Chapter 4

Engineering Applications

Introduction

This chapter is intended to illustrate how the preceding work may be applied to real problems in software engineering. The occam programming language is introduced for this purpose and its relationship with CSP is elaborated. We then present three examples of designing and building industrial-scale deadlock-free concurrent systems.

The first problem considered is the numerical solution to Laplace's equation using the method of *successive over-relaxation*. This is typical of the sort of computationally-intensive task that parallel computers are often required to perform. Deadlock-freedom is incorporated into the design by using the cyclic-PO paradigm.

Next we describe the construction of a deadlock-free message routing program for a multiprocessor computer system. 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. For this reason a great deal of effort has been directed towards developing deadlock-free message routing programs. The intention of this is to separate all the physical message passing onto a lower conceptual level, and to implement *virtual* channels between any two locations in a processor network. Here we describe the construction of a *store and forward* deadlock-free message routing system for a network of eight processors configured as a cube. The client-server paradigm is used for this purpose. We then modify the program to implement *wormhole* routing, which is generally more efficient than store and forward-routing. In so doing we breach the rules of the client-server paradigm. However the resulting system is proven deadlock-free using the SDD algorithm.

The final example involves a published algorithm for a control system for a television studio. The system is shown to be prone to deadlock. However, with a simple modification, it may be transformed into a circuit-free client-server network resulting in guaranteed deadlock-freedom.

Table 4.1: Relationship between occam and CSP

occam	**CSP**

`SEQ`
`  P`
`  Q`
	$P\,;\,Q$

`PAR`
`  P`
`  Q`
	$(P \;[\![\, \alpha P \mid \alpha Q \,]\!]\; Q) \setminus (\alpha P \cap \alpha Q)$

`a?x`	$a?x \rightarrow \mathit{SKIP}$
`b!y`	$b!y \rightarrow \mathit{SKIP}$

`ALT`
`  c?x`
`    P`
`  d?y`
`    Q`
	$c?x \rightarrow P \;\Box\; d?y \rightarrow Q$

`IF`
`  b`
`    P`
`  NOT b`
`    Q`
	$P \lhd b \rhd Q$

`WHILE TRUE` `  P`	$\mu X \bullet P\,;\,X$

4.1 The occam Programming Language

The occam programming language, which is described in [INMOS 1988], was originally derived from the CSP model. The notation is somewhat different, but is elegant nonetheless. The language is unusual in that the indentation of the lines of code is syntactically significant. In the absence of an efficient compiler for CSP itself, occam represents the most appropriate implementation language for programs designed using CSP specifications.

Table 4.1 lists some roughly equivalent constructions between the two languages. One significant difference is that the occam parallel operator incorporates automatic hiding of communication events, which remain visible in CSP. This feature has the potential to introduce the phenomenon of livelock into a network. There are also certain extra high-level aspects to occam, such as prioritised external choice, timers and the assignment of variables.

Ideally we would like to build checks for deadlock-freedom and livelock-freedom into occam compilers. One way to do this would be to convert into CSP state-transition digraphs as used by the algorithms of Deadlock Checker. The translation of occam into CSP is considered informally in [Scattergood and Seidel 1994]. Problems arise from the treatment of the values of variables, leading to a potential explosion in the state-size of the resulting CSP. For instance a process that has a local variable which can take any real numeric value, usually needs to have at least one CSP state for each value.

Realistically we have to look at how much information can be discarded in the conversion, without removing any potential deadlocks or livelocks so that these may be detected. We need to establish a safe level of abstraction which maximises the performance of the tools. It is usually safe to represent communication events in occam purely by their channel names in the CSP specification. The one exception is when using a *variant protocol* on a particular channel. If the inputting process is unwilling to accept the type of datum offered by the outputting process, a local deadlock will ensue. (However, if an exhaustive case list is offered by the inputting process there can be no problem, but this may be impractical.).

The opposite route of translation from CSP to occam is considered in [Scott 1994]. The conversion is based on denotational semantics for occam [Goldsmith *et al* 1993].

4.2 Case Studies

Numerical Solution to Laplace's Equation

We consider the design and implementation of a parallel program to calculate the first order finite difference solution of Laplace's equation, by the method of *successive over-relaxation*. This technique is described in [Fox *et al* 1988].

The two dimensional Laplace equation is given by

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

We seek a solution for the unknown potential, U, across a rectangular grid domain, given fixed boundary values. We define an $m \times n$ array $U^{(k)}$, to represent the kth approximation to the result. Individual array elements are denoted $U_{ij}^{(k)}$, where *i* ranges from 0 to $m-1$, and *j* ranges from 0 to $n-1$. Each generation of U is calculated by the following iterative equation (We assume that $U^{(0)}$ is known.)

$$\begin{aligned} U_{i,j}^{(k)} &= \frac{\omega}{4}\left[U_{i-1,j}^{(k)} + U_{i,j-1}^{(k)} + U_{i+1,j}^{(k-1)} + U_{i,j+1}^{(k-1)}\right] + (1-\omega)\,U_{i,j}^{(k-1)} \\ &\quad \text{where} \quad (0 < i < m-1) \wedge (0 < j < n-1) \\ U_{i,j}^{(k)} &= U_{i,j}^{(0)} \quad \text{otherwise (fixed boundary condition)} \end{aligned}$$

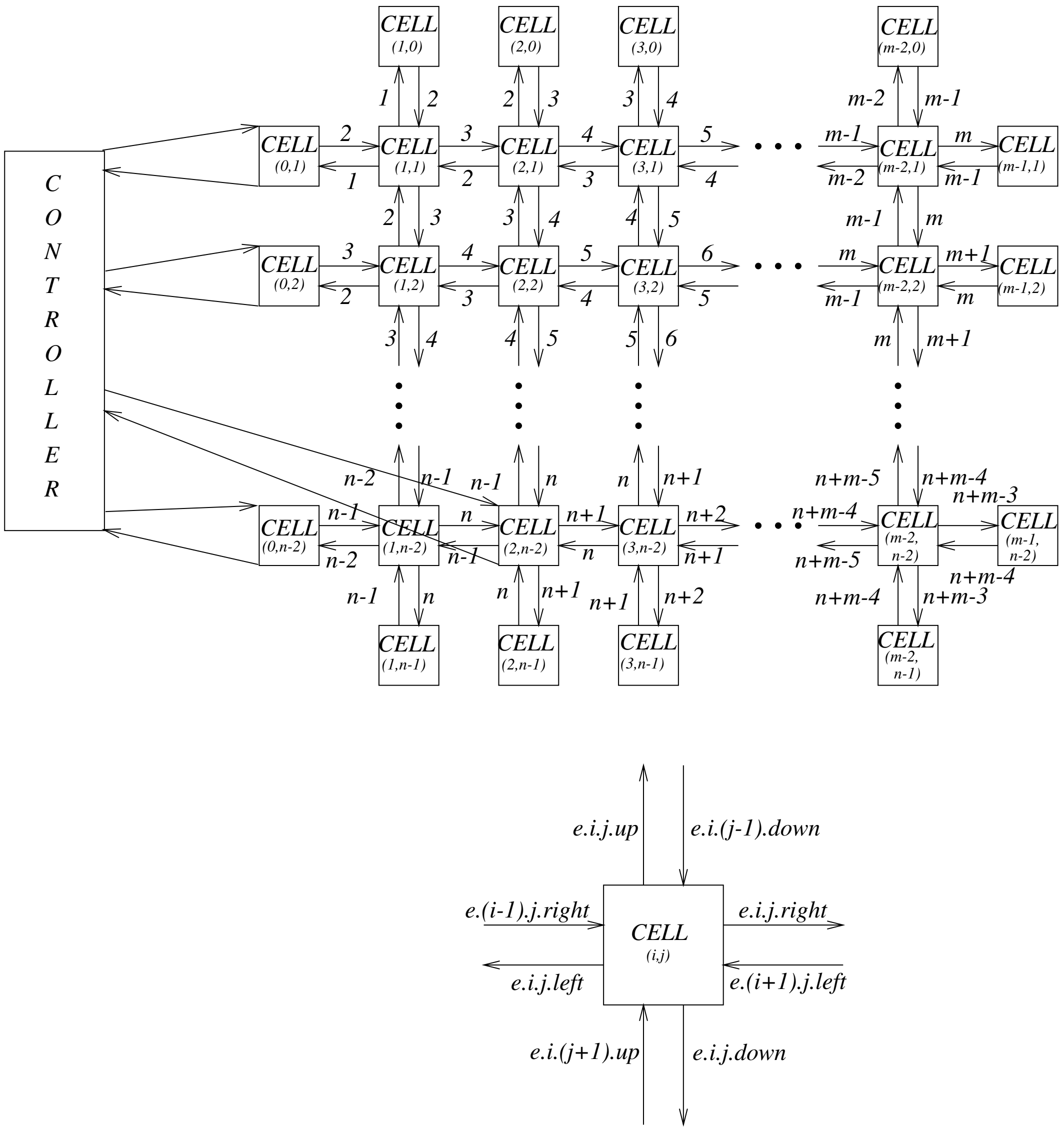
where ω is the *relaxation factor*.

The design of this parallel program is similar to the toroidal cellular automaton of section 2.1. We allocate a cyclic-PO process, *CELL*(i,j) to each grid element, connected by input and output channels to its neighbours. Process *CELL*(i,j) is responsible for calculating successive iterations of $U_{i,j}$. (The processes representing the boundary elements perform a trivial task as their state is fixed.) Each process also has bidirectional client-server connections to a control process, *CONTROL*, for periodic resets.

It will be seen that the iterative equation imposes an ordering on channels between neighbouring grid cells – on a given *I/O* cycle, a process needs to wait for its immediate left and upper neighbours to compute their new states, before it can inquire their new values and compute its own new state. Figure 4.1 illustrates a feasible deadlock-free channel ordering for this strategy. Based on this labelled connection diagram, the communication pattern of each process in the network is defined as follows.

$$
\begin{array}{rcl}
\mathit{CHAT}(i,j) & = & \mathit{SKIP} \sqcap \mathit{out}.i.j \rightarrow \mathit{in}.i.j \rightarrow \mathit{SKIP} \\
\mathit{CELL}(i,j) & = & \mathit{CHAT}(i,j)\ ; \\
 & & \left(\begin{array}{l} (e.i.j.\mathit{left} \rightarrow \mathit{SKIP}) \quad ||| \\ (e.i.j.\mathit{up} \rightarrow \mathit{SKIP}) \end{array} \right); \\
 & & \left(\begin{array}{l} (e.(i+1).j.\mathit{left} \rightarrow \mathit{SKIP}) \quad ||| \\ (e.i.(j+1).\mathit{up} \rightarrow \mathit{SKIP}) \quad ||| \\ (e.(i-1).j.\mathit{right} \rightarrow \mathit{SKIP}) \quad ||| \\ (e.i.(j-1).\mathit{down} \rightarrow \mathit{SKIP}) \end{array} \right); \\
 & & \left(\begin{array}{l} (e.i.j.\mathit{right} \rightarrow \mathit{SKIP}) \quad ||| \\ (e.i.j.\mathit{down} \rightarrow \mathit{SKIP}) \end{array} \right); \\
 & & \mathit{CELL}(i,j) \\
 & & \\
 & & \text{where} \quad (0 < i < m-1) \wedge (0 < j < n-1) \\
 & & \\
\mathit{CELL}(0,j) & = & \mathit{CHAT}(0,j)\ ; \\
 & & e.1.j.\mathit{left} \rightarrow e.0.j.\mathit{right} \rightarrow \mathit{CELL}(0,j) \\
 & & \\
\mathit{CELL}(m-1,j) & = & \mathit{CHAT}(m-1,j)\ ; \\
 & & e.(m-1).j.\mathit{left} \rightarrow e.(m-2).j.\mathit{right} \rightarrow \mathit{CELL}(m-1,j) \\
 & & \\
 & & \text{where} \quad (0 < j < n-1) \\
 & & \\
\mathit{CELL}(i,0) & = & \mathit{CHAT}(i,0)\ ; \\
 & & e.i.1.\mathit{up} \rightarrow e.i.0.\mathit{down} \rightarrow \mathit{CELL}(i,0) \\
 & & \\
\mathit{CELL}(i,n-1) & = & \mathit{CHAT}(i,n-1)\ ; \\
 & & e.i.(n-1).\mathit{up} \rightarrow e.i.(n-2).\mathit{down} \rightarrow \mathit{CELL}(i,n-1)
\end{array}
$$

Figure 4.1: Labelled Connection Diagram for Laplace Solver



$$\text{where} \quad (0 < i < m - 1)$$

$$\begin{aligned} CONTROL \;=\; & (\Box_{i=1}^{m-2} \Box_{j=1}^{n-2}\, out.i.j \rightarrow in.i.j \rightarrow CONTROL)\, \Box \\ & \left(\Box_{i=1}^{m-2} \left(\begin{array}{c} out.i.0 \rightarrow in.i.0 \rightarrow CONTROL\, \Box \\ out.i.(n-1) \rightarrow in.i.(n-1) \rightarrow CONTROL \end{array} \right) \right) \Box \\ & \left(\Box_{j=1}^{n-2} \left(\begin{array}{c} out.0.j \rightarrow in.0.j \rightarrow CONTROL\, \Box \\ out.(m-1).j \rightarrow in.(m-1).j \rightarrow CONTROL \end{array} \right) \right) \end{aligned}$$

The CSDD algorithm of Deadlock Checker can be used to verify that this particular network is deadlock-free, for given values of n and m. It is straightforward to develop an occam implementation of the program based on this specification. There follows a possible implementation of an interior cell process.

```
PROC CELL (VAL INT i, j)
  REAL32 w, x, y, z, state:
  INT k, ncycles:
  SEQ
    state := 0.0 (REAL32)
    WHILE TRUE
      SEQ
        out[i][j] ! state            -- Communicate with
        in[i][j] ? state; ncycles    -- CONTROL
        k := 0
        WHILE k<ncycles
          SEQ                        -- Perform next
            k:=k+1                   -- iteration
            PAR
              e[i][j][LEFT] ! state
              e[i][j][UP] ! state
            PAR
              e[i+1][j][LEFT] ? w
              e[i][j+1][UP] ? x
              e[i-1][j][RIGHT] ? y
              e[i][j-1][DOWN] ? z
            state := ((((w+x)+y)+z) *
                      (OMEGA/4.0(REAL32))) +
                      (state *(1.0(REAL32) - OMEGA)))
            PAR
              e[i][j][RIGHT] ! state
              e[i][j][DOWN] ! state
:
```

A Message Router

Suppose we wished to realise the Laplace solving network on a parallel machine constructed from a collection of Inmos transputers. We would most probably need to run a considerable number of *CELL* processes on each processor. However each transputer only has four hardware links to neighbouring processors which would be insufficient compared with the number of communication channels that would need to be implemented. Some form of multiplexing would be required.

Historically this has been a somewhat irritating problem for programmers of parallel machines. Even for a simple process network a large amount of work has often been needed to map it onto the target hardware configuration. Frequently the resulting implementation has not even been semantically equivalent to the original, sometimes resulting in unforeseen deadlocks.

Using a deadlock-free routing algorithm it is possible to implement unlimited virtual channels between transputers that *are* semantically equivalent to synchronous hardware links [Roscoe 1988b]. This work can be performed by a compiler, either partially or totally, freeing the programmer from much low-level effort.

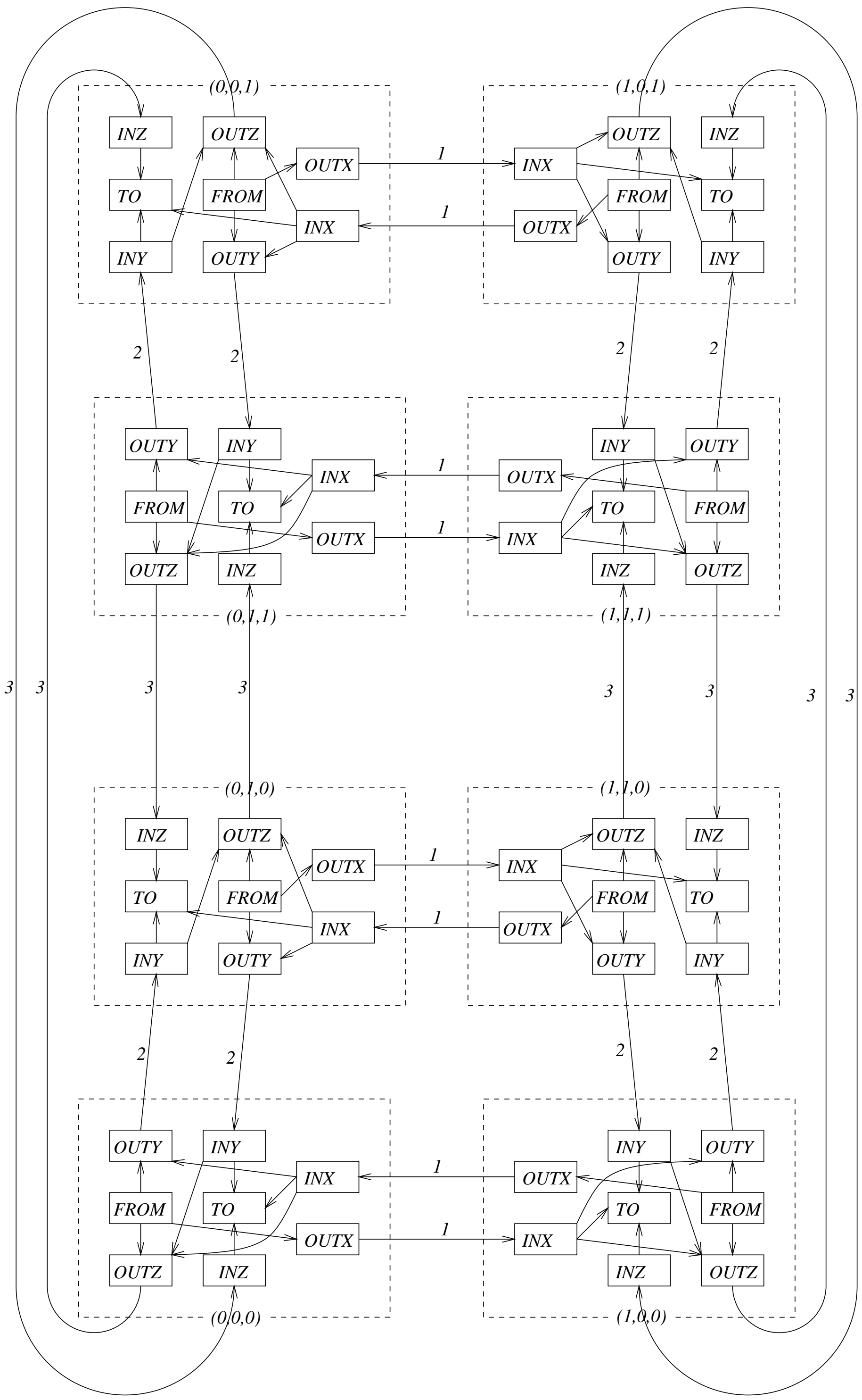
We now consider the design of a deadlock-free routing algorithm for a network of eight transputers configured as a cube, based on a program from [Shumway 1990]. The client-server paradigm will be employed. The guiding principle that we shall use is to assign a *level* to each link between processors, 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 that the client-server digraph is circuit-free. Figure 4.2 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
```

Figure 4.2: Cube Router



```
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)

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 adherence to the client-server protocol. In this case each individual channel is a client-server bundle of size one. A process acts as a server on its input channels and as a client on its output channels. This means that the client-server digraph for the system is the same as the connection digraph. The condition that messages must always travel upwards guarantees that it is circuit-free. (Note that the network could be represented rather more compactly using an exploded client-server digraph, treating the set of processes that run on each transputer as a single composite-client-server process.) Deadlock-freedom is easily verified using Deadlock Checker.

```
Welcome to Deadlock Checker
Command (h for help, q to quit):l router.net
Command (h for help, q to quit):w
Network router.net is busy
Network router.net is triple-disjoint
Process FROM(0,0,0) obeys client-server protocol
clients(FROM(0,0,0)) =
{<in.0.0.0.dz>,
 <in.0.0.0.dy>,
 <in.0.0.0.dx>}
servers(FROM(0,0,0)) =
{}
...
Process TO(1,1,1) obeys client-server protocol
clients(TO(1,1,1)) =
{}
servers(TO(1,1,1)) =
{<out.1.1.1.dx>,
 <out.1.1.1.dy>,
 <out.1.1.1.dz>}
Network router.net is deadlock-free
```

The system may also be shown to be livelock-free at this stage.

```
Command (h for help, q to quit):t
```

```
Network router.net is triple-disjoint
Network router.net is livelock-free
```

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.

From the abstract design we are now able to develop a working occam implementation without difficulty. For instance, here is the process INX which runs on each transputer.

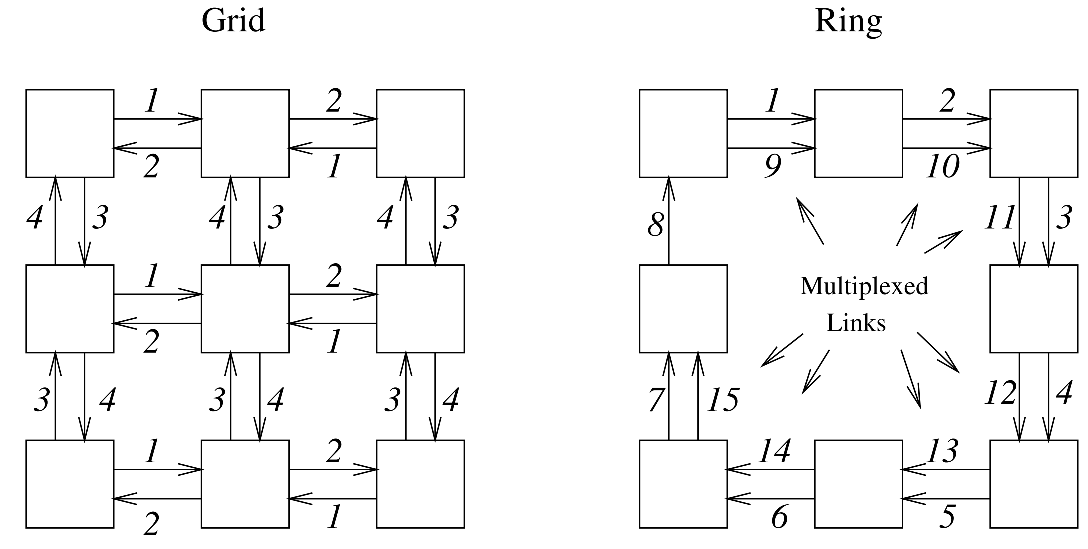
```
PROC INX(VAL INT x, y, z, processor)
  ...  local declarations
  WHILE TRUE
    SEQ
      i[x][y][z][dx] ? length :: packet
      IF
        packet[0] = processor  -- Arrived at destination
          out[x][y][z][dx] ! length :: packet
        ycoord(packet[0]) <> y -- Need to fix Y coordinate
          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 [Debbage *et al* 1993] and [Pritchard 1992].) 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 4.3 illustrates link labelling schemes for a ring and a grid. The former involves the use of virtual multiplexed links, but the latter does not. Multiplexing is a potential pitfall and must be implemented with great care. A good method of multiplexing is described in [Jones and Goldsmith 1988]. A process is constructed which utilises a single transputer link and yet is semantically equivalent to a collection of independent one-place buffers. (Note that it cannot be assumed in general that it is safe to add buffering along a channel of a network. Any such modification needs to be considered as part of the overall deadlock analysis.)

Worm-hole Routing

Worm-hole routing differs from store and forward routing in that a message is split up into small packets and these are sent across the network together by cutting a virtual path through it, and holding this path open until the last packet has passed through. The following CSP code illustrates a modification to the design for the cube router which uses this strategy.

Figure 4.3: Routing Strategies for Ring and Grid




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

-- 3 input links for each transputer

pragma channel i : coords.coords.coords.direction.packets

-- Internal channels

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

-- Channels for interface to applications program

pragma channel to, from : coords.coords.coords.packets

-- Processes to service input links

INX(x,y,z) = i.x.y.z.dx.data ->
             (out.x.y.z.dx.data -> INX1(x,y,z) |~|
              q.x.y.z.xy.data -> INX2(x,y,z,xy) |~|
              q.x.y.z.xz.data -> INX2(x,y,z,xz))
INX1(x,y,z) = i.x.y.z.dx?p -> out.x.y.z.dx.p ->
              if p == data then INX1(x,y,z) else INX(x,y,z)
INX2(x,y,z,cd) = i.x.y.z.dx?p -> q.x.y.z.cd.p ->
                 if p == data then INX2(x,y,z,cd) else INX(x,y,z)
```

```
INY(x,y,z) = i.x.y.z.dy.data ->
             (out.x.y.z.dy.data -> INY1(x,y,z) |~|
              q.x.y.z.yz.data -> INY2(x,y,z,yz))
INY1(x,y,z) = i.x.y.z.dy?p -> out.x.y.z.dy.p ->
              if p == data then INY1(x,y,z) else INY(x,y,z)
INY2(x,y,z,cd) = i.x.y.z.dy?p -> q.x.y.z.cd.p ->
                 if p == data then INY2(x,y,z,cd) else INY(x,y,z)

INZ(x,y,z) = i.x.y.z.dz.data ->
             out.x.y.z.dz.data -> INZ1(x,y,z)
INZ1(x,y,z) = i.x.y.z.dz?p -> out.x.y.z.dz.p ->
              if p == data then INZ1(x,y,z) else INZ(x,y,z)

-- Processes to service output links


OUTX(x,y,z) = in.x.y.z.dx.data ->
              i.((x+1)%2).y.z.dx.data -> OUTX1(x,y,z)
OUTX1(x,y,z) = in.x.y.z.dx?p -> i.((x+1)%2).y.z.dx.p ->
               if p == data then OUTX1(x,y,z) else OUTX(x,y,z)

OUTY(x,y,z) = in.x.y.z.dy.data ->
              i.x.((y+1)%2).z.dy.data -> OUTY1(x,y,z) []
              q.x.y.z.xy.data ->
              i.x.((y+1)%2).z.dy.data -> OUTY2(x,y,z,xy)
OUTY1(x,y,z) = in.x.y.z.dy?p -> i.x.((y+1)%2).z.dy.p ->
               if p == data then OUTY1(x,y,z) else OUTY(x,y,z)
OUTY2(x,y,z,cd) = q.x.y.z.cd?p -> i.x.((y+1)%2).z.dy.p ->
                  if p == data then OUTY2(x,y,z,cd) else OUTY(x,y,z)

OUTZ(x,y,z) = in.x.y.z.dz.data ->
              i.x.y.((z+1)%2).dz.data -> OUTZ1(x,y,z)  []
              q.x.y.z.xz.data ->
              i.x.y.((z+1)%2).dz.data -> OUTZ2(x,y,z,xz) []
              q.x.y.z.yz.data ->
              i.x.y.((z+1)%2).dz.data -> OUTZ2(x,y,z,yz)
OUTZ1(x,y,z) = in.x.y.z.dz?p -> i.x.y.((z+1)%2).dz.p ->
               if p == data then OUTZ1(x,y,z) else OUTZ(x,y,z)
OUTZ2(x,y,z,cd) = q.x.y.z.cd?p -> i.x.y.((z+1)%2).dz.p ->
                  if p == data then OUTZ2(x,y,z,cd) else OUTZ(x,y,z)

-- Interface to application program

TO(x,y,z) =  out.x.y.z?d?p -> to.x.y.z.p -> TO(x,y,z)

FROM(x,y,z) = from.x.y.z.data ->
              (in.x.y.z.dx.data -> FROM2(x,y,z,dx) |~|
               in.x.y.z.dy.data -> FROM2(x,y,z,dy) |~|
               in.x.y.z.dz.data -> FROM2(x,y,z,dz) )
FROM2(x,y,z,d) = from.x.y.z?p -> in.x.y.z.d.p ->
```

```
                 if p == data then FROM2(x,y,z,d) else FROM(x,y,z)
```

This design actually contravenes the rules for client-server communication. Once the first packet of a message has been received, a process will then only be prepared to communicate on one of its server channels. However the network is still easily proven deadlock-free using the SDD algorithm. Livelock-freedom is also preserved.

```
Command (h for help, q to quit):l wormhole.net
Command (h for help, q to quit):v
Network wormhole.net is triple-disjoint
Network wormhole.net is busy
Checking INZ(1,1,1) with TO(1,1,1)
Checking INZ(1,1,0) with TO(1,1,0)
...
Network wormhole.net is deadlock-free
Command (h for help, q to quit):t
Network wormhole.net is triple-disjoint
Network wormhole.net is livelock-free
```

This is an interesting example because although a reasonable solution was achieved to the initial problem using only design rules, in order to develop a more efficient solution it was necessary to bend the rules.

A Television Studio Control System

This example differs from the previous two, in that we start with a published algorithm which is closely related to our design rules, but ultimately breaches them. First we show that this algorithm is theoretically prone to deadlock. Then we consider how the design can be modified to remove this problem. The system considered 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. The main motivation for this thesis is that such problems are almost inevitable in practice unless suitable design rules for avoiding them are provided.

The algorithm was developed by N. Miller and Y. Bouchlaghem for the control of audio communications in a television studio [Miller and Bouchlaghem 1995]. 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. Figure 4.4 shows the connection digraph for the processes running on this system. 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 *PANEL.MGR*. This in turn is a client of a rack management process *RACK.MGR* which is a client of a process *XBAR.MGR* which controls a cross-bar switch.

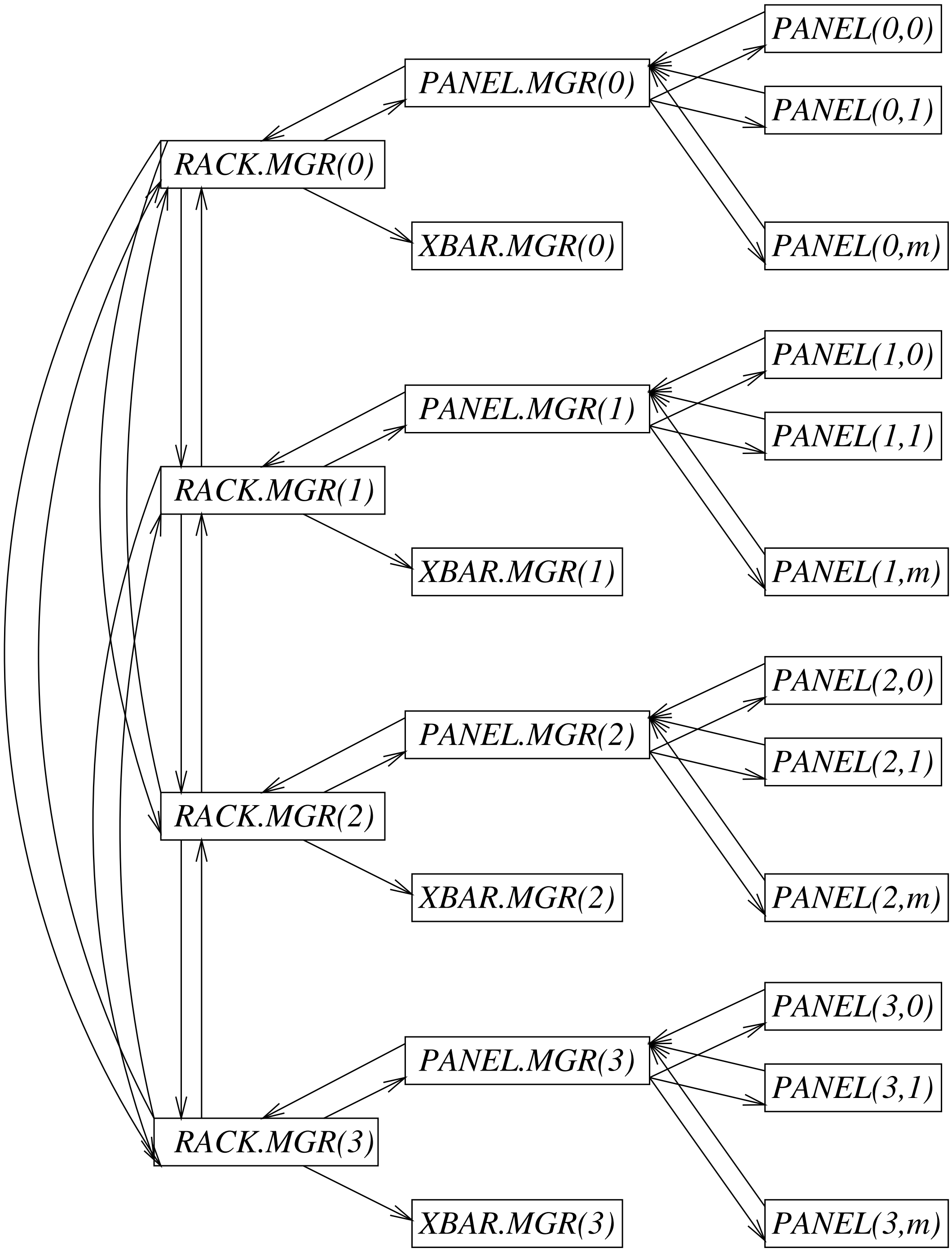
The only place where Miller and Bouchlaghem diverge from the client-server paradigm is in the inter-rack communications. Unfortunately we shall see that their system can deadlock because of this. We shall concentrate on the CSP definition for the sub-network of *RACK.MGR* processes. (Note that this definition conceals communications with *XBAR-MGR* processes.)

$$
\begin{aligned}
\mathit{RACK.MGR}(i) &= \mathit{from.panel.mgr.i} \rightarrow (\mathit{ACTION}(i)\,;\mathit{RACK.MGR}(i)) \;\Box \\
&\quad\; (\Box_{j \neq i}\; \mathit{chan}.j.i.\mathit{req} \rightarrow \mathit{chan}.i.j.\mathit{ack} \rightarrow \mathit{RACK.MGR}(i)) \\[1ex]
\mathit{ACTION}(i) &= \mathit{SKIP} \sqcap (\sqcap_{j \neq i}\; \mathit{INITIATE}(i, j, \mathit{req})) \\[1ex]
\mathit{INITIATE}(i, j, x) &= \begin{pmatrix} \mathit{chan}.i.j!x \rightarrow \Box_{k \neq i}\; \mathit{chan}.k.i?z \rightarrow \\ (\mathit{SKIP} \lhd (z = \mathit{ack}) \rhd \mathit{INITIATE}(i, k, ack)) \end{pmatrix} \Box \\
&\quad\; \begin{pmatrix} \Box_{k \neq i}\; \mathit{chan}.k.i?z \rightarrow \mathit{chan}.i.j!x \rightarrow \\ (\mathit{SKIP} \lhd (z = \mathit{ack}) \rhd \mathit{INITIATE}(i, k, ack)) \end{pmatrix} \\[1ex]
\mathit{RACKS} &= \left\langle \begin{matrix} \mathit{RACK.MGR}(0), & \mathit{RACK.MGR}(1), \\ \mathit{RACK.MGR}(2), & \mathit{RACK.MGR}(3) \end{matrix} \right\rangle
\end{aligned}
$$

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 network *RACKS* is analysed by Deadlock Checker, using the SDD algorithm, it is reported that strong-conflict can occur between neighbouring processes. As the number of states of the system is relatively small (about three thousand), exhaustive state analysis is feasible, using the FDR tool. This reveals that the network may dead-

Figure 4.4: Connection Digraph for COMMANDER



lock after the following trace.

$$\left\langle \begin{array}{ccc} \mathit{from.panel.mgr.0}, & \mathit{from.panel.mgr.1}, & \mathit{from.panel.mgr.2}, \\ \mathit{from.panel.mgr.3}, & \mathit{chan.2.3.req}, & \mathit{chan}, \mathit{0.1.req} \end{array} \right\rangle$$

At this point both $\mathit{RACK.MGR}(0)$ and $\mathit{RACK.MGR}(2)$ are waiting for a message to arrive from another rack. But it is possible that $\mathit{RACK.MGR}(1)$ and $\mathit{RACK.MGR}(3)$ have both already committed to sending a message to each other, which would mean deadlock. Of course we have only considered a subnetwork of the system as a whole, so we need to check that this deadlock could still arise in the wider context. It is fairly obvious that this is indeed the case.

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. However this type of uncertainty could certainly not be tolerated in a safety critical application, such as an air traffic control system.

It is a simple matter to modify the definition of *RACK.MGR* to render the system deadlock-free, through adherence to the client-server protocol. This is achieved by splitting the process onto two levels, $\mathit{RACK.MGR}'$ and $\mathit{RACK.MGR}''$. Each lower level process $\mathit{RACK.MGR}'$ handles signals from the local panel manager as a server and also makes requests to any of the four higher level processes $\mathit{RACK.MGR}''$ as a client. The new CSP definitions are as follows.

$$\mathit{RACK.MGR}'(i) = \mathit{from.panel.mgr.i} \rightarrow \sqcap_{j=0}^{3} \mathit{req}.i.j \rightarrow \mathit{ack}.i.j \rightarrow \mathit{RACK.MGR}'(i)$$

$$\mathit{RACK.MGR}''(i) = \square_{j=0}^{3} \mathit{req}.j.i \rightarrow \mathit{ack}.j.i \rightarrow \mathit{RACK.MGR}''(i)$$

$$\mathit{RACKS}' = \left\langle \begin{array}{ll} \mathit{RACK.MGR}'(0), & \mathit{RACK.MGR}'(1), \\ \mathit{RACK.MGR}'(2), & \mathit{RACK.MGR}'(3), \\ \mathit{RACK.MGR}''(0), & \mathit{RACK.MGR}''(1), \\ \mathit{RACK.MGR}''(2), & \mathit{RACK.MGR}''(3) \end{array} \right\rangle$$

The client-server digraph of this improved design is given in figure 4.5. It is circuit-free which guarantees deadlock-freedom for the new system. It is notable that, as well as being deadlock-free, the new design is far simpler and somewhat more elegant. This shows how, far from being overly restrictive, design rules can enhance the creative process of parallel software design.

Figure 4.5: Client-Server Digraph for Improved Design