected component is a tree. (The justification for this algorithm is left as an exercise to the interested reader.)

This particular extension to the SDD proof technique, which we call SDD3, enables us to prove deadlock-freedom for networks which might sometimes have conflicts (although not strong conflicts).

For example one can imagine a message routing network where an individual pair of neighbouring processes might sometimes each be waiting for the other one to pass across a message across. We have used this algorithm to prove deadlock-freedom for the switching system described in [15] which was also of this nature.

Vertex Colouring

Another form of bogus circuit that may be found in the SDD is one which passes through two or more states of the same process. It is clearly impossible for a process to be in two states at the same time so such a circuit cannot represent a real cycle of ungranted requests.

Suppose that we now colour the *vertices* of the state dependence digraph, where each colour represents the states of a particular process. To avoid the problem described above we are looking for an algorithm to determine whether this digraph contains a circuit in which every vertex has a different colour. At the time of writing no efficient algorithm has been found to decide this question in general. However even an inefficient algorithm would be useful in cases where the state-dependence digraph contains only a small number of circuits. Further investigation is in progress.

The Edge Selection Principle

Perhaps a more promising approach involving the aforementioned vertex colouring is based on the concept of *request selector functions*[2]. Suppose that there is some vertex of the state dependence digraph, $P.\sigma$, which has outgoing arcs to vertices which have two or more different colours. Now suppose that we choose one particular such colour $C(P.\sigma)$ and delete every outgoing arc from $P.\sigma$ that points to a vertex with a different colour from $C(P.\sigma)$. If the stripped down version of the state dependence digraph which results contains no circuit then it is still the case that the network must be deadlock-free. The result still holds no matter how many vertices are treated in this manner. There is insufficient room here to justify the edge-selection principle, but the interested reader should refer to [9] for further details of the theory on which it is based.

This property is potentially useful as follows. Suppose that a state-dependence digraph has been constructed and is found to contain circuits. An artificial-intelligence style algorithm is envisaged which would attempt to find a sequence of vertex and colour selections leading to the removal of sufficient arcs to render the digraph circuit-free, and hence prove deadlock-freedom. Finding the correct set of edge deletions appears to be difficult – it is rather like playing solitaire. At the moment we have tried an algorithm which works by trying to break as many circuits as possible at each stage. This appears promising, but it does not, as yet, play a strong *end-game* and is prone to leaving a few circuits behind that could have been broken through a wiser choice of edges to delete.

We are now investigating the possibility of using edge selection can in a different way to help remove circuits where each vertex has a different colour, as described in the previous section. This might be an interesting avenue for future research.

5 Proving Livelock Freedom

Deadlock Checker does not overlook the important property of livelock-freedom. We implement the proof rule of Roscoe (described in [2]) which works in many cases. First we need to check that each individual process in the network is itself free of livelock. We use FDR to check this at the time of network compilation. (It checks that there is no circuit of hidden actions in the transition system that is generated.) Then we need to show that no process can communicate indefinitely with those before it in the network list. These two properties can be shown to guarantee livelock freedom for the network by induction. The order in which the processes are supplied affects the success of this particular technique.

We check the condition as follows

1. Start with a network of processes $\langle P_1 \, .. \, P_n \rangle$ For each process P_i we calculate the subset of its alphabet shared with predecessors in the process list and call this N_i.

2. We then consider the subgraph of the transition system of each P_i containing only those arcs labelled with events which lie in N_i. If this subgraph contains no circuit then P_i cannot communicate indefinitely with its predecessors in the network list.

6 Case Studies

A Cube Router

Traditionally one of the most laborious tasks in parallel programming has been the routing of messages between processes which run on non-adjacent physical processors. Using a deadlock-free routing algorithm it is possible to implement unlimited virtual channels between transputers that are semantically equivalent to synchronous hardware links[14]. This work can be performed by a compiler, either partially or totally, freeing the programmer from much low-level effort.

We now consider the design of a deadlock-free routing algorithm for a network of eight transputers configured as a cube. The guiding principle that we shall use is to assign a *level* to each link between processors, and then to ensure than any message arriving at a processor on level n can only depart on a level greater than n. In this way deadlock can be avoided by ensuring that all messages travel "upwards" to their destination, which guarantees the free-flow of messages through the system. The system actually conforms to the *client-server* design rule for deadlock-freedom[11].

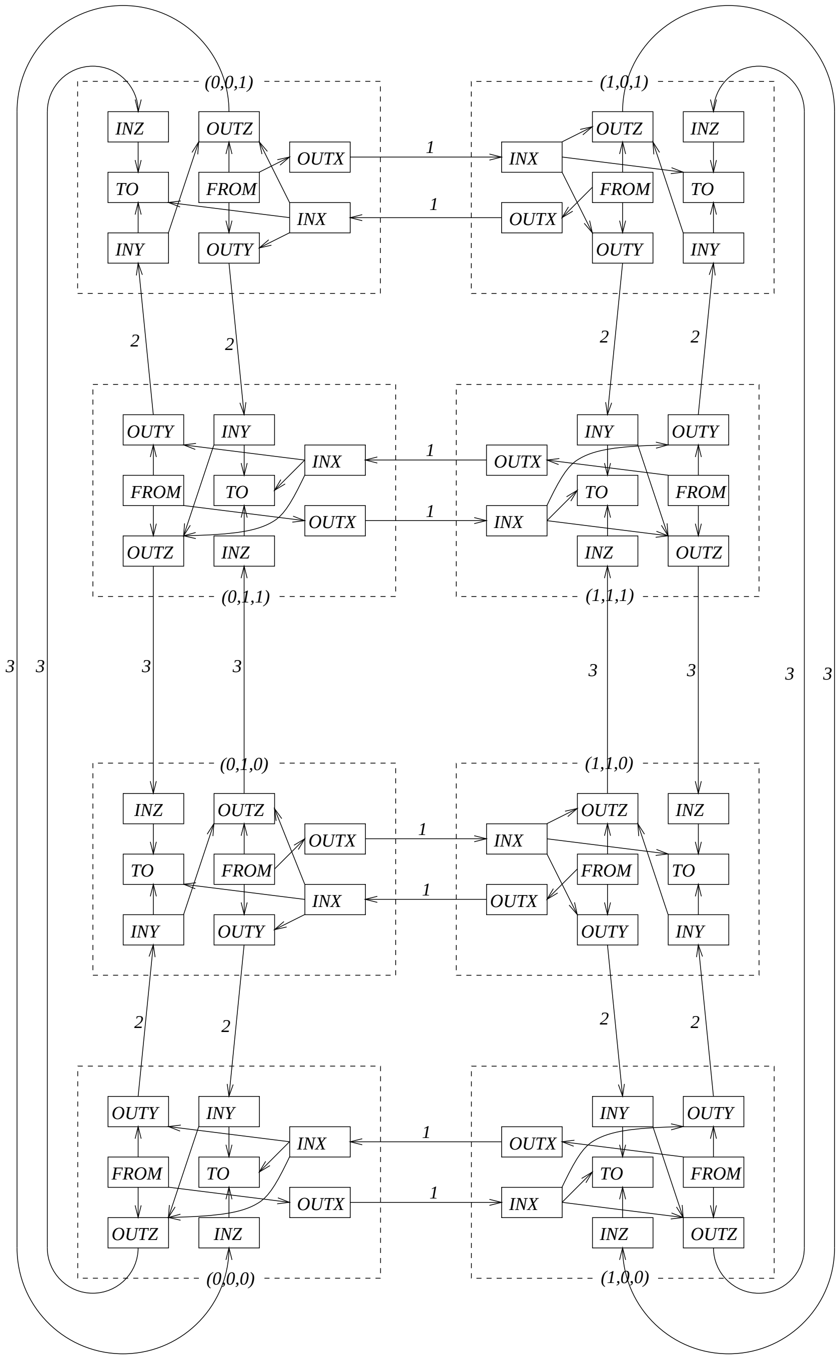
Figure 6 illustrates the router process topology superimposed on top of the processor topology. Each processor runs a separate process to control each of its input and output links. It also runs two interface processes, *TO* and *FROM*. The former collects messages which have arrived at their destination, and passes them to the local application process. The latter routes messages from the local application destined for other processes.

Links in the x direction are assigned level one, those in the y direction level two, and those in the z direction level three. In order to send a message to its destination the strategy used is first to get the x coordinate right, then the y coordinate, and finally the z coordinate.

The abstract CSP design of the program is listed below.

```
coords = {0,1}
direction = {dx, dy, dz}
change_direction = {xy,xz,yz}

-- 3 input links for each transputer

pragma channel i : coords.coords.coords.direction

-- Internal channels

pragma channel in, out : coords.coords.coords.direction
pragma channel q : coords.coords.coords.change_direction

-- Channels for interface to applications program

pragma channel to, from : coords.coords.coords

-- Processes to service input links

INX(x,y,z) = i.x.y.z.dx -> (out.x.y.z.dx -> INX(x,y,z) |~|
                            q.x.y.z.xy -> INX(x,y,z) |~|
                            q.x.y.z.xz -> INX(x,y,z))

INY(x,y,z) = i.x.y.z.dy -> (out.x.y.z.dy -> INY(x,y,z) |~|
                            q.x.y.z.yz -> INY(x,y,z))

INZ(x,y,z) = i.x.y.z.dz -> out.x.y.z.dz -> INZ(x,y,z)

-- Processes to service output links

OUTX(x,y,z) = in.x.y.z.dx -> i.((x+1)%2).y.z.dx -> OUTX(x,y,z)

OUTY(x,y,z) = in.x.y.z.dy -> i.x.((y+1)%2).z.dy -> OUTY(x,y,z) []
                q.x.y.z.xy -> i.x.((y+1)%2).z.dy -> OUTY(x,y,z)
```

Figure 6: Cube Router



```
OUTZ(x,y,z) = in.x.y.z.dz -> i.x.y.((z+1)%2).dz -> OUTZ(x,y,z) []
               q.x.y.z.xz -> i.x.y.((z+1)%2).dz -> OUTZ(x,y,z) []
               q.x.y.z.yz -> i.x.y.((z+1)%2).dz -> OUTZ(x,y,z)

-- Interface to application program

TO(x,y,z) =   out.x.y.z.dx -> to.x.y.z -> TO(x,y,z) []
              out.x.y.z.dy -> to.x.y.z -> TO(x,y,z) []
              out.x.y.z.dz -> to.x.y.z -> TO(x,y,z)

FROM(x,y,z) = from.x.y.z -> (     in.x.y.z.dx -> FROM(x,y,z) |~|
                                  in.x.y.z.dy -> FROM(x,y,z) |~|
                                  in.x.y.z.dz -> FROM(x,y,z) )


-- Now specify network for Deadlock Checker. The processes are
-- listed according to their "client-server" ordering.

--+FROM(0,0,0),FROM(0,0,1),FROM(0,1,0),FROM(0,1,1),
--+FROM(1,0,0),FROM(1,0,1),FROM(1,1,0),FROM(1,1,1),
--+OUTX(0,0,0),OUTX(0,0,1),OUTX(0,1,0),OUTX(0,1,1),
--+OUTX(1,0,0),OUTX(1,0,1),OUTX(1,1,0),OUTX(1,1,1),
--+INX (0,0,0),INX (0,0,1),INX (0,1,0),INX (0,1,1),
--+INX (1,0,0),INX (1,0,1),INX (1,1,0),INX (1,1,1),
--+OUTY(0,0,0),OUTY(0,0,1),OUTY(0,1,0),OUTY(0,1,1),
--+OUTY(1,0,0),OUTY(1,0,1),OUTY(1,1,0),OUTY(1,1,1),
--+INY (0,0,0),INY (0,0,1),INY (0,1,0),INY (0,1,1),
--+INY (1,0,0),INY (1,0,1),INY (1,1,0),INY(1,1,1),
--+OUTZ(0,0,0),OUTZ(0,0,1),OUTZ(0,1,0),OUTZ(0,1,1),
--+OUTZ(1,0,0),OUTZ(1,0,1),OUTZ(1,1,0),OUTZ(1,1,1),
--+INZ (0,0,0),INZ (0,0,1),INZ (0,1,0),INZ (0,1,1),
--+INZ (1,0,0),INZ (1,0,1),INZ (1,1,0),INZ (1,1,1),
--+TO  (0,0,0),TO  (0,0,1),TO  (0,1,0),TO  (0,1,1),
--+TO  (1,0,0),TO  (1,0,1),TO  (1,1,0),TO  (1,1,1)
```

This initial design avoids the issue of how to make routing decisions. When a message arrives on an input channel at a particular process it is redirected non-deterministically along any one of its output channels. Despite this disregard for any routing information the design is sufficiently robust to be proven deadlock-free by the SDD algorithm. The system is also shown to be livelock-free (see figure 7). The order in which the processes were supplied was crucial to proving livelock-freedom.

It is interesting to note that each of the sixty-four processes of this network may, or may not, be holding a message at any given time, which means that the system as a whole has at least 2^{64} states. This would put it well out of the range of any program using exhaustive state checking, such as FDR.

This application is a good example of the proper use of Deadlock Checker. The network was designed according to a tried and tested design rule which guarantees deadlock-freedom – the client-server protocol. Deadlock Checker was then able to give a formal guarantee that the no deadlock had crept into the system due to human error.

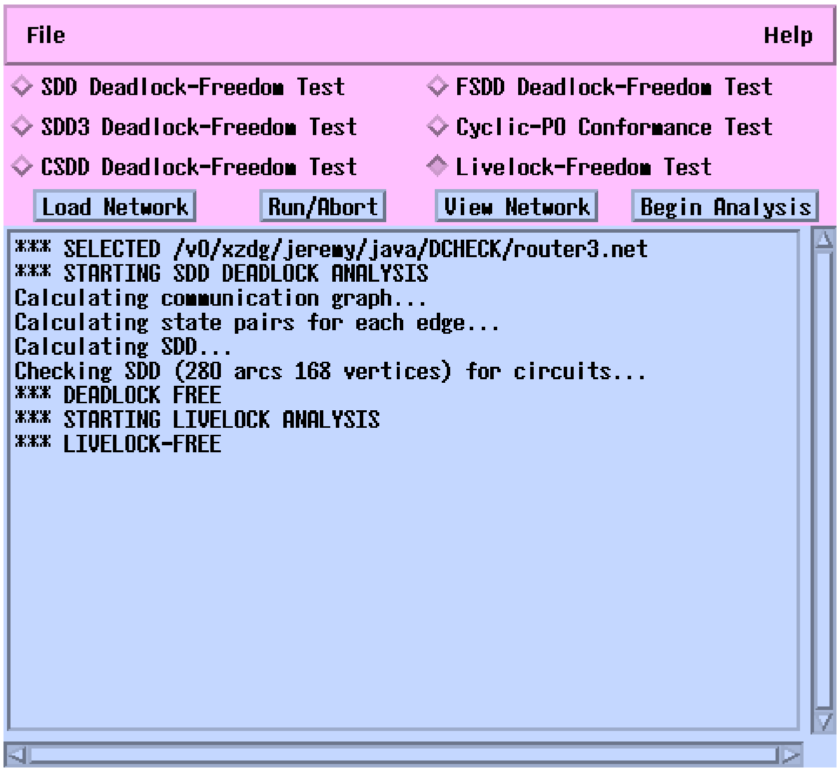
From the abstract design we were able to develop a working occam[8] implementation without difficulty. For instance, here is the process `INX` which runs on each transputer.

```
PROC INX(VAL INT x, y, z, processor)
  ...  local declarations
  WHILE TRUE
    SEQ
      i[x][y][z][dx] ? length :: packet
      IF
        packet[0] = processor  -- Arrived at destination
          out[x][y][z][dx] ! length :: packet
        ycoord(packet[0]) <> y -- Need to fix Y coordinate
          q[x][y][z][xy] ! length :: packet
        TRUE                   -- Need to fix Z coordinate
```

```
        q[x][y][z][xz] ! length :: packet
  :
```

The technique of assigning levels to processor links in order to effect a routing strategy can be generalised to processor networks of arbitrary construction. (Details are given in [3] and [13].) For certain topologies it is necessary to multiplex a number of virtual links on different levels, along a particular hardware link, in order to guarantee that there is always an upwards path between each pair of processors.

Figure 7: Deadlock and Livelock Analysis of the Cube Router



A Television Studio Control System

Now we consider an algorithm developed by N. Miller and Y. Bouchlaghem for the control of audio communications in a television studio [12]. The system, which is called Commander, consists of up to 384 control panels each of which has an associated analogue audio sound channel. The control panels are each connected to one of four central racks via a 96-way multiplexor. Each of these racks is then connected up to a cross-bar switch which is used to control audio connections between users. The four racks are also connected to each other so as to pass on switching requests from users, and to request information.

The hardware is based on transputers. There is one behind each control panel, and there are three in each rack: one to manage the multiplexor, one to control the cross-bar switch, and the third responsible for communication with the other racks, and the implementation of the high level system functionality. Apart from the inter-rack connections, all message passing conforms to the client-server paradigm. Each control panel runs a process *PANEL* which is a client of a multiplexor control process *PANELMGR*. This in turn is a client of a rack management process *RACKMGR* which is a client of a process *XBARMGR* which controls a cross-bar switch.

The only place where Miller and Bouchlaghem diverge from the client-server paradigm is in the inter-rack communications. Unfortunately we shall see that their system can deadlock because of this. The machine-readable CSP definition of this core subnetwork is as follows.

```
rack = {0,1,2,3}
pragma channel signal:rack
pragma channel arc:rack.rack.{req,ack}
sigma = {| arc, signal |}

RACKMGR(i) = ([] j:{x| x <- rack, x != i} @ arc.j.i?req ->
                 arc.i.j!ack -> RACKMGR(i))   []
             signal.i -> |~| j:{x| x <- rack, x != i} @ SEND(i,j,req)

SEND(i,j,x) = arc.i.j!x ->
                 ([] k:{x| x <- rack, x != i} @ arc.k.i?z ->
                 (if z == ack then RACKMGR(i) else SEND(i,k,ack)))   []
              ([] k:{x| x <- rack, x != i} @ arc.k.i?z ->
                 arc.i.j!x ->
                 (if z == ack then RACKMGR(i) else SEND(i,k,ack)))

--+ RACKMGR(0), RACKMGR(1), RACKMGR(2), RACKMGR(3)
```

Each rack manager process is initially waiting either for a signal to arrive from its panel manager, or a request from another rack. If it receives a request from another rack, this is immediately answered. If it receives a signal from its panel manager it may need to communicate with another rack. In this case it goes into "action" mode. First it sends out its request, and in parallel waits for a message to arrive from another rack. This message could either be the required answer to its request, or another request requiring an answer. In the former case the process returns to its initial state, in the latter it begins another cycle of parallel input and output. This time the output is an answer to the request that has just been received. The process continues with cycles of parallel inputs and outputs until an answer has been received to its original request.

When this network is analysed by Deadlock Checker, using the SDD algorithm, a potential cycle of ungranted requests is reported. This does not necessarily mean that the system actually does deadlock. However executing the network by pressing the Run button results in global deadlock – usually after a few minutes. Miller and Bouchlaghem report that their software has been running without problems on a system with over one hundred users, for some time. Perhaps this indicates that there is a very low probability of deadlock occurring in practice, where all action sequences need to be initiated by a human flicking a switch. However this type of uncertainty could certainly not be tolerated in a safety critical application, such as an air traffic control system.

It is possible to modify the system in order to render it deadlock-free, through adherence to the client-server protocol. Deadlock Checker can then be used to verify deadlock-freedom. This is explained in [9] where there is a more detailed analysis of this particular network.

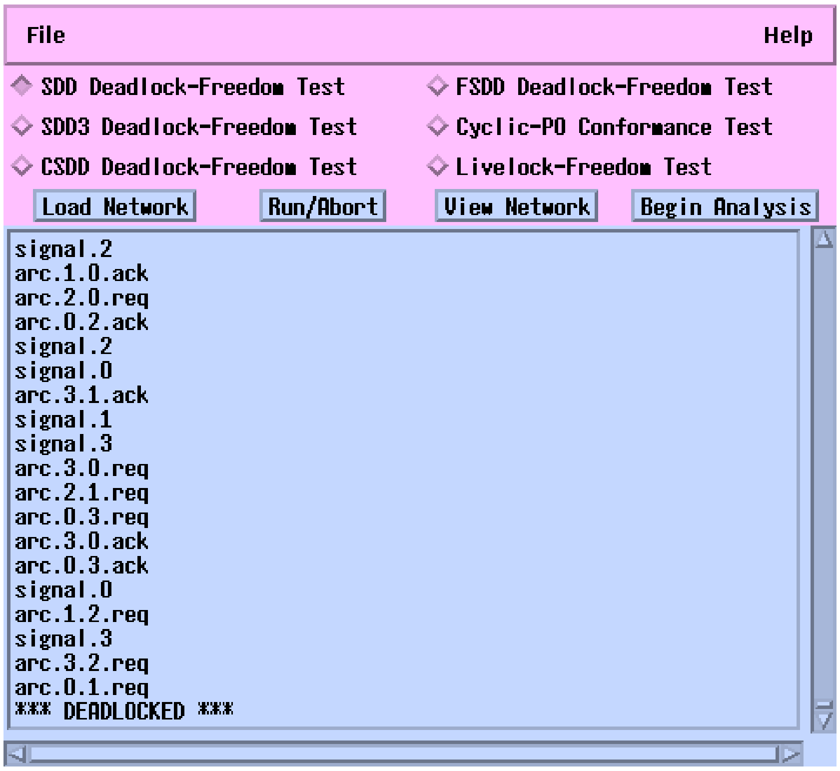
The system that has been considered here is in many ways a very fine piece of engineering. The fact that it has such a fundamental flaw is by no means a reflection on its developers. But it serves as a good example as to why tools like Deadlock Checker are so important.

7 Prospects for the Future

The main thread of this paper has been putting formal methods into action. We have described a tool which enables software engineers to design, implement and debug concurrent systems which are guaranteed free of deadlock, with the aid of design rules such as those described in [4, 9, 10, 11, 14, 15, 18, 19]. Existing tools which do this only work on a very small scale due to the problem of exponential state explosion, but our techniques overcome this restriction by being based on local analysis of process pairs rather than global states.

At the time of writing, Deadlock Checker incorporates our SDD, CSDD, FSDD and SDD3 deadlock tests, as well as Roscoe's livelock test. Testing for exact adherence to the cyclic-po design rule[10] is also included. Experimental code for removing circuits from the SDD involving edge-selection and vertex-colouring is under development but not yet part of the main program. Work is also underway to interface the program to version 2 of FDR, which offers a more flexible network input format.

Figure 8: Execution of Commander Resulting in Deadlock



The performance of the program is impressive. A network of 100 Dining Philosophers and Forks can be tested for deadlock in under 10 seconds on a Sun Sparcstation. It is predicted that Java compilers will produce code that runs 10 to 20 times faster in the future so there should be much improvement to come for no effort. The time required for the various checks appears to scale almost linearly in the number of processes for well-defined networks (see [9]).

It would be nice to use the tool to analyse source code for real programming languages such as occam, Ada, and perhaps Java. A tool has been developed by Formal Systems UK for extracting the CSP communication patterns from occam[17] so it should certainly be possible to analyse occam code. Ada also uses the CSP model for parallelism, so there is hope that something similar could be done. The designers of Java appear to have rejected CSP for Hoare's earlier system of *monitors*, which was superseded by CSP. So there would appear to be less hope for analysing Java programs with Deadlock Checker *unless* those programs incorporate an implementation of the CSP channel model, which is one of the important themes of this conference.

References

[1] S. D. Brookes and A. W. Roscoe *Deadlock Analysis of Networks of Communicating Processes*, Distributed Computing (1991)4, Springer Verlag 1991

[2] N. Dathi *Deadlock and Deadlock-Freedom*, Oxford University D.Phil Thesis 1990.

[3] M. Debbage, M. B. Hill and D. A. Nicole, *Global Communications on Locally Connected Message-Passing Parallel Computers*, Concurrency, Practice and Experience, September 1993.

[4] E. W. Dijkstra *A Class of Simple Communication Patterns*, Selected Writings on Computing: A Personal Perspective, Springer-Verlag 1982.

[5] S. Even, *Graph Algorithms* Computer Science Press, Inc. 1979.

[6] *FDR User Manual and Tutorial* Formal Systems (Europe) Ltd. 3 Alfred Street, Oxford OX1 4EH. Version 1.4 1994

[7] C. A. R. Hoare *Communicating Sequential Processes*, Prentice-Hall 1985.

[8] INMOS Limited occam2 *Reference Manual*, Prentice Hall 1988.

[9] J. M. R. Martin The Design and Construction of Deadlock-Free Concurrent Systems. D. Phil. Thesis, University of Buckingham 1996,also available at `http://www.hensa.ac.uk/parallel/theory/formal/csp`

[10] J. M. R. Martin, I. East and S. Jassim *Design Rules for Deadlock-Freedom*, Transputer Communications, September 1994.

[11] J. M. R. Martin and P. H. Welch *A Design Strategy for Deadlock-Free Concurrent Systems* to appear in Transputer Communications

[12] N. Miller and Y. Bouchlaghem *A Reliable Studio Control System – The Theory and the Practice*, Transputer Applications and Systems '95, IOS Press 1995.

[13] D. J. Pritchard *Load Balanced Deadlock-Free Deterministic Routing of Arbitrary Networks*, Proceedings of the 1992 ACM Computer Science Conference, 1992.

[14] A. W. Roscoe *Routing Messages Through Networks: An Exercise in Deadlock Avoidance*, Proceedings of the 7th occam User Group Technical Meeting, IOS Press 1988.

[15] A. W. Roscoe and Naiem Dathi *The Pursuit of Deadlock Freedom*, Oxford University Computing Laboratory (Technical Monograph PRG-57) 1986.

[16] B. Scattergood, *A Parser for CSP* 1992, available by anonymous ftp from `ftp://ftp.comlab.ox.ac.uk/pub/CSP/Parser`

[17] B. Scattergood and K. Seidel, *Converting occam to CSP*, Transputer Applications and Systems '94, IOS Press 1994.

[18] P. H. Welch *Emulating Digital Logic Using Transputer Networks*, Parallel Architectures and Languages Europe, LNCS 258, Springer-Verlag 1987.

[19] P. H. Welch, G. R. R. Justo, and C. J. Willcock *High-Level Paradigms for Deadlock-Free High-Performance Systems*, Transputer Applications and Systems '93, IOS Press 1993.