Chapter 2

Design Rules for Deadlock Freedom

Introduction

The problem of determining whether any given concurrent system can ever deadlock is similar to the famous halting problem of Turing machines – it is undecidable. This means that there can never be an algorithmic method for proving deadlock freedom which will work in the general case [Mairson 1989].

If the system consists only of finite-state processes then we can always theoretically check deadlock-freedom by exhaustive state analysis, but as the number of states of the system as a whole tends to be exponentially proportional to the number of processes this technique is only viable for very small networks.

The previous chapter details efficient proof techniques which will work in a wide variety of cases, but there is no guarantee that existing systems will be amenable to them in practice. What is needed is a set of rules which enable us to guarantee deadlock-freedom at the design stage before the major work of building the system has been done.

Here we describe three practical design paradigms which may be used for this purpose.

- Networks of *cyclic-ordered* processes: Each process behaves according to a fixed cyclic communication pattern. Useful for computationally intensive tasks, such as finite-element analysis or neural network simulation.
- Client-Server systems: Processes communicate according to a master-slave protocol. Applications include process farms and message routing systems.
- User-Resource systems: User processes compete for shared resources. Applicable to distributed databases and operating systems.

These rules have the joint advantages of being easy to use and also being backed up with mathematical rigour. We use the theoretical results of the previous chapter to prove them correct, and to show how they may be combined hierarchically. Used in this way they are suitable for the construction of a rich variety of concurrent systems.

2.1 Cyclic Processes

Many parallel applications consist of large arrays of simple processes, with fixed cyclic communications patterns. P. H. Welch discovered some deadlock-prevention rules for certain processes of this type [Welch 1987]. He presented these results informally in the context of the occam programming language. We shall now state and prove them in the formal context of CSP.

A process P is called *I/O-SEQ* if it operates cyclically such that, once per cycle, it communicates on a finite set of input channels I in parallel, then it communicates on a finite set of output channels O also in parallel.

Abstracting away any data that is passed, we can write a *I/O-SEQ* process, with input channel set I, and output channel set O with the following CSP equation.

$$\begin{aligned} \mathit{I/O\text{-}SEQ}(I, O) &= (\ |||_{c:I}\ c \rightarrow \mathit{SKIP})\ ;\ (\ |||_{d:O}\ d \rightarrow \mathit{SKIP})\ ;\ \mathit{I/O\text{-}SEQ}(I, O) \\ \alpha \mathit{I/O\text{-}SEQ}(I, O) &= I \cup O \end{aligned}$$

A process which communicates on all its channels in parallel on every cycle is called *I/O-PAR*. In CSP we write it like this

$$\begin{aligned} \mathit{I/O\text{-}PAR}(I, O) &= (\ |||_{c:I\cup O}\ c \rightarrow \mathit{SKIP})\ ;\ \mathit{I/O\text{-}PAR}(I, O) \\ \alpha \mathit{I/O\text{-}PAR}(I, O) &= I \cup O \end{aligned}$$

When *I/O-PAR* and *I/O-SEQ* processes are combined in a network we observe the *I/O* convention. Recall that this means that a channel may be used by at most two processes, one for input and the other for output. The *connection digraph* of a network of *I/O-PAR* and *I/O-SEQ* processes is constructed in the following way. A vertex is used to represent each process and an arc is used to represent each shared channel, directed from the process for which it is an output channel towards the process for which it is an input channel. A sequence of channels which forms a path in the connection diagram of a network is called a *data-flow path* ; a sequence of channels which forms a circuit is called a *data-flow circuit*

These processes may be composed in ways which guarantee deadlock freedom according to some simple design rules.

Rule 1 (Welch 1987) *Any network of I/O-PAR processes is deadlock-free.*

In other words, any network constructed exclusively from *I/O-PAR* components, no matter how large will never deadlock.

Rule 2 (Welch 1987) *A connected network of I/O-SEQ processes is deadlock-free if, and only if, it has no data-flow circuits.*

Rule 3 (Welch 1987) *A connected network of I/O-SEQ and I/O-PAR processes is free of deadlock if, and only if, it has no data-flow circuits which pass through only I/O-SEQ processes.*

We shall now prove the correctness of these rules using theorem 2 (page 29). Note that rules 1 and 2 are corollaries of rule 3, so it is only necessary to prove the last result.

Proof. Let $V = \langle P_1, .., P_n \rangle$ be a connected network of *I/O-SEQ* and *I/O-PAR* processes. Then for each maximal failure (s, X) of P_i we define a variant function, $f_i((s, X))$, which calculates the number of complete cycles of *I/O* operations that P_i has completed after trace s. This is given by

$$f_i((s,X)) = \left\lfloor \frac{|s|}{|\alpha P_i|} \right\rfloor$$

From the definitions we can deduce that a process in this network can never be waiting for an *I/O-SEQ* process which has performed more cycles than it has, and can only be waiting to communicate with an *I/O-PAR* process which has performed less cycles than it has. So let σ be a state $(s, \langle X_i, X_j \rangle)$ of the subnetwork $\langle P_i, P_j \rangle$. Then if P_j is *I/O-SEQ*

$$P_i \overset{\sigma,\Lambda}{\rightarrow} \bullet P_j \Longrightarrow f_i((s \upharpoonright \alpha P_i, X_i)) \geq f_j((s \upharpoonright \alpha P_j, X_j))$$

but if P_j is *I/O-PAR*

$$P_i \overset{\sigma,\Lambda}{\rightarrow} \bullet P_j \Longrightarrow f_i((s \upharpoonright \alpha P_i, X_i)) > f_j((s \upharpoonright \alpha P_j, X_j))$$

Suppose that V has a deadlock state σ, then by theorem 2 there must be a cycle of ungranted requests in state σ such that the variant function of each process is the same. It follows from the above observations that all the processes in the cycle of ungranted requests must be *I/O-SEQ*. Each of these processes must be waiting for input from its successor in the cycle, so the cycle of ungranted requests corresponds to a data-flow circuit (in the opposite direction) passing through only *I/O-SEQ* processes.

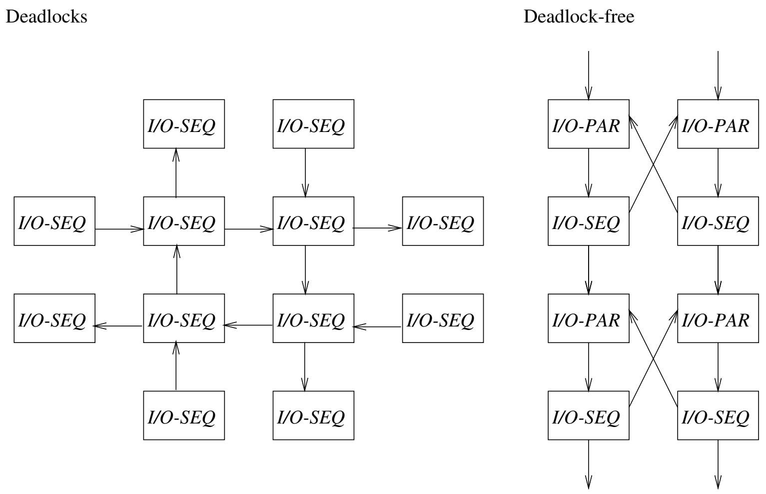
Otherwise suppose that the network contains a data-flow circuit through *I/O-SEQ* processes. Each process on this circuit is bound to come to a halt during its first cycle to wait forever for input from its predecessor. No process in the network can ever advance more than one cycle beyond any of its neighbours in the connection digraph, so deadlock will eventually ensue because the network is connected□

Figure 2.1 illustrates examples of networks constructed from *I/O-SEQ* and *I/O-PAR* elements. One of these has a data-flow circuit passing exclusively through *I/O-SEQ* processes and so deadlocks; the other has no such circuit and so is deadlock-free.

Composite Processes

Sometimes we may build a component from *I/O-SEQ* and *I/O-PAR* processes, and then wish to replicate it many times in a larger system. The next rule describes how, in the

Figure 2.1: Networks of *I/O-SEQ* and *I/O-PAR* Processes



right circumstances, we may treat such a component as a single process for the purpose of deadlock analysis. We shall start with some new definitions.

If a connected network, V, of *I/O-SEQ* processes has no data-flow circuits we say that *PAR*(V) is a *composite-I/O-SEQ* process.

The input and output channels of *PAR*(V) are taken to be those channels which do not belong to the vocabulary of V and so are used by only a single process. We call these the *external* channels of V.

If a connected network, V, of *I/O-SEQ* and *I/O-PAR* components, has neither a data-flow circuit, passing through only *I/O-SEQ* processes, nor a data-flow path from an *I/O-SEQ* process with an external input channel to an *I/O-SEQ* process with an external output channel, passing through only *I/O-SEQ* processes, we say that *PAR*(V) is a *composite-I/O-PAR* process.

We find that Welch's rules generalise to composite processes as follows.

Rule 4 (Welch 1987) *A connected network* V *of composite-I/O-SEQ and composite-I/O-PAR processes is deadlock-free if, and only if, it has no data-flow circuits which pass through only composite-I/O-SEQ processes.*

Proof. Let V' be the network of *I/O-PAR* and *I/O-SEQ* processes that may be derived from V by breaking each process down into its basic components. This rule follows from rule 3 by proving that V contains a data-flow circuit through *composite-I/O-SEQ* processes if, and only if, V' contains a data-flow circuit through *I/O-SEQ* processes.

In graph-theoretic terms the connection diagram of V is a *contraction* of that of V' (see appendix B). Suppose that V contains a data-flow circuit through only *composite-I/O-SEQ* processes, then, clearly, V' contains a data-flow circuit passing only through *I/O-SEQ* processes.

Alternatively, suppose that a data-flow circuit, C, is contained within V', passing only through *I/O-SEQ* processes. Under the contraction of connection diagrams from V' to V, C maps either to a directed *closed trail* of V, or to a single vertex. (Note that a closed trail differs from a circuit in that its vertices are not necessarily distinct – it may 'cross' itself.) The latter option may be eliminated immediately as it implies the presence of a data-flow circuit within a composite process, which is prohibited by definition. The former option implies that V contains a directed closed trail through *composite-I/O-SEQ* processes, because there can be no path through a *composite-I/O-PAR* process that does not cross a simple *I/O-PAR* element. Any directed closed trail of V entails at least one circuit.

So V contains a data-flow circuit through *composite-I/O-SEQ* processes, if and only if V' contains a data-flow circuit through *I/O-SEQ* processes, and the required result may now be deduced from rule 3□

It is useful to note that basic *I/O-SEQ* and *I/O-PAR* processes are also *composite-I/O-SEQ* and *composite-I/O-PAR* respectively. This enables us to build a deadlock-free network from a mixture of basic and composite processes.

Example – Emulating VLSI Circuits

Welch originally formulated these design rules in order to emulate VLSI circuits, using the occam programming language. He used rule 4 to construct various 'circuits' hierarchically. For example, a 'latch' component is shown in figure 2.2. This is built from two *I/O-PAR* 'nand' gates and two *I/O-SEQ* 'delta' processes (which simply duplicate their input signal). The latch component is *composite-I/O-PAR*.

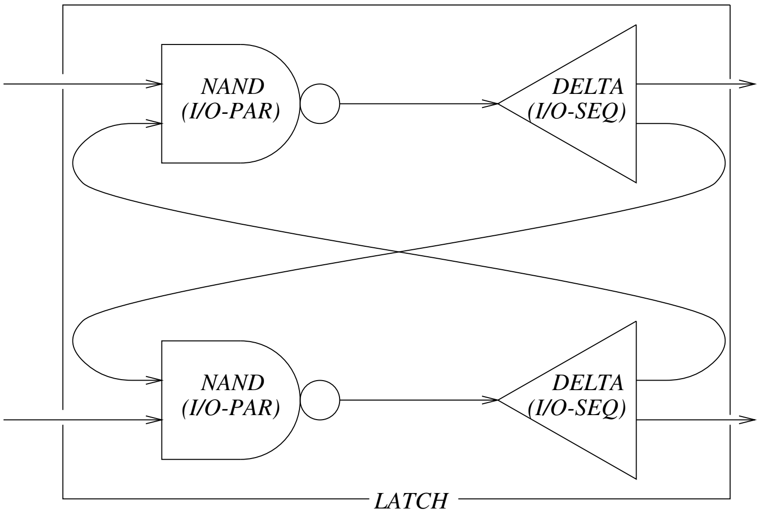
Welch used this technique to predict the behaviour of complex electronic circuits prior to their realisation in silicon. He was able to construct deadlock-free networks with hundreds of thousands of processes, using design rules 1 to 4. These rules have subsequently been used for many other applications by occam programmers. (*e.g.* See [Macfarlane 1992].) Rules 1 and 2 were also reported in [Roscoe and Dathi 1986].

A General Rule

Dijkstra and Scholten developed a rule for cyclic processes which communicate exactly once with each of their neighbours on each cycle in fixed sequence [Dijkstra 1982]. This was extended by Roscoe and Dathi to allow sets of communications to be performed in parallel, as with Welch's rules. Here we generalise all these results to produce a partial order based rule.

A *cyclic-PO* process is a process P with a finite set of communication channels C, which operates cyclically, communicating on each of its channels once per cycle. The order of communication is governed by a *strict partial order* $(C, >)$, whereby P becomes ready to communicate on a channel c for the nth time, once it has completed

Figure 2.2: *LATCH*: a Composite *I/O-PAR* Process



its $(n-1)$th cycle, and has communicated on all the channels below c by $>$ on its nth cycle. This can be defined formally as follows.

$$
\begin{aligned}
\textit{CYCLIC-PO}(C,>) &= C2(C,\{\},>)\\
C2(C,\textit{DONE},>) &= C2(C,\{\},>)\\
&\quad \triangleleft \textit{DONE} = C \triangleright\\
&\quad \Box_{x:\textit{mins}(C-\textit{DONE},>)}\ x \rightarrow C2(C,\textit{DONE}\cup\{x\},>)
\end{aligned}
$$

$$\alpha\textit{CYCLIC-PO}(C,>) = C$$

Where *mins*$(Y,>)$ is defined as the minimal elements of subset Y of C, given by

$$\textit{mins}(Y,>) = \{y \in Y \mid \nexists z \in Y.\quad y > z\}$$

Now we consider a network of cyclic-PO processes, $V = \langle P_1 \ldots P_N\rangle$, where

$$P_i = \textit{CYCLIC-PO}(\alpha P_i, >_i)$$

The set of communication channels of the network as a whole, $\bigcup_{i=1}^{N} \alpha P_i$, is called αV. We use symbol $\triangleright$ to represent the aggregate of the various partial orderings, $>_i$, *i.e.*

$$c_i \triangleright c_j \iff \exists k.\quad c_i >_k c_j$$

The direction of data-flow along communication channels, if any, is irrelevant to the deadlock properties of cyclic-PO networks. Sometimes it is meaningless to assign any direction to a channel. For this reason we shall here consider the *connection graph* of a network rather than the connection digraph. This is constructed in the same way except that it is undirected.

An *undirected, closed trail of dependent channels* is a sequence of channels of αV, $\langle c_1 \,..\, c_m \rangle$, which forms a closed trail in the connection graph of V (see appendix B for definitions), and satisfies

$$c_1 \rhd c_2 \rhd .. \rhd c_m \rhd c_1$$

Theorem 7 *A connected network of cyclic-PO processes is deadlock-free if, and only if, it has no undirected, closed trail of dependent channels.*

Proof. Suppose there exists an undirected, closed trail of dependent channels, such that

$$c_1 \rhd c_2 \rhd .. \rhd c_m \rhd c_1$$

No communication can ever take place on any of these channels, so the processes they are connected to will never complete their first *I/O* cycle. No cyclic-PO process can ever have advanced more than one *I/O* cycle beyond its neighbours in the connection graph of V, so there is a limit to the number of events that any component process can execute. Hence deadlock will eventually ensue.

Now suppose instead that we have arrived at a deadlock state σ of V. Every process is unable to proceed, and has at least one ungranted request (with respect to Λ).

Consider any ungranted request $P_{i_1} \overset{\sigma,\Lambda}{\to} \bullet P_{i_2}$, where P_{i_1} wants to communicate on some channel c_1 for the n_1th time, but P_{i_2} is refusing to participate. Either P_{i_2} and P_{i_1} have both completed the same number of *I/O* cycles, but P_{i_2} has not yet communicated on all its channels below c_1 by $\rhd$ on the current cycle, or P_{i_2} has completed one less *I/O* cycle than P_{i_1}. It follows that P_{i_2} is waiting to communicate on some channel c_2 for the n_2th time, where either $(n_1 = n_2) \wedge (c_1 \rhd c_2)$ or $n_1 > n_2$.

We can repeat this argument to construct an arbitrarily long sequence of pairs

$$(c_1, n_1), (c_2, n_2), (c_3, n_3) \,..$$
$$\text{Where} \quad \forall\, i. \quad ((n_i = n_{i+1}) \wedge (c_i \rhd c_{i+1})) \vee (n_i > n_{i+1})$$

The channels of this sequence correspond to a walk in the underlying graph of V.

The decreasing sequence $n_1, n_2, n_3 ..$ must have a limit, *i.e.*

$$\exists\, p. \quad \forall\, j \geq p. \quad n_j = n_p$$
$$\text{Hence} \quad c_p \rhd c_{p+1} \rhd c_{p+2} \rhd \,..$$

As αV is finite, this sequence must eventually repeat a term, *i.e.*

$$\exists\, q, r. \quad c_{p+q} \rhd c_{p+q+1} \rhd .. \rhd c_{p+q+r} = c_{p+q}$$
$$\text{where } c_{p+q}, .., c_{p+q+r-1} \text{ are all distinct.}$$

This sequence is represented by a closed trail in the connection graph of $V\Box$

This theorem describes the deadlock properties of networks of cyclic processes in general. If each process can complete its first *I/O* cycle the network will never deadlock.

It is worth mentioning a special case of cyclic-PO processes. We define a *cyclic-LOP* process to be a cyclic-PO process where $(\alpha P, >)$ takes the form of a *linearly ordered partition*. This means that αP is partitioned into subsets $\lambda_1, .., \lambda_m$ such that

$$\begin{array}{c} \forall\, c_1 : \lambda_1 . \quad \forall\, c_2 : \lambda_2 . \quad .. \quad \forall\, c_m : \lambda_m . \qquad c_1 > c_2 > .. > c_m \\ \forall\, i : \{1, .., m\}. \quad \forall\, c, c' : \lambda_i \qquad \neg c > c' \end{array}$$

The *I/O-PAR* and *I/O-SEQ* cyclic processes, defined by Welch, both have *cyclic-LOP* communication patterns. The $>$ relation is empty for an *I/O-PAR* process. For an *I/O-SEQ* process, $c_i > c_j$ if and only if c_i is an output channel and c_j is an input channel. For a network, V, of cyclic-LOP processes we can derive a result with a simpler topological requirement than for cyclic-PO processes. This is a slight extension of a theorem due to Roscoe and Dathi.

An *undirected, circuit of dependent channels* is a sequence of channels of αV, $\langle c_1 .. c_m \rangle$, which forms a circuit in the connection graph of V, and satisfies

$$c_1 \rhd c_2 \rhd .. \rhd c_m \rhd c_1$$

Theorem 8 *A connected network consisting of cyclic-LOP processes is free of deadlock if, and only if, it has no undirected circuit of dependent channels*

This is proved in virtually the same manner as Welch's rules. In a deadlock state of a network of cyclic-LOP processes, there must be a cycle of ungranted requests where each process has performed the same number of *I/O* cycles. The crucial observation is that if P_i has an ungranted request to P_j, trying to perform some event c and both processes have performed the same number of *I/O* cycles then *every* event that P_j is ready to perform is beneath c in the partial ordering $>_j \Box$

The result that Roscoe and Dathi proved was the same as this except that it enforced the extra restriction that at most one channel be permitted between any two processes

These theorems may be too complicated to be considered design rules in their own right, however a suite of design rules for computationally intensive parallel systems can be derived. For instance Welch's rules drop out as simple corollaries. Here is an example of a new design rule.

Rule 5 *A connected network of cyclic-PO processes is deadlock-free if, and only if, there exists a labelling of the connection graph, given by* $l : \alpha V \rightarrow \mathrm{N}$*, which satisfies*

$$c_i \rhd c_j \Longrightarrow l(c_i) > l(c_j)$$

Proof. We use the technique of *reductio ad absurdum*. Suppose the conditions of the rule hold, and yet V can deadlock. Then, by theorem 7, there is a sequence of channels satisfying

$$\begin{aligned} & c_1 \rhd c_2 \rhd .. \rhd c_m \rhd c_1 \\ \Longrightarrow\ & l(c_1) > l(c_2) > ..l(c_m) > l(c_1) \\ \Longrightarrow\ & l(c_1) > l(c_1)\ \sharp \end{aligned}$$

Conversely if V is deadlock-free, by the nature of its construction it must have a trace s of finite length which includes every element of αV. We label each element of αV according to the position of its first appearance in s to derive a labelling which satisfies the conditions of the theorem. This completes the proof□

To design a network using this rule, we first draw a connection graph (or digraph if we prefer) and label each channel with a numeric value, representing a logical order. Then if each process is implemented as a cyclic-PO process, capable of communicating on its channels in order of increasing value, the network is deadlock-free. (When a process has more than one channel of the same value, it should be implemented to communicate in parallel on those channels.)

Example – A Toroidal Cellular Automaton

To demonstrate this approach, we consider a 4×4 cellular automaton program, where each cell compares its state with those of its four neighbours in strict, clockwise order. This is based on a program described in [Dewdney 1989]. The idea is that each cell maintains an integer state and whenever it finds that its state is exactly one less than that of a neighbour, it changes state to match. (All comparisons are done using modulo arithmetic.). When a large grid is used some interesting patterns evolve.

Blind to the risk of deadlock we might give each cell process an identical communication pattern, such as defined by the following processes where each cell communicates with its neighbours in the order left, up, right, then down.

$$\begin{aligned} \mathit{CELL}(i,j) &= \mathit{LEFT}(i,j) \\ \mathit{LEFT}(i,j) &= (e.i.j.\mathit{left} \rightarrow \mathit{SKIP} \mid\mid\mid e.(i-1).j.\mathit{right} \rightarrow \mathit{SKIP})\ ;\ \mathit{UP}(i,j) \\ \mathit{UP}(i,j) &= (e.i.j.\mathit{up} \rightarrow \mathit{SKIP} \mid\mid\mid e.i.(j-1).\mathit{down} \rightarrow \mathit{SKIP})\ ;\ \mathit{RIGHT}(i,j) \\ \mathit{RIGHT}(i,j) &= (e.i.j.\mathit{right} \rightarrow \mathit{SKIP} \mid\mid\mid e.(i+1).j.\mathit{left} \rightarrow \mathit{SKIP})\ ;\ \mathit{DOWN}(i,j) \\ \mathit{DOWN}(i,j) &= (e.i.j.\mathit{down} \rightarrow \mathit{SKIP} \mid\mid\mid e.i.(j+1).\mathit{up} \rightarrow \mathit{SKIP})\ ;\ \mathit{LEFT}(i,j) \\ \alpha\mathit{CELL}(i,j) &= \left\{ \begin{array}{cccc} e.i.j.\mathit{left}, & e.i-1.j.\mathit{right}, & e.i.j.\mathit{up}, & e.i.j-1.\mathit{down} \\ e.i.j.\mathit{right}, & e.i+1.j.\mathit{left}, & e.i.j.\mathit{down}, & e.i.j+1.\mathit{up} \end{array} \right\} \end{aligned}$$

In this process definition all integer arithmetic is *modulo 4*. The network is given by

$$\langle \mathit{CELL}(0,0),..,\mathit{CELL}(0,3),..,\mathit{CELL}(3,0),..,\mathit{CELL}(3,3)\rangle$$

This arrangement leads to immediate deadlock because there exist many undirected, closed trails of dependent channels. We tackle this problem by labelling each channel of the network, and then recoding each process to communicate on its channels according to the ascending order of its labels. The labelling scheme shown in figure 2.3 allows each component to communicate in strict clockwise order as required. But cells alternate as to whether to start by communicating on the left or on the right. This gives us a new definition for *CELL* as follows

$$\mathit{CELL}(i,j) = \mathit{LEFT}(i,j) \triangleleft ((i+j) \quad \mathit{modulo}\ 2 = 0) \triangleright \mathit{RIGHT}(i,j)$$

An implementation of this network, programmed in occam2, is given in [Martin *et al* 1994].

In practice it would be desirable to add extra channels to this network to monitor the state of each cell, and reset the system when required. Use of the cyclic-PO paradigm would require that each channel be used on every *I/O* cycle, which might be unnecessary. In the next section theorem 7 will be extended to allow processes to communicate on a subset of their channels on any given *I/O* cycle (as long as neighbouring processes are in agreement as to which channels are to be used), and also to allow the channel ordering to be changed between successive cycles.

Multi-phase Communication Patterns

A *multi-phase-PO* process is a deadlock-free process, P, with a set of communication channels, αP, which operates cyclically, communicating once on a predefined subset of its channels on each cycle. On its kth cycle, P communicates according to a partial order

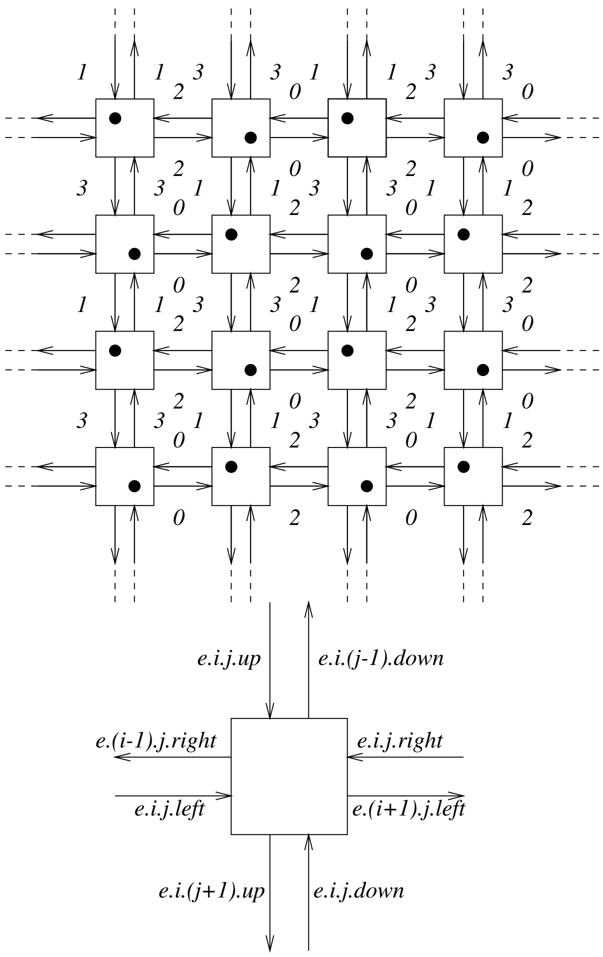
$$(\alpha^{(k)}P, >^{(k)})$$

where $\alpha^{(k)}P \subseteq \alpha P$: P communicates on channel c on its kth cycle if and only if $c \in \alpha^{(k)}P$, in which case it becomes ready to do so once it has completed its $(k-1)$th cycle, and has communicated on all the channels of $\alpha^{(k)}P$ below c by $>^{(k)}$ on its kth cycle□

We say that a network of *multi-phase-PO* processes, $V = \langle P_1 \,..\, P_N\rangle$, is *concordant* if neighbouring processes agree on which subset of channels to use on each *I/O* cycle:

$$\forall\, k : \mathrm{N} - \{0\}. \quad \forall\, i,j : \{0,1,..N\}. \quad \alpha^{(k)}P_i \cap \alpha P_j = \alpha P_i \cap \alpha^{(k)} P_j$$

Figure 2.3: Connection Digraph with Channel Labelling



Theorem 9 *A connected, concordant network of multi-phase-PO processes is free of deadlock if, and only if,* $\forall\, k\ :\ \mathrm{N} - \{0\}$ *there is no undirected, closed trail of* $\rhd^{(k)}$*-dependent channels*$\Box$

The proof of this is virtually identical to that of theorem 7, and we can derive a similar design rule.

Rule 6 *If there exists a partial labelling of the channels of a network of multi-phase-PO processes, for every I/O cycle, given by the partial functions* $l_k : \alpha\, V \mapsto \mathrm{N}$*, which satisfies*

$$c_i \rhd^{(k)} c_j \Longrightarrow l_k(c_i) > l_k(c_j)$$
$$\forall\, i : \{1, .., N\}. \quad \alpha P_i \cap \mathit{domain}(l_k) = \alpha^{(k)} P_i$$

then the network is deadlock-free$\Box$

Figure 2.4 illustrates how this rule may be used to add a control process to the toroidal cellular automaton. In this design each cell communicates bidirectionally with the control process after the completion of every fourth cycle of communication with its neighbours.

2.2 Client-Server Protocol

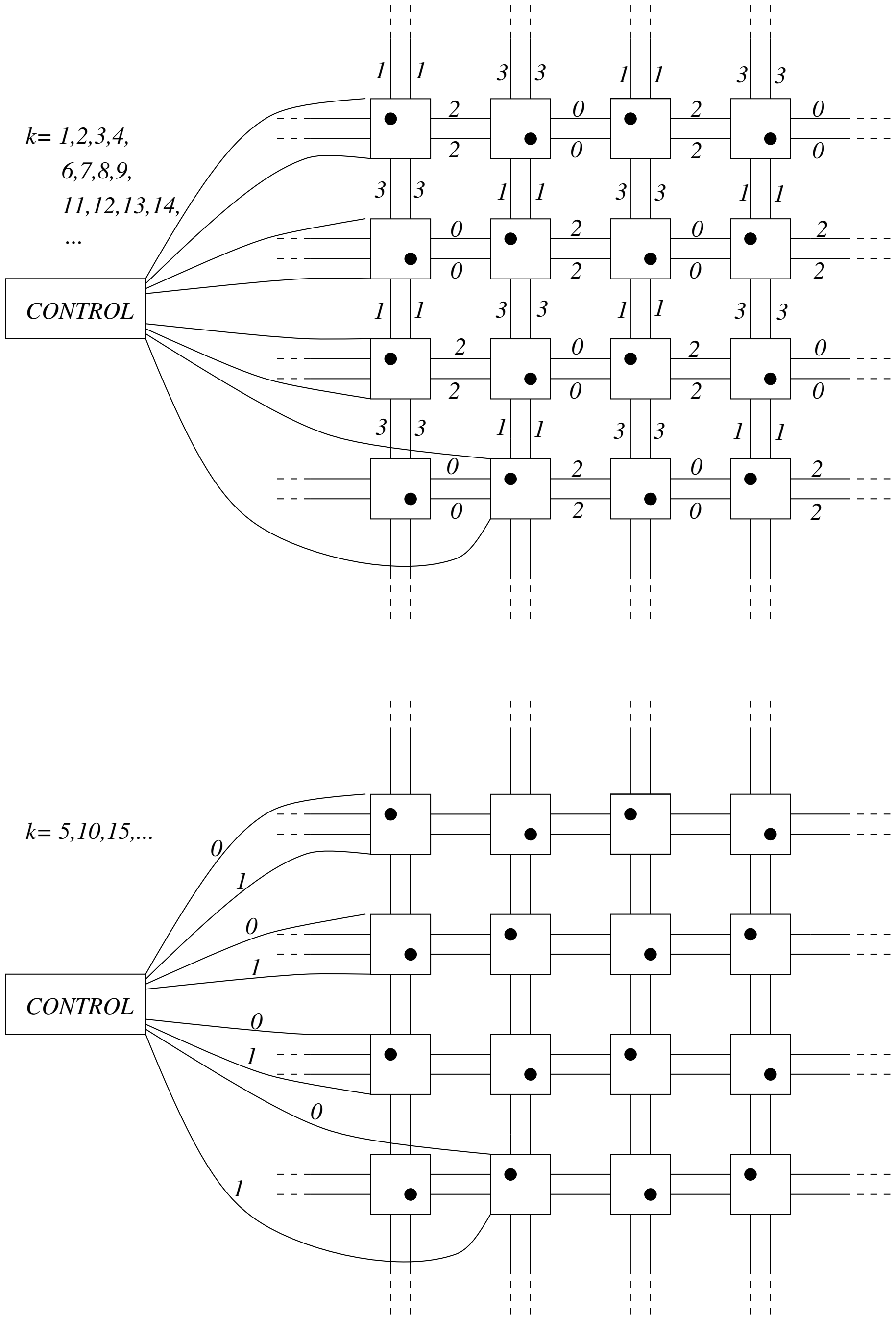
The cyclic paradigm may be used effectively to solve many common problems in parallel computing. However for certain problems it is too restrictive in the respect that it enforces a pre-determined communication pattern. In practice, we often need to allow the communication patterns of processes to vary according to external requirements. A more flexible design rule from this perspective is the *Client-Server Protocol*. This was originally formulated by P. Brinch Hansen in the context of operating systems [Brinch Hansen 1973]. It has since been adapted by Welch, G. R. R. Justo and C. J. Willcock as a means of designing deadlock-free concurrent systems using occam [Welch *et al* 1993]. The version of the protocol presented here is a formal adaptation and extension of the ideas of these authors, which were stated informally.

The main requirement is that processes communicate on each one of their channels either as a *client* or as a *server*, according to a strict protocol. A network of client-server processes is deadlock-free if it has no cycle of client-server relationships.

A *basic client-server* CSP process P has a finite set of external channels partitioned into separate *bundles*, each of which has a type, in relation to P, which is either *client* or *server*. Each channel bundle consists of *either* a pair of channels, a *requisition* and an *acknowledgement*, $\langle r, a\rangle$, *or* a single channel (which we call a *drip*) $\langle d\rangle$. (This allows client-server conversations to be either one way or two way). We write the set of client bundles of P as *clients*(P), and the server bundles as *servers*(P).

In the subsequent analysis, the event of communication on a channel is again represented purely by the channel name, ignoring any data that is passed. The purpose of

Figure 2.4: Multi-phase Channel Labelling



this is clarity and simplicity. Following this convention, a basic client-server process, P, must obey the following rules

(a) P is divergence-free, deadlock-free and non-terminating.

$$\forall (s, X) : \mathit{failures}(P). \quad X \neq \Sigma$$

(b) When P is in a stable state (no internal activity possible), *either* it is ready to communicate on all its *requisition* and *drip* server channels *or* it is ready to communicate on none of them. In CSP terms this means that *maximal* refusal sets of P include either all the requisition and drip server channels or none of them, *i.e.*

$$\begin{array}{c} \forall (s, X) : \mathit{failures}(P). \quad X \quad \text{maximal} \implies \\ \left(\begin{array}{l} (\forall \langle d \rangle : \mathit{servers}(P). \quad d \notin X) \;\wedge \\ (\forall \langle r, a \rangle : \mathit{servers}(P). \quad r \notin X) \end{array} \right) \vee \\ \left(\begin{array}{l} (\forall \langle d \rangle : \mathit{servers}(P). \quad d \in X) \;\wedge \\ (\forall \langle r, a \rangle : \mathit{servers}(P). \quad r \in X) \end{array} \right) \end{array}$$

(c) P always communicates on any bundle pair $\langle r, a \rangle$, in the sequence $r, a, r, a, ..$, *i.e.*

$$\forall \langle r, a \rangle : \mathit{clients}(P) \cup \mathit{servers}(P). \quad \forall\, s : \mathit{traces}(P). \quad 1 \geq (s \downarrow r - s \downarrow a) \geq 0$$

(d) When P communicates on a client *requisition* channel, it must guarantee to accept the corresponding *acknowledgement*, *i.e.*

$$\forall \langle r, a \rangle : \mathit{clients}(P). \quad \forall (s, X) : \mathit{failures}(P). \quad s \downarrow r > s \downarrow a \implies (a \notin X)$$

When we construct a *client-server* network V from a set of client-server processes $\langle P_1, ..P_N \rangle$, each client bundle of a process must *either* be a server bundle of exactly one other process, *or* consist of channels external to the network. Similarly each server bundle of any process must either be a client bundle of exactly one other process or be external to the network. No other communication between processes is permitted, *i.e.*

$$\begin{array}{c} \forall\, i \in \{1, ..N\}. \quad \forall\, s \in \mathit{clients}(P_i) \\ \text{Either} \quad \exists! j. \quad j \neq i \wedge s \in \mathit{servers}(P_j) \\ \text{Or let} \quad s = \langle s_1, .., s_k \rangle \quad \text{then} \quad \{s_1, .., s_k\} \cap \Lambda = \{\} \end{array}$$

$$\begin{array}{c} \forall\, i \in \{1, ..N\}. \quad \forall\, s \in \mathit{servers}(P_i) \\ \text{Either} \quad \exists! j. \quad j \neq i \wedge s \in \mathit{clients}(P_j) \\ \text{Or let} \quad s = \langle s_1, .., s_k \rangle \quad \text{then} \quad \{s_1, .., s_k\} \cap \Lambda = \{\} \end{array}$$

$$\begin{array}{ll} i \neq j \implies & \text{Let} \quad (\mathit{clients}\, P_i \cap \mathit{servers}\, P_j) \cup (\mathit{clients}\, P_j \cap \mathit{servers}\, P_i) = \\ & \quad \{\langle s_{1,1}, .., s_{1,k_1} \rangle, .., \langle s_{m,1}, .., s_{m,k_m} \rangle\} \\ & \text{Then} \quad \alpha P_i \cap \alpha P_j = \{s_{1,1}, .., s_{1,k_1}, .., s_{m,1}, .., s_{m,k_m}\} \end{array}$$

The *client-server digraph* of a client-server network consists of a vertex representing every process, and, an arc representing each shared bundle, directed from the process for which it is of type client, towards that for which it is of type server.

Rule 7 (Client-Server Theorem) *A client-server network, composed from basic processes, which has a circuit-free client-server digraph, is deadlock-free.*

Proof. First we observe that the matching requirements for client and server bundles within a network enforce triple-disjointedness within a client-server network. Rule **(a)** ensures that basic client-server networks are also busy.

Let V be a client-server network, composed from basic processes, the client-server digraph of which contains no circuit. Suppose it has a deadlock state σ. There must be a cycle of ungranted requests in state σ by theorem 1 (page 29). Because the client-server digraph is circuit-free this cycle of ungranted requests cannot consist entirely of requests from client to server or vice-versa. It must contain a subsequence

$$P_i \overset{\sigma,\Lambda}{\rightarrow} \bullet P_j \overset{\sigma,\Lambda}{\rightarrow} \bullet P_k$$

where P_i communicates with P_j as client to server and P_j communicates with P_k as server to client. (Note that if the cycle of ungranted requests has length only two then P_i and P_k are the same process.)

We shall now show that the basic client-server protocol renders this situation impossible. First we note that by rules **(c)** and **(d)** P_j can only be waiting to communicate with P_k on a server requisition or drip channel; an acknowledgement is never refused. Hence P_j is ready to communicate on all its server requisition and drip channels by rule **(b)**. So P_i must be waiting for an acknowledgement from P_j. However, by rule **(c)**, P_j must have already acknowledged every previous requisition event in order to be ready to communicate on all its requisition channels. So P_i cannot have an ungranted request to P_j after all. This contradiction proves that the system has no deadlock state$\Box$.

Example – A Simple Process Farm

We consider an application where computing-intensive tasks are performed in parallel using a standard farm network configuration. A farmer employs n foremen each of whom is responsible for m workers. When a worker process becomes idle it reports the result of any work done to its foreman, using channel $a.i.j$, where j denotes worker and i denotes foreman. The foreman reports this on channel $c.i$ to the farmer who, in turn, replies with a new task using channel $d.i$, The foreman then assigns the new task to the worker with channel $b.i.j$. Here the relationship between worker and foreman and the relationship between foreman and farmer are both client to server.

The CSP communication patterns of the component processes are given as follows.

$$\mathit{FARMER} \;=\; \Box_{i=0}^{n-1}\; c.i \rightarrow d.i \rightarrow \mathit{FARMER}$$

$$\begin{array}{rcl}
\mathit{clients}(\mathit{FARMER}) & = & \{\} \\
\mathit{servers}(\mathit{FARMER}) & = & \{\langle c.0, d.0\rangle, .., \langle c.(n-1), d.(n-1)\rangle\} \\
\alpha\mathit{FARMER} & = & \{c.0, .., c.(n-1), d.0, .., d.(n-1)\rangle\}
\end{array}$$

$$\mathit{FOREMAN}(i) \;=\; \Box_{j=0}^{m-1}\; a.i.j \rightarrow c.i \rightarrow d.i \rightarrow b.i.j \rightarrow \mathit{FOREMAN}(i)$$

$$\begin{array}{rcl}
\mathit{clients}(\mathit{FOREMAN}(i)) & = & \{\langle c.i, d.i\rangle\} \\
\mathit{servers}(\mathit{FOREMAN}(i)) & = & \{\langle a.i.0, b.i.0\rangle, .., \langle a.i.(m-1), b.i.(m-1)\rangle\} \\
\alpha\mathit{FOREMAN}(i) & = & \{a.i.0, .., a.i.(m-1), b.i.0, .., b.i.(m-1), c.i, d.i\}
\end{array}$$

$$\mathit{WORKER}(i,j) \;=\; a.i.j \rightarrow b.i.j \rightarrow \mathit{WORKER}(i,j)$$

$$\begin{array}{rcl}
\mathit{clients}(\mathit{WORKER}(i,j)) & = & \{\langle a.i.j, b.i.j\rangle\} \\
\mathit{servers}(\mathit{WORKER}(i,j)) & = & \{\} \\
\alpha\mathit{WORKER}(i,j) & = & \{a.i.j, b.i.j\}
\end{array}$$

It is straightforward to verify that each process obeys the basic client-server protocol. The client-server digraph is illustrated in figure 2.5. It has no circuits hence the network is guaranteed deadlock-free.

Polling on a Channel

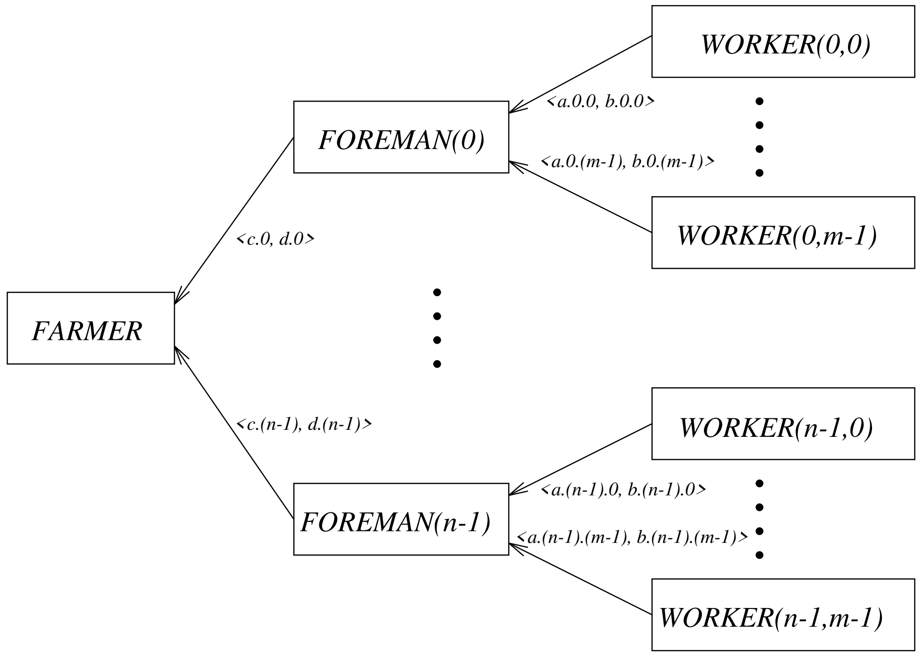
The technique of *polling* on a channel is a means by which a process can attempt to communicate on a channel without the risk of becoming blocked. In high level implementation languages this is achieved by the use of time-outs, possibly of zero duration. The version of CSP that we are using is untimed so there is no direct equivalent to this. However polling may still be represented using the available syntax. Consider the process

$$(\mathit{in} \rightarrow P \;\Box\; \mathit{timeout} \rightarrow Q) \setminus \mathit{timeout}$$

This process cannot become blocked trying to communicate on channel *in*, because it is always able to perform the internal event *timeout*.

While a process is attempting to poll a channel its state is unstable. Note that rule **(b)** of the basic client-server definition only applies to stable states. This means that the restriction that a process must either offer its services to all its clients or none of them at a given time may be overcome if polling is used. (However one has to be very careful in order to avoid introducing divergence.) An example of using polling in a client-server network is given in [Martin and Welch 1996].

Figure 2.5: Client-Server Digraph for *FARM*



Composite Processes

A *composite* client-server process *V* is a client-server network $\langle P_1, .., P_N \rangle$ composed solely from basic client-server processes, of which the client-server digraph contains no circuits; we define

$$\begin{aligned} clients(V) &= \left(\bigcup_{i=1}^{N} clients(P_i) - \bigcup_{j=1}^{N} servers(P_j) \right) \\ servers(V) &= \left(\bigcup_{i=1}^{N} servers(P_i) - \bigcup_{j=1}^{N} clients(P_j) \right) \end{aligned}$$

In other words the client and server bundles of V are those of the component processes P_i which are not paired off.

We represent a composite client-server process by a single vertex in a client-server digraph. The following result shows that this is consistent with the composition rule governing basic processes.

Rule 8 (Client-Server Closure) *A client-server network, composed from composite processes, with a circuit-free client-server digraph, is deadlock-free.*

Proof. Starting with a network such as described in the statement of the theorem with client-server digraph D, consider the client-server digraph D' of the network which is derived when each composite process is separated back into its basic components. Digraph D is a contraction of D'. Suppose that D' contains a circuit. By definition this cannot be local to a single composite process, and so it must map onto a closed trail in D. But as D has no circuit it has no closed trail either – a contraction. So D' has no circuit and the result follows from rule 7□

It is important to note that any basic client-server process is itself composite client-server (although the reverse is not true). Hence we can apply the result to mixtures of composite and basic processes. This rule is clearly useful for designing networks hierarchically. Complex subnetworks may be reused with ease. Black-box processes, that have been shown to abide by the composite client-server specifications, may be safely incorporated.

However the rule is too weak in some circumstances, as we shall demonstrate below. We need to find a generalisation.

We define a dependence relationship $\gg$ between server bundles and client bundles of a composite client-server process V as follows: if $x \in$ *servers*(V) and $y \in$ *clients*(V) then $x \gg y$ means that there is a path from the process with server bundle x to that with client bundle y, in the client-server digraph of V.

Figure 2.6: Composite Client-Server Process

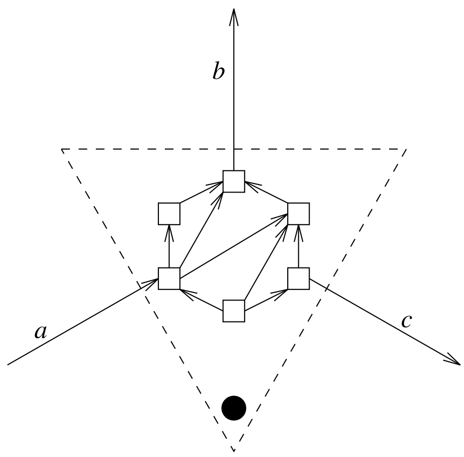


Figure 2.6 shows a hypothetical composite client-server process *BLACKBOX*, with external client-server channel bundles a, b, and c. Here we have

$$\begin{aligned} \textit{servers}(\textit{BLACKBOX}) &= \{a\} \\ \textit{clients}(\textit{BLACKBOX}) &= \{b, c\} \\ a \gg b \quad , & \quad \neg(a \gg c) \end{aligned}$$

We construct an *exploded* client-server digraph of a network of composite processes in the following way. The digraph contains a vertex for every client and server bundle of each process. If v and v' are vertices representing bundles b and b' of the same composite process P_i we draw an arc from v to v' if, and only if, $b \gg b'$ in P. If v and v' represent bundles of different processes P and P' then we draw an arc from v to v' if, and only if, both vertices represent the same channel bundle, v as a client bundle and v' as a server bundle.

We can derive the following result from these definitions.

Rule 9 *A client-server network, composed from composite client-server processes and with a circuit-free exploded client-server digraph, is deadlock-free.*

Proof. Starting with a network such as described in the statement of the theorem, consider the client-server digraph D' of the network which is derived when each composite process is separated back into its basic components. Suppose that this contains a circuit. This must be of the form

$$\langle a_1, b_{1,1}, b_{1,2}, ..b_{1,n_1}, a_2, b_{2,1}, b_{2,2}, ..b_{2,n_2}, ..a_m, b_{m,1}, b_{m,2}, ..b_{m,n_m}\rangle$$

where each subsequence $\langle b_{k,1}, ..b_{k,n_k}\rangle$ corresponds to a path through the client-server digraph of one of the original composite client-server processes, say P_k, and each arc a_k represents a channel bundle shared by two such composite processes P_{k-1} and P_k (where arithmetic is modulo m).

In the exploded client-server digraph of the original network, let each external channel bundle a of composite process P be represented by a vertex $a.P$. Then each bundle a_k is represented by two *vertices*, say $a_k.P_{k-1}$ and $a_k.P_k$, because bundle a_k is shared by processes P_{k-1} and P_k. These two vertices will be joined by an arc. Now for each pair of bundles a_k, a_{k+1} it is clear that $a_k \gg a_{k+1}$. Hence each pair of vertices $a_k.P_k$, $a_{k+1}.P_k$ will also be joined by an arc. So the exploded client-server digraph must contain a circuit

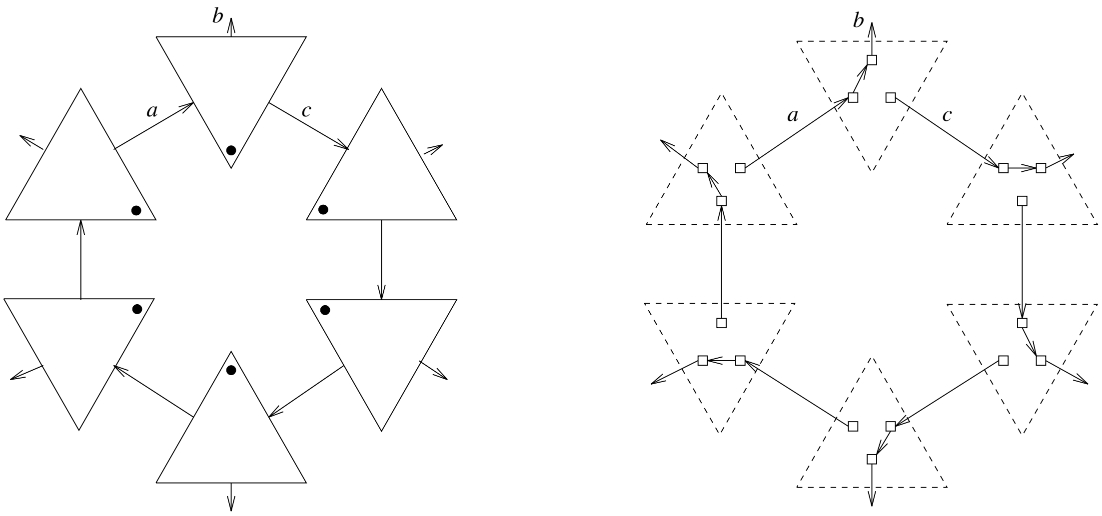
$$\langle(a_1.P_1, a_2.P_1), (a_2.P_1, a_2.P_2), .., (a_n.P_n, a_1.P_n), (a_1.P_n, a_1.P_1)\rangle$$

This is a contradiction so there is no circuit in D' and the result follows by rule 7□

Figure 2.7 displays two representations of a network constructed from six copies of *BLACKBOX* (with suitably relabelled channels): the client-server digraph, and an exploded client-server digraph. The former contains a circuit, so we cannot use rule 8 to show that the network is deadlock-free. However the latter contains none. So the network *is* deadlock-free by rule 9.

Note that when "exploding" a composite process, it is not always necessary to allocate a new vertex to every client or server bundle. Sometimes we can use a single node to represent several client or server bundles, without losing any information. This depends on the structure of the relation $\gg$.

Figure 2.7: Client-Server Digraph and Exploded Client-Server Digraph



The benefit of rules 8 and 9 is that we avoid repeating superfluous information in the diagrams we draw to design our programs. Instances of complex subnetworks are reduced to single nodes (or simplified representations when rule 8 is too weak).

Adding a Client-Server Interface to an Arbitrary Network

Rules 8 and 9 make available a hierarchical approach to software construction, based on multiple layers of the client-server model. It would also be nice to be able to use other paradigms to design subnetworks, and then wrap them up with a client-server interface for inclusion in a wider context.

Here we consider how to modify an arbitrary network, so that it appears as a single basic client-server process to its environment.

We start with a deadlock-free network $V = \langle P_1, .., P_n \rangle$, where each process P_i is itself divergence-free, deadlock-free and non-terminating. We want to add external communications to the components of this network to make it behave like a single basic client-server process. The resulting network will be called

$$V' = \langle P_1{}', .., P_n{}' \rangle$$

where each process $P_i{}'$ performs events in the alphabet of P_i and possibly additional events, which are external to the network, *i.e.*

$$i \neq j \Longrightarrow (\alpha P_i{}' - \alpha P_i) \cap \alpha P_j{}' = \{\}$$

The basic rule of thumb is that we may freely add client connections to any component process P_i, but we may add server connections to at most one such process.

Adherence to the following rules will guarantee that V' will behave as a single basic client-server process.

1. The additional channels of each process $P_i{}'$ are partitioned into client and server bundles, and $P_i{}'$ must obey the basic client-server protocol on these bundles.

   (The client-server bundles of V' are taken to be the union of those of each component, which will be disjoint. It is clear that V' will adhere to rules **(c)** and **(d)** of the basic client-server protocol if each process $P_i{}'$ does)

2. No more than one process $P_i{}'$ may have *server* connections.

   (This is to ensure that V' obeys rule **(b)** of the basic protocol. This restriction may be avoided if polling is used)

3. The new connections added to each process P_i must not interfere with its internal behaviour, *i.e.*

$$P_i{}' \setminus (\alpha P_i{}' - \alpha P_i) = P_i$$

   (By lemma 5 this condition guarantees that V' is deadlock-free, divergence-free and non-terminating – rule **(a)** of the basic client-server protocol.)

Example – Adding a Flexible Control Mechanism

In section 2.1 we designed a deadlock-free toroidal cellular array monitored by a control process, to be constructed using the *multi-phase-po* protocol. That approach required monitoring to be performed at fixed, predetermined intervals. A more flexible design is to add client connections to each cell, served by the control process. The new version looks like this.

$$\begin{array}{rcl}
\mathit{CELL}'(i,j) & = & \mathit{LEFT}'(i,j) \lhd (i+j) \quad \mathit{modulo}\ 2 = 0 \rhd \mathit{RIGHT}'(i,j) \\
\mathit{CHAT}(i,j) & = & \mathit{SKIP} \sqcap out.i.j \rightarrow in.i.j \rightarrow \mathit{SKIP} \\
\mathit{LEFT}'(i,j) & = & (e.i.j.\mathit{left} \rightarrow \mathit{SKIP} \mid\mid\mid e.(i-1).j.\mathit{right} \rightarrow \mathit{SKIP})\ ; \\
 & & \mathit{CHAT}(i,j)\ ;\ \mathit{UP}'(i,j) \\
\mathit{UP}'(i,j) & = & (e.i.j.\mathit{up} \rightarrow \mathit{SKIP} \mid\mid\mid e.i.(j-1).\mathit{down} \rightarrow \mathit{SKIP})\ ; \\
 & & \mathit{CHAT}(i,j)\ ;\ \mathit{RIGHT}'(i,j) \\
\mathit{RIGHT}'(i,j) & = & (e.i.j.\mathit{right} \rightarrow \mathit{SKIP} \mid\mid\mid e.(i+1).j.\mathit{left} \rightarrow \mathit{SKIP})\ ; \\
 & & \mathit{CHAT}(i,j)\ ;\ \mathit{DOWN}'(i,j) \\
\mathit{DOWN}'(i,j) & = & (e.i.j.\mathit{down} \rightarrow \mathit{SKIP} \mid\mid\mid e.i.(j+1).\mathit{up} \rightarrow \mathit{SKIP})\ ; \\
 & & \mathit{CHAT}(i,j)\ ;\ \mathit{LEFT}'(i,j)
\end{array}$$

$$\alpha \mathit{CELL}'(i,j) = \left\{ \begin{array}{cccc}
e.i.j.\mathit{left}, & e.i-1.j.\mathit{right}, & e.i.j.\mathit{up}, & e.i.j-1.\mathit{down} \\
e.i.j.\mathit{right}, & e.i+1.j.\mathit{left}, & e.i.j.\mathit{down}, & e.i.j+1.\mathit{up} \\
 & in.i.j, & out.i.j &
\end{array} \right\}$$

After each interaction with a neighbour the cell may non-deterministically decide to talk to a *CONTROL* process, implemented as follows.

$$\mathit{CONTROL} = \Box_{i=0}^{3} \Box_{j=0}^{3}\ \mathit{out}.i.j \rightarrow \mathit{in}.i.j \rightarrow \mathit{CONTROL}$$

We have added a client bundle of the form

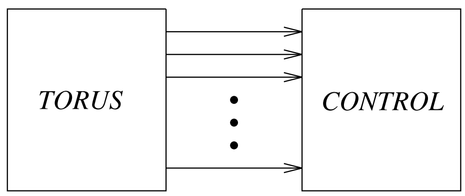
$$\langle \mathit{out}.i, j, \mathit{in}.i.j \rangle$$

to each cell. No server bundles have been added, and the additional channels do not affect the internal working of each process, *i.e.*

$$\mathit{CELL}'(i, j) \setminus \{\mathit{in}.i.j, \mathit{out}.i.j\} = \mathit{CELL}(i, j)$$

This may be proved using the algebraic laws of CSP. It follows that the complete cellular array now appears as a single basic client-server process to its environment. The client-server digraph which results is shown in figure 2.8. This contains no circuits so the entire system is deadlock free. It is now a simple matter to build extra client-server components onto the system, such as a user interface and a graphics handler.

Figure 2.8: Adding Client-Server Connections



2.3 Resource Allocation Protocol

The Resource Allocation Protocol was discussed briefly in the introduction. It will now be formalised, based on the treatment given in [Roscoe and Dathi 1986]. Then an extended version will be presented which allows resources to be built on to existing deadlock-free networks.

A *user-resource* network consists of a set of user processes $\{U_1, .., U_N\}$ which compete for a linearly ordered set of resource processes $(\{R_1, .., R_M\}, >)$ which have the following communication pattern.

$$R_j \quad = \quad \Box_{i:\{1,..,N\}}\, (\mathit{claim}_{ij} \rightarrow \mathit{release}_{ij} \rightarrow R_j)$$

Each resource j is initially ready to be claimed by any user process i using channel *claim*$_{ij}$. Then once it has been claimed it waits to be released, on channel *release*$_{ij}$, before returning to its initial state. Note that in this abstract model any details of message passing corresponding to the *claim* and *release* events are omitted.

Clearly the channels *claim*$_{ij}$ and *release*$_{ij}$ are only meant to be used by user process U_i, *i.e.*

$$k \neq i \implies \{\mathit{claim}_{ij}, \mathit{release}_{ij}\} \cap \alpha U_k = \{\}$$

We assume that each user process U_i is deadlock-free and non-terminating. It never tries to *claim* a resource that it already holds, nor to *release* one that it does not, *i.e.*

$$\forall s : \mathit{traces}(U_i). \quad 1 \geq (s \downarrow \mathit{claim}_{ij} - s \downarrow \mathit{release}_{ij}) \geq 0$$

Rule 10 (Resource Allocation Protocol) *Consider a user-resource network V constructed from users $\{U_1, .., U_N\}$ and resources $(\{R_1, .., R_M\}, >)$. Suppose that no user process ever attempts to acquire a higher resource than any that it already holds, i.e.*

$$\forall s : \mathit{traces}(U_i).$$
$$(s \downarrow \mathit{claim}_{ij} > s \downarrow \mathit{release}_{ij}) \wedge (R_k > R_j) \implies s^\frown\langle \mathit{claim}_{ik}\rangle \notin \mathit{traces}(U_i)$$

and also that it never communicates with any other user process

$$i \neq j \implies \alpha U_i \cap \alpha U_j = \{\}$$

Then the network is deadlock-free.

Proof. Suppose the condition of the protocol is adhered to, yet there is a deadlock state σ. So there exists a cycle of ungranted requests by theorem 1 (page 29), which must be of the following form due to the bipartite nature of the network.

$$U_{i_1} \xrightarrow{\sigma} \bullet R_{j_1} \xrightarrow{\sigma} \bullet U_{i_2} \xrightarrow{\sigma} \bullet R_{j_2} \, .. \, U_{i_k} \xrightarrow{\sigma} \bullet R_{j_k} \xrightarrow{\sigma} \bullet U_{i_1}$$

Here user U_{i_1} wants to claim resource R_{j_1}, which is already held by user U_{i_2}, which wants to claim resource R_{j_2}, etc. This implies the following contradiction

$$R_{j_1} > R_{j_2} > .. > R_{j_k} > R_{j_1} \quad \sharp$$

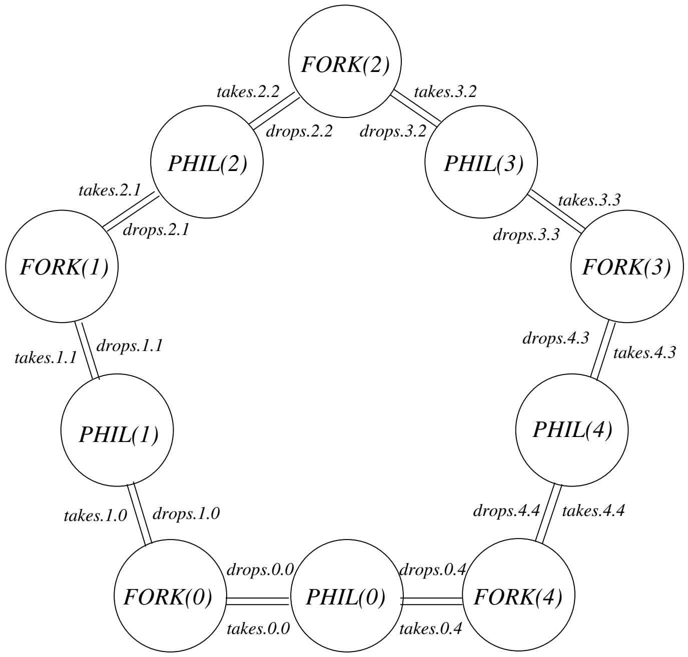
We conclude that the network can never deadlock$\square$

The Dining Philosophers network can be modelled in CSP as follows:

$$
\begin{aligned}
\mathit{PHIL}(i) &= \mathit{takes}.i.i \rightarrow \mathit{takes}.i.(i-1) \rightarrow \mathit{eats}.i \rightarrow \\
&\quad\ \mathit{drops}.i.(i-1) \rightarrow \mathit{drops}.i.i \rightarrow \mathit{PHIL}(i) \\[1em]
\alpha\mathit{PHIL}(i) &= \{\mathit{takes}.i.i, \mathit{takes}.i.(i-1), \mathit{eats}.i, \mathit{drops}.i.(i-1), \mathit{drops}.i.i\} \\[1em]
\mathit{FORK}(j) &= \Box_{i=0}^{4}\ \mathit{takes}.i.j \rightarrow \mathit{drops}.i.j \rightarrow \mathit{FORK}(j) \\[1em]
\alpha\mathit{FORK}(j) &= \{\mathit{takes}.0.j, \mathit{drops}.0.j, .., \mathit{takes}.4.j, \mathit{drops}.4.j\} \\[1em]
V &= \left\langle \begin{array}{ccccc} \mathit{PHIL}(0), & \mathit{PHIL}(1), & \mathit{PHIL}(2), & \mathit{PHIL}(3), & \mathit{PHIL}(4) \\ \mathit{FORK}(0), & \mathit{FORK}(1), & \mathit{FORK}(2), & \mathit{FORK}(3), & \mathit{FORK}(4) \end{array} \right\rangle
\end{aligned}
$$

where integer arithmetic is *modulo* 5.

Figure 2.9: Connection Graph for Dining Philosophers



The connection graph of this network is displayed in figure 2.9. If we take the forks to be the resource processes, ordered by

$$\mathit{FORK}(4) > \mathit{FORK}(3) > \mathit{FORK}(2) > \mathit{FORK}(1) > \mathit{FORK}(0)$$

and the philosophers to be the user processes, we see that the Resource Allocation Protocol is adhered to by all processes except $PHIL(0)$. As explained in the introduction, deadlock is possible for this system, for instance after trace

$$\langle takes.0.4, takes.1.0, takes.2.1, takes.4.3, takes.3.2\rangle$$

This is rectified by redefining $PHIL(0)$ to pick up his right-hand fork first.

$$\begin{aligned} PHIL(0) \;\; = \;\; & takes.0.4 \rightarrow takes.0.0 \rightarrow eats.0 \rightarrow \\ & drops.0.4 \rightarrow drops.0.0 \rightarrow PHIL(0) \end{aligned}$$

The resulting network is deadlock-free.

An Extended Protocol

The user processes will now be allowed to communicate with each other, so long as they do not attempt to do so while they are still holding any resources. The following result, inspired by an example from [Roscoe and Dathi 1986], will make it possible to build resources onto an existing deadlock-free network, without introducing any risk of deadlock.

Rule 11 (Extended Resource Allocation Protocol) *Take a user-resource network V constructed from users $\{U_1, .., U_N\}$ and resources $(\{R_1, .., R_M\}, >)$. Suppose that no user process ever attempts to acquire a higher resource than any it already holds, and never attempts to communicate with another user process while holding a resource, i.e.*

$$\begin{array}{rcl} & \forall\, s : traces(U_i). & \\ (s \downarrow claim_{ij} > s \downarrow release_{ij}) \wedge (R_k > R_j) & \Longrightarrow & s^\frown\langle claim_{ik}\rangle \notin traces(U_i) \\ (\exists\, j.\quad s \downarrow claim_{ij} > s \downarrow release_{ij}) \wedge l \neq i & \Longrightarrow & \begin{array}{c} \forall\, e : \alpha U_i \cap \alpha U_l. \\ s^\frown\langle e\rangle \notin traces(U_i) \end{array} \end{array}$$

If the subnetwork of user processes $\langle U_1, .., U_N\rangle$ is deadlock-free then the combined network of user processes and resource processes $\langle U_1, .., U_N, R_1, .., R_M\rangle$ is also deadlock-free.

Proof. Suppose that the conditions of the protocol are adhered to and also that the subnetwork of user processes $\langle U_1, .., U_N\rangle$ is deadlock-free, yet there is a deadlock state σ of the network. In this state every process is blocked. First we consider the possibility that in state σ no resource has been claimed, and therefore every resource is available to be claimed by any user process. It follows that each user process is only waiting to communicate with other user processes, *i.e* it is unable to perform any event outside the vocabulary of the subnetwork of user processes. So the subnetwork $\langle U_1, .., U_N\rangle$ itself has a state, derived from σ, in which every process is blocked. This is a deadlock state which contradicts our hypothesis.

So it must be the case that in state σ at least one resource R_i has been claimed. It is therefore waiting to be released by some user process U_j. Because U_j is currently holding resource R_i, it is not allowed to attempt communication with another user process so it must be waiting to claim another resource. In this way we can proceed to construct a cycle of ungranted requests, as was done in the proof of the basic Resource Allocation Protocol, leading to the same contradiction. We conclude that the network is deadlock-free□.

Example – The Arm-Wrestling Philosophers

To illustrate this we present a slight variation of the Dining Philosophers story, with arm-wrestling contests introduced to relieve the tedium of endless spaghetti eating and thinking. The philosophers are ranked according to seniority, given by

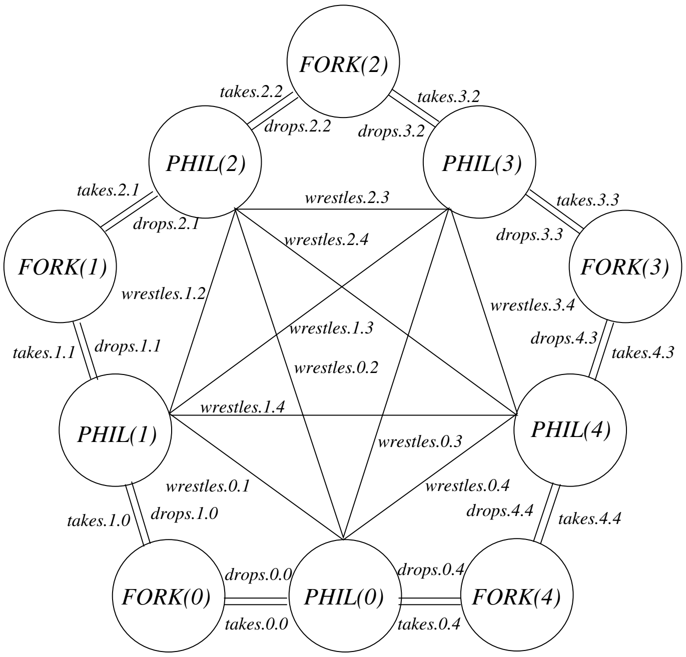
$$\mathit{PHIL}(4) > \mathit{PHIL}(3) > \mathit{PHIL}(2) > \mathit{PHIL}(1) > \mathit{PHIL}(0)$$

A philosopher may decide to eat some spaghetti or to challenge a senior philosopher to an arm-wrestling bout. Between meals he is also prepared to accept a challenge from any of his juniors. The new CSP definitions for the philosophers are given as follows.

$$\begin{array}{rcl}
\mathit{PHIL}(0) & = & \left(\begin{array}{l} \mathit{takes}.0.4 \rightarrow \mathit{takes}.0.0 \rightarrow \mathit{eats}.0 \rightarrow \\ \mathit{drops}.0.4 \rightarrow \mathit{drops}.0.0 \rightarrow \mathit{PHIL}(0) \end{array}\right) \sqcap \\
 & & \left(\sqcap_{i=1}^{4}\ \mathit{wrestles}.0.i \rightarrow \mathit{PHIL}(0)\right) \\
\mathit{PHIL}(i) & = & \left(\begin{array}{l} \left(\begin{array}{l} \mathit{takes}.i.i \rightarrow \mathit{takes}.i.(i-1) \rightarrow \mathit{eats}.i \rightarrow \\ \mathit{drops}.i.(i-1) \rightarrow \mathit{drops}.i.i \rightarrow \mathit{PHIL}(i) \end{array}\right) \sqcap \\ \left(\sqcap_{k=i+1}^{4}\ \mathit{wrestles}.i.k \rightarrow \mathit{PHIL}(i)\right) \end{array}\right) \Box \\
 & & \left(\Box_{k=0}^{i-1}\ \mathit{wrestles}.k.i \rightarrow \mathit{PHIL}(i)\right) \qquad\qquad i = 1, 2, 3 \\
\mathit{PHIL}(4) & = & \left(\begin{array}{l} \mathit{takes}.4.4 \rightarrow \mathit{takes}.4.3 \rightarrow \mathit{eats}.4 \rightarrow \\ \mathit{drops}.4.3 \rightarrow \mathit{drops}.4.4 \rightarrow \mathit{PHIL}(4) \end{array}\right) \Box \\
 & & \left(\Box_{k=0}^{3}\ \mathit{wrestles}.k.4 \rightarrow \mathit{PHIL}(4)\right) \\
\alpha\mathit{PHIL}(i) & = & \{\mathit{takes}.i.i, \mathit{takes}.i.(i-1), \mathit{eats}.i, \mathit{drops}.i.(i-1), \mathit{drops}.i.i\} \\
 & \cup & \{\mathit{wrestles}.i.k \mid k > i\} \quad \cup \quad \{\mathit{wrestles}.k.i \mid k < i\}
\end{array}$$

The subnetwork of philosophers is a simple example of a client-server network, where each philosopher interacts with his juniors as a server and his seniors as a client. It is easily shown to conform to the basic client-server protocol. Also the Extended Resource Allocation Protocol is observed when it comes to the use of forks. Hence the complete network of philosophers and forks is deadlock-free.

Figure 2.10: Arm-Wrestling Philosophers



Example – A Parallel Database

The Extended Resource Allocation Protocol is generally applicable to parallel algorithms for manipulating and processing large datasets. For example, figure 2.11 illustrates a simple design for a bank database. Each account is modelled as a resource process $ACCOUNT_j$. The user processes are configured as a *farm* network (consisting of a master and some slaves) to perform operations in parallel. Carrying out a transaction between two accounts requires that they be simultaneously held by a particular user process. There is clearly potential for deadlock here. Suppose that $SLAVE_p$ is told to move some money from $ACCOUNT_r$ to $ACCOUNT_s$, while at the same time $SLAVE_q$ is told to move some money from $ACCOUNT_s$ to $ACCOUNT_r$. If $SLAVE_p$ first opens $ACCOUNT_r$ and $SLAVE_q$ first opens $ACCOUNT_s$ they will become involved in a deadly embrace, which is likely to propagate throughout the system with disastrous consequences. The worst thing about this kind of deadlock is that it may take months or years of running time to appear, and so might not be revealed by testing. The possibility of deadlock in this situation could be removed through placement of an ordering on the accounts (which may be arbitrary) followed by adherence to the Extended Resource Allocation Protocol. The system might be generalised to a multi-user distributed database, with more complicated transactions. As long as all the database records required for a transaction are known in advance, the protocol is easily obeyed by claiming them in ascending order. A similar approach to this is described in [Wolfson 1987].

In practice, deadlock is found to be a significant problem in multi-user databases. P. Marcino reports on an insurance database application which regularly experiences over a hundred deadlocks in a single day [Marcino 1995]. He points out that the deadlock issue was ignored during the design phase, and only became apparent during initial testing. This is an all too common scenario. Much effort has been directed towards *deadlock-detection* algorithms [Knapp 1987]. Once a deadlock has been detected steps can then be taken to remove it by "rewinding" certain processes. It would seem to be much better programming practice to prevent deadlock from arising in the first place.

Figure 2.11: Bank Database System