Chapter 3

A Tool for Proving Deadlock-Freedom

Introduction

This chapter describes the development of Deadlock Checker, a tool which checks for adherence to the various design rules. It provides a vital safeguard against human error in their application.

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

An important feature of the design rules of the previous chapter is that they are easy to describe in an informal, intuitive manner as well as having precise, formal statements. The algorithms employed by Deadlock Checker, described below, scale efficiently to networks of arbitrary size. The combination of simple design rules and efficient machine verification would seem to be a powerful weapon against deadlock. It offers a solution to both the problems described above in the specific context of building deadlock-free concurrent systems.

Deadlock Checker operates by testing properties of individual CSP processes, or pairs of processes, within a network. This is done using *normal form* transition systems, which were devised by Roscoe for use in the refinement checking program FDR. The act of normalising a transition system is described below. A method of checking failures specifications for individual processes and pairs of processes, using normalised transition systems, is then developed. This technique enables the automatic verification of adherence to the design rules of the previous chapter.

Deadlock Checker also implements a more general deadlock analysis algorithm. A network's state dependence digraph is defined where each vertex corresponds to a state of an individual process, and each arc represents a potential ungranted request between processes. It is shown that if the state dependence digraph is circuit-free then the network is deadlock-free. This can be used to prove many useful networks deadlock-free, going beyond the bounds of the design rules. The programmer is allowed to be more adventurous and perhaps to bend the rules. The drawback with this approach is that diagnostic messages are less informative.

3.1 Normal Form Transition Systems

The design rules which Deadlock Checker understands are defined by specifications in the failures-divergences model of CSP. The processes to be analysed are non-terminating, which means that they have failures sets of infinite size. These are clearly unwieldy objects to use for machine verification. Fortunately Roscoe has developed a method for forming a unique finite representation of any process which has a finite number of operational states [Roscoe 1994]. This is basically a hybrid form of its operational and denotational representations which is called a *normal form transition system*. It is a digraph where each arc represents an event, and each vertex a *composite state*, labelled with either a set of minimal acceptance sets or a flag $\perp$ to symbolise divergence.

Rather than offering a precise description, we shall outline the process of normalisation with the aid of a worked example. Consider a process P defined by the mutually recursive CSP equations

$$\begin{aligned} P &= a \rightarrow b \rightarrow Q \sqcap c \rightarrow P \\ Q &= a \rightarrow b \rightarrow P \sqcap c \rightarrow P \end{aligned}$$

This process description is somewhat over-complicated for the behaviour it describes, as we shall soon see.

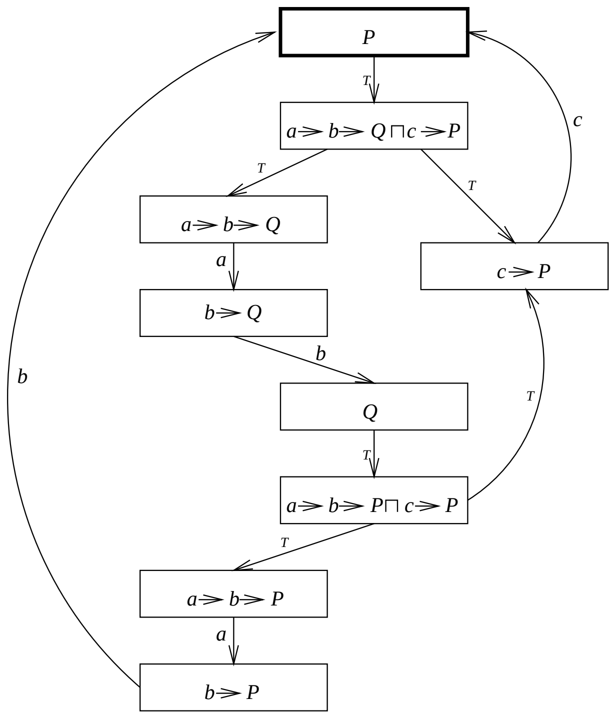
First the syntax is parsed into a tree of operators acting on processes or pairs of processes. This in turn is converted into a state transition system using the inference rules for operational semantics. (See section 1.3 for a description of this procedure.) Figure 3.1 illustrates the transition system for P. Recall that τ represents an internal decision – this is to cater for nondeterminism. States which have no τ transition event are described as *stable*, as no further internal activity is possible in those states.

Normalisation of a transition system is performed in three stages. Firstly a search is made for states from which an infinite series of hidden events is immediately possible, (*i.e.* states from which an indefinitely long walk of τ-labelled arcs can be constructed). Any such state is divergent and is labelled with $\perp$. In our example P is found to have no divergent state.

The second stage, called *pre-normalisation,* involves the elimination of τ arcs from the transition system, and also results in a unique event labelling of arcs originating from any node.

First the initial state is grouped together with any state that is reachable from there by performing a sequence of τ events. This group of states, which we shall call G_0,is

Figure 3.1: Transition System Resulting from Compilation

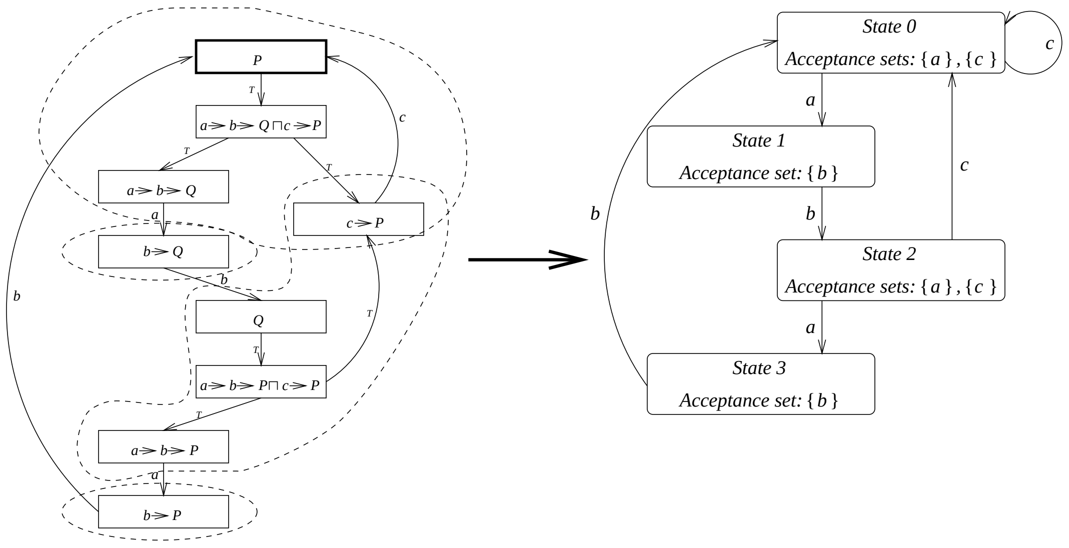


collectively mapped to the initial state of the pre-normal state-transition system. Figure 3.2 shows how the initial state in the transition system for P (itself labelled P) is grouped with states labelled $a \rightarrow b \rightarrow Q \sqcap c \rightarrow P$, $a \rightarrow b \rightarrow Q$ and $c \rightarrow P$.

If G_0 contains any divergent state then the new state is also labelled as divergent. Otherwise the new state is labelled with a list of minimal acceptance sets. (Minimal acceptance sets are the complement of maximal refusal sets. Acceptance sets are used here only because they typically smaller than refusal sets. The information carried is the same.) This is constructed by looking at all the stable states within G_0 and, for each one, the set of initial events that it offers. In figure 3.2 the state labelled $a \rightarrow b \rightarrow Q$ offers $\{a\}$ and the state labelled $c \rightarrow P$ offers $\{c\}$, so the initial state in the new transition system is labelled with minimal acceptance sets $\{\{a\},\{c\}\}$.

For each initial event x that is offered by states of G_0, apart from τ, a single transition is formed in the pre-normal transition system, leading to a new state constructed from the group of states reachable from states within G_0 by performing event x possibly followed by a sequence of τ events. The new state is labelled using the technique described above. Each time that a new group of states is formed a check is made to see whether it has already been discovered. The activity terminates once there are no more new state groupings to be found. Figure 3.2 illustrates the entire pre-normalisation procedure for process P.

Figure 3.2: Pre-normalisation

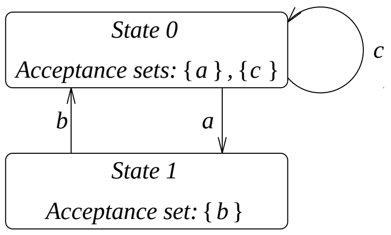


In the third stage any states which are indistinguishable in terms of subsequent behaviour are combined to form a unique compact normal form. Those states to be identified together are determined by first marking each state with *either* $\perp$ if it is divergent, *or* its initial actions and minimal acceptance sets, and then computing the fixed point of the following sequence of equivalence relations:

- $S_1 \sim_0 S_2$ if, and only if, they have the same marking.
- $S_1 \sim_{n+1} S_2 \iff (S_1 \sim_n S_2) \wedge \forall S_1', S_2'.(S_1 \xrightarrow{x} S_1' \wedge S_2 \xrightarrow{x} S_2' \implies S_1' \sim_n S_2')$

In our example $\sim_0$ partitions the states of the pre-normal form into $\{\{0, 2\}, \{1, 3\}\}$. This partition is preserved by $\sim_1$, and so it represents the fixed point of $\sim_n$. This gives us the unique representation of P of figure 3.3. Given that the initial state of this system is 0 it is simple to calculate the failures and divergences of P from this representation (by walking around the digraph).

Figure 3.3: Normal Form Transition System



Let us be more precise about the relationship between the failures and divergences of a general process P and its normalised transition system N (if one exists). For every minimal divergent trace s of P there will be a unique walk from the initial state σ_0 of N to a divergent state, with the transitions labelled according to s. Conversely the labels of any walk from σ_0 to a divergent state of N form a minimal divergence of P. For every maximal failure (s, X) of P, such that s is not a divergence of P, there will be a unique walk labelled as s, going from σ_0 to a non-divergent state σ, which has a minimal acceptance set $\Sigma - X$. Conversely, for every walk labelled s from σ_0 to a non-divergent state σ, P has maximal failures $(s, \Sigma - A_1) \mathinner{..} (s, \Sigma - A_k)$ where $A_1 \mathinner{..} A_k$ are the minimal acceptance sets of state σ.

FDR uses normal form transition systems to check for the refinement relation $\sqsubseteq$ between two processes S and P. By stepping through the states of the two processes simultaneously, it is checked whether every possible behaviour of P is permitted by S [Roscoe 1994]. In particular, FDR is often used to prove deadlock freedom by checking for refinement against the worst possible deadlock-free process of a given alphabet. Full details of how it is used are given in [Formal Systems 1993]. It is a very general tool but it runs into problems with large networks because of the exponential network state explosion as the number of processes increases.

3.2 Deadlock Checker

Deadlock Checker is implemented on top of FDR version 1.4, using the powerful functional programming language ML. (An excellent introduction is given to ML in [Paul-

Table 3.1: Machine Readable CSP

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

son 1991].) It takes a network of CSP programs as input, in the machine-readable syntax of Scattergood [Scattergood 1992]. FDR is used to compile the network into a set of individual normal form transition systems – one for each process. These are then used for performing the local checks required to guarantee adherence to the various design paradigms and prove deadlock-freedom. In this way networks with very large numbers of states may rapidly be proven deadlock-free.

The main difference between machine readable CSP and the algebraic form is that, in the former, the *type* of communication channels has to be explicitly defined using a `pragma` statement. The representation of various CSP operators in ASCII format is given in table 3.1.

Comment lines beginning with `--+` are used to specify to Deadlock Checker exactly which processes constitute the network to be analysed. There is no need to define the alphabets of these processes as the compiler calculates them automatically (as being exactly those events that each process may ever perform). However, there are circumstances where one might wish explicitly to define the process alphabets, and this feature could be included in a future version of the program. Dijkstra's classic Dining Philosophers network may be defined as follows.

```
--- CSP process definitions

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

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

FORK(i) = takes.i.i -> drops.i.i -> FORK(i) []
```

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

--- Define network for Deadlock Checker

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

This file, which is called `phils.csp` is processed by Deadlock Checker into a file `phils.net` containing a set of normalised transition systems - one for each process in the network, by starting up the program and typing the following command.

compile ”phils.csp” ”phils.net”;

Figure 3.4 illustrates the normal form transition systems for the Dining Philosophers network.

The interactive analysis may now proceed. First we must type a command to put Deadlock Checker into interactive mode.

teletype ();

```
Welcome to Deadlock Checker
Command (h for help, q to quit):
```

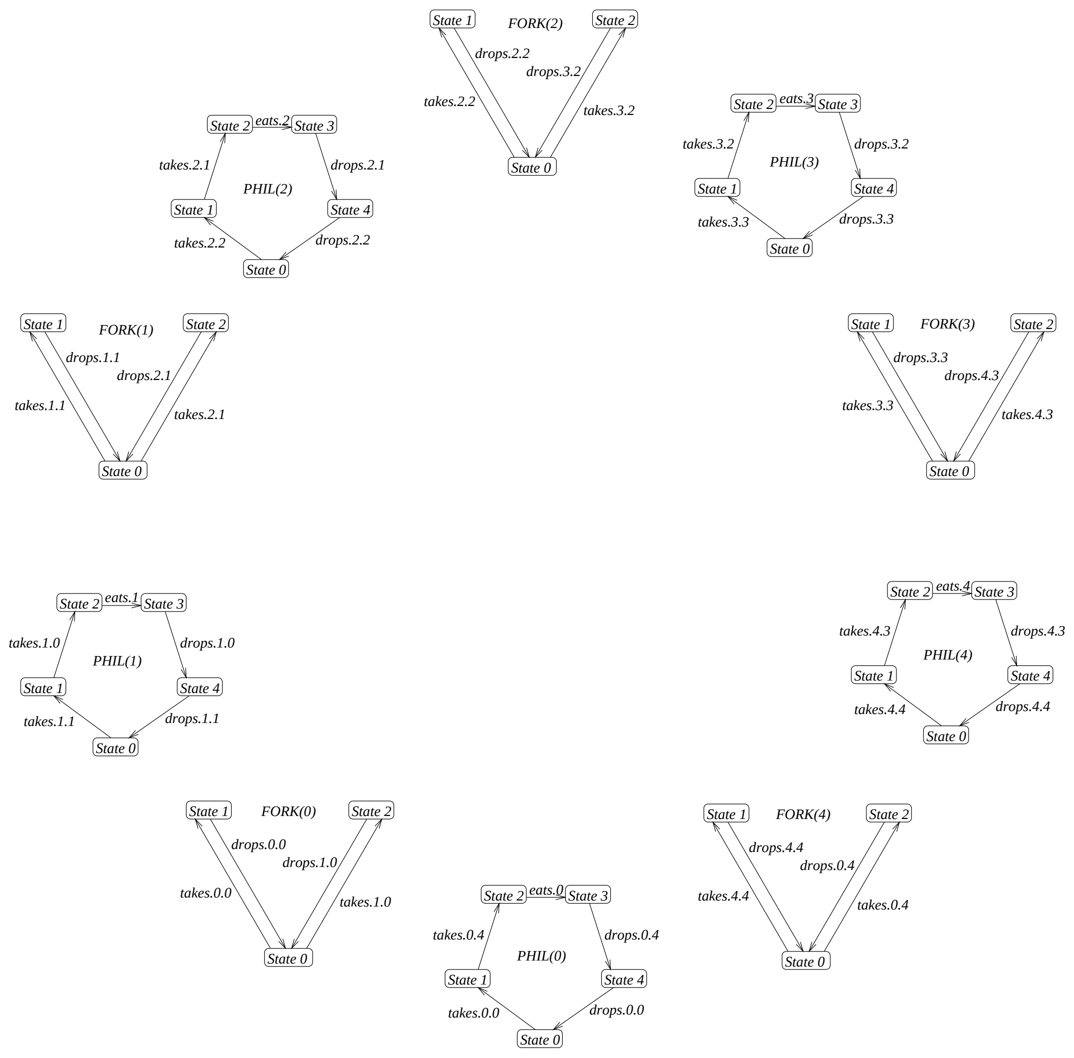
Typing *h* summons the following menu of commands.

```
h          - help: display this menu
l <file>   - load network file
n          - display list of networks in memory
s <name>   - select network
c          - display currently selected network
p          - display list of processes in current network
d          - decompose network analysis
v          - check for acyclic deadlock freedom
             (SDD algorithm)
x          - check for acyclic deadlock freedom
             (CSDD algorithm)
o          - check for deadlock in cyclic-po network
w          - check for deadlock in client-server network
a          - check for resource allocation protocol
r          - restrict network to its vocabulary
t          - test for livelock-freedom (Roscoe’s rule)
```

We load the compiled network definition as follows.

```
Command (h for help, q to quit):l phils.net
```

Figure 3.4: Normal Form Transition Systems for Dining Philosophers



Then we instruct Deadlock Checker to check for adherence to the Resource Allocation Protocol.

```
Command (h for help, q to quit):a
Network phils.net is busy
Network phils.net is triple-disjoint
Process FORK(4) acts as a resource
Process FORK(3) acts as a resource
Process FORK(2) acts as a resource
Process FORK(1) acts as a resource
Process FORK(0) acts as a resource
Process PHIL(4) is not a resource
User process PHIL(4) obeys resource allocation protocol
User process PHIL(3) obeys resource allocation protocol
User process PHIL(2) obeys resource allocation protocol
User process PHIL(1) obeys resource allocation protocol
User process PHIL(0) claims resource FORK(4) still holding FORK(0)
```

This network is not deadlock-free, and Deadlock Checker reveals the problem. The techniques used by Deadlock Checker to perform this analysis, and the other commands, will now be explained in detail. Further details are also to be found in [Martin 1995].

3.3 Checking Adherence to Design Rules

In this section we shall give details of the various algorithms employed by Deadlock Checker to test adherence to design rules. These algorithms will be illustrated with examples. We shall also estimate their time complexity as a function of n, where n is the number of processes in the network, unless otherwise stated.

Checking Network Prerequisites

Recall that our networks must be *triple-disjoint*, meaning that no event may be shared by more than two processes, and *busy*, meaning that each process must be deadlock-free, divergence-free and non-terminating. The property of triple-disjointedness can be established by the following algorithm

1. Assume that the events in the network $\langle P_1, .., P_n\rangle$ are numbered from 1 to m (we use the integer keys that FDR assigns to each event during compilation). Set up two arrays, *first* and *second*, with dimension m which are initially "undefined".

2. Scan the alphabet of each process P_i in turn. For each event $e \in \alpha P_i$, if *first*(e) is undefined then set
$$\textit{first}(e) := i$$

otherwise if *second*(e) is undefined then set

$$\textit{second}(e) := i$$

otherwise halt, because event e lies in the alphabet of three processes, and so the network is not triple-disjoint.

If we assume that the average number of events in the alphabet of each process remains fixed as the number of processes in the network, n, increases then the time complexity is $O(n)$

'Business' is also checked in $O(n)$ time, if we assume that the average number of states of each process remains roughly constant as n increases. We simply check every state of every process to make sure that it is not labelled as divergent and also does not have the empty set as a minimal acceptance set.

The prototype version of Deadlock Checker is programmed using only the standard core of ML. As this has no imperative arrays, the program does not achieve the theoretical efficiency of certain algorithms that it implements.

Checking Trace and Refusal Specifications

Any information about failures and divergences of a process may be extracted from its normalised transition system. Specifications on refusal sets are easy to check because all the required information may be deduced from the list of minimal acceptance sets stored at each vertex, and each vertex only needs to be looked at once. However a trace specification could potentially lead to an infinite search if not carefully stated.

Consider the specification

$$\forall s \in \textit{traces}(P). \quad (s \downarrow b + 4) \geq 2(s \downarrow a) \geq s \downarrow b$$

Starting at the initial state of P we might search through the transition digraph, keeping a record of the current trace, and checking every possible trace for $s \downarrow a$ and $s \downarrow b$. This search might never terminate for a component of a busy network.

There is a much better approach to this problem, as follows. We write our specification like this

$$\forall s \in \textit{traces}(P). 4 \geq 2(s \downarrow a) - s \downarrow b \geq 0$$

Then we define an *incremental* trace function f as follows

$$f(\langle\rangle) = 0$$

$$f(s \frown \langle x \rangle) = \begin{cases} f(s) + 2 & \text{if} \quad x = a \\ f(s) - 1 & \text{if} \quad x = b \\ f(s) & \text{otherwise} \end{cases}$$

It is clear that

$$f(s) = 2(s \downarrow a) - s \downarrow b$$

We start an exhaustive search through the transition system for pairs of the form (σ, v), where σ is a state and v is a possible value of $f(s)$ at that state. The search terminates either when there are no new such pairs to be found, or if we find a pair for which $\neg\ (4 \geq v \geq 0)$.

There are two reasons why this approach is better. Firstly we have defined our variant function, f, in an incremental way, which means that we do not need to store any information about traces. The value of $f(s)$ at each point in the search can be calculated purely from the information stored at the previous point. Secondly we have converted an endless search into one that is guaranteed to terminate, due to the bounds placed on the range of f.

This technique can be extended to a network of two processes $\langle P, Q\rangle$, and a specification on network states $(s, \langle X_P, X_Q\rangle)$. We assume that the specification is expressed as a predicate

$$\mathit{PRED}(f_1(s), .., f_n(s), X_P, X_Q)$$

involving a number of incremental trace functions f_i and maximal refusal sets X_P and X_Q of P and Q.

Two sets of records are maintained: *pending* and *done*. Each record is of the form $(\sigma_P, \sigma_Q, v_1, .., v_n)$, where (σ_P, σ_Q) is a pair of normal form states in which P and Q may simultaneously rest, and each v_i is the value of $f_i(s)$ for a corresponding trace s. The algorithm proceeds as follows.

1. Initially *pending* consists of a single record corresponding to the original state of the system, and *done* is empty.

$$\begin{array}{rcl} \mathit{pending} & := & \{(0, 0, f_1(\langle\rangle), .., f_n(\langle\rangle))\} \\ \mathit{done} & := & \{\} \end{array}$$

2. Take a new record from *pending* to be processed.

$$\begin{array}{rcl} r & := & (\sigma_P, \sigma_Q, v_1, .., v_n) \in \mathit{pending} \\ \mathit{pending} & := & \mathit{pending} - \{r\} \end{array}$$

3. Now check whether record r satisfies the specification. Suppose that σ_P has a set A of minimal acceptance sets and σ_Q has a set B of minimal acceptance sets.

   If $\exists\, a : A, b : B.\quad \neg\mathit{PRED}(v_1, .., v_n, \alpha P - a, \alpha Q - b)$ then halt. (The specification is *not* satisfied). Otherwise

$$\mathit{done} := \mathit{done} \cup \{r\}$$

4. Now construct the set *new* of successor records of r, by considering every transition that is possible for $\mathit{PAR}(\langle P, Q\rangle)$ from state pair (σ_P, σ_Q). Assume that r

corresponds to some trace s of *PAR*$(\langle P, Q\rangle)$. Then

$$\begin{array}{rcl} \textit{new} & := & \left\{ \begin{array}{c} (\sigma'_P, \sigma_Q, f_1(s^\frown\langle x\rangle), .., f_n(s^\frown\langle x\rangle))| \\ x \in \alpha P - \alpha Q \wedge \sigma_P \xrightarrow{x} \sigma'_P \end{array} \right\} \\ & \cup & \left\{ \begin{array}{c} (\sigma_P, \sigma'_Q, f_1(s^\frown\langle x\rangle), .., f_n(s^\frown\langle x\rangle))| \\ x \in \alpha Q - \alpha P \wedge \sigma_Q \xrightarrow{x} \sigma'_Q \end{array} \right\} \\ & \cup & \left\{ \begin{array}{c} (\sigma'_P, \sigma'_Q\, f_1(s^\frown\langle x\rangle), .., f_n(s^\frown\langle x\rangle))| \\ x \in \alpha P \cap \alpha Q \wedge \sigma_P \xrightarrow{x} \sigma'_P \wedge \sigma_Q \xrightarrow{x} \sigma'_Q \end{array} \right\} \end{array}$$

Although we have not stored any record of a value of s that corresponds to r, it is not actually required in order to perform this calculation due to the incremental method of defining the various trace functions.

5. Now we eliminate records from *new* that have already been processed and merge the remainder into *pending*.

$$\textit{pending} := \textit{pending} \cup (\textit{new} - \textit{done})$$

6. If *pending* $= \{\}$ then halt. (The specification is satisfied.) Otherwise return to step 2.

This algorithm is not certain to terminate for every given set of incremental trace functions f_i and predicate *PRED*. But if there is a finite range of values for each f_i outside which satisfaction of *PRED* is impossible then termination is guaranteed for any network $\langle P, Q\rangle$.

The following example is included in order to illustrate this technique. Consider the network $V = \langle \textit{LEFT}, \textit{RIGHT}\rangle$ with the following process definitions.

$$\begin{array}{rcl} \textit{LEFT} & = & \textit{in} \rightarrow \textit{mid} \rightarrow \textit{LEFT} \\ \alpha\textit{LEFT} & = & \{\textit{in,mid}\} \\ \\ \textit{RIGHT} & = & \textit{mid} \rightarrow \textit{out} \rightarrow \textit{RIGHT} \\ \alpha\textit{RIGHT} & = & \{\textit{mid,out}\} \end{array}$$

Suppose we wish to prove that the following trace specification is satisfied by *PAR*(V).

$$2 \geq s \downarrow \textit{in} - s \downarrow \textit{out} \geq 0$$

V is an abstract representation of a double buffer, which inputs information on channel *in* and outputs it on channel *out*. The specification simply states that the number of messages held in the buffer at any given time lies between nought and two inclusive.

We proceed by defining an incremental trace function f as follows

$$\begin{array}{rcl} f(\langle\rangle) & = & 0 \\ f(s^\frown\langle x\rangle) & = & \left\{ \begin{array}{ll} f(s) + 1 & \text{if} \quad x = \textit{in} \\ f(s) - 1 & \text{if} \quad x = \textit{out} \\ f(s) & \text{otherwise} \end{array} \right. \end{array}$$

It is clear that

$$f(s) = s \downarrow \mathit{in} - s \downarrow \mathit{out}$$

In this case our predicate function *PRED* is given by

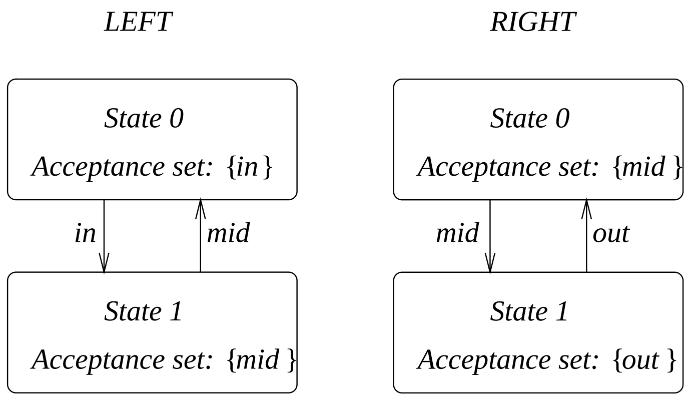
$$\mathit{PRED}(f(s)) = (2 \geq f(s) \geq 0)$$

Normal form state transition systems for the network V are shown in figure 3.5. We now proceed to form an exhaustive set of records of the form

$$(\sigma_{\mathit{LEFT}}, \sigma_{\mathit{RIGHT}}, \mathit{val})$$

consisting of a state of process *LEFT*, a corresponding state of process *RIGHT* and a possible value for $f(s)$ when the processes are in those states.

Figure 3.5: Normal Form Transition Systems for Two-Place Buffer



The search proceeds as follows. First we have

$$\mathit{pending} = \{(0,0,0)\}, \quad \mathit{done} = \{\}$$

Check $(0,0,0)$; possible transition is *in*; leads to record: $(1,0,1)$. Now we have

$$\mathit{pending} = \{(1,0,1)\}, \quad \mathit{done} = \{(0,0,0)\}$$

Check $(1,0,1)$; possible transition is *mid*; leads to record: $(0,1,1)$. Now we have

$$\mathit{pending} = \{(0,1,1)\}, \quad \mathit{done} = \{(0,0,0),(1,0,1)\}$$

Check $(0,1,1)$; possible transitions are *in, out*; lead to records: $(1,1,2)$, $(0,0,0)$. Now we have

$$\mathit{pending} = \{(1,1,2)\}, \quad \mathit{done} = \{(0,0,0),(1,0,1),(0,1,1)\}$$

Check $(1, 1, 2)$; possible transition is *out*; leads to record: $(1, 0, 1)$. Now we have

$$\mathit{pending} = \{\}, \quad \mathit{done} = \{(0,0,0), (1,0,1), (0,1,1), (1,1,2)\}$$

The search is now complete. Every record that was found satisfies the original specification, and we shall conclude that it is satisfied by $\mathit{PAR}(V)$. This is rather a bold claim given that the set of traces of $\mathit{PAR}(V)$ is infinite and we have only examined four cases. But it may be justified by using induction on traces, as follows.

Every trace s of $\mathit{PAR}(V)$ corresponds to a unique pair of normal-form states

$$(\sigma_{\mathit{LEFT}}, \sigma_{\mathit{RIGHT}})$$

These are found by constructing the unique walk in the normal-form transition system of *LEFT* with labels $s \upharpoonright \alpha \mathit{LEFT}$, and the unique walk in the normal-form transition system of *RIGHT* with labels $s \upharpoonright \alpha \mathit{RIGHT}$. We shall call this state pair

$$(\sigma_{\mathit{LEFT}}(s), \sigma_{\mathit{RIGHT}}(s))$$

Now suppose that for a certain trace t, we know that record

$$(\sigma_{\mathit{LEFT}}(t), \sigma_{\mathit{RIGHT}}(t), f(t))$$

lies in set *done*, constructed above. Now consider a trace $t^\frown\langle x\rangle$ of V. This corresponds to a state pair

$$(\sigma_{\mathit{LEFT}}(t^\frown\langle x\rangle), \sigma_{\mathit{RIGHT}}(t^\frown\langle x\rangle))$$

which must be reachable from $(\sigma_{\mathit{LEFT}}, \sigma_{\mathit{RIGHT}})$ by one or both of the processes performing event x. It follows that record

$$(\sigma_{\mathit{LEFT}}(t^\frown\langle x\rangle), \sigma_{\mathit{RIGHT}}(t^\frown\langle x\rangle), f(t^\frown\langle x\rangle))$$

must also lie in set *done*, due to the incremental way in which this set was constructed.

We actually know that

$$(\sigma_{\mathit{LEFT}}(\langle\rangle), \sigma_{\mathit{RIGHT}}(\langle\rangle), f(\langle\rangle)) = (0, 0, 0) \in \mathit{done}$$

because this is the record that was used to start the search. Hence, by induction, *every* trace s of V is represented in *done* by a record of the form

$$(\sigma_{\mathit{LEFT}}(s), \sigma_{\mathit{RIGHT}}(s), f(s))$$

So we conclude that the original specification is satisfied by all traces of $\mathit{PAR}(V)$.

Although this proof technique is tedious for humans it is very easy to automate on a computer. It would be feasible to extend the technique to networks of more than two processes, but due to the exponential state explosion as networks grow larger, this would have limited potential in practice.

Note that, for individual processes, it is often feasible to perform this kind of specification check using FDR directly. In order to prove that a process P satisfies some specification one constructs a process S that is the worst possible process that satisfies the specification and then shows that $P \sqsupseteq S$. However specifications of networks of two processes which involve the refusal sets of individual processes, such as the formal statement of conflict-freedom, cannot be checked directly using FDR.

Resource Allocation Protocol

Deadlock Checker includes a check for adherence to the Extended Resource Allocation Protocol. This depends on the processes which constitute the network being presented in a particular order. The network is assumed to consist of a sequence of user processes $\langle U_1, .. U_M \rangle$ followed by an *ordered* sequence of resource processes $\langle R_1 .. R_N \rangle$. Observe that in the example of the Dining Philosophers network (page 67) the processes are presented in the following order (which conforms to this requirement)

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

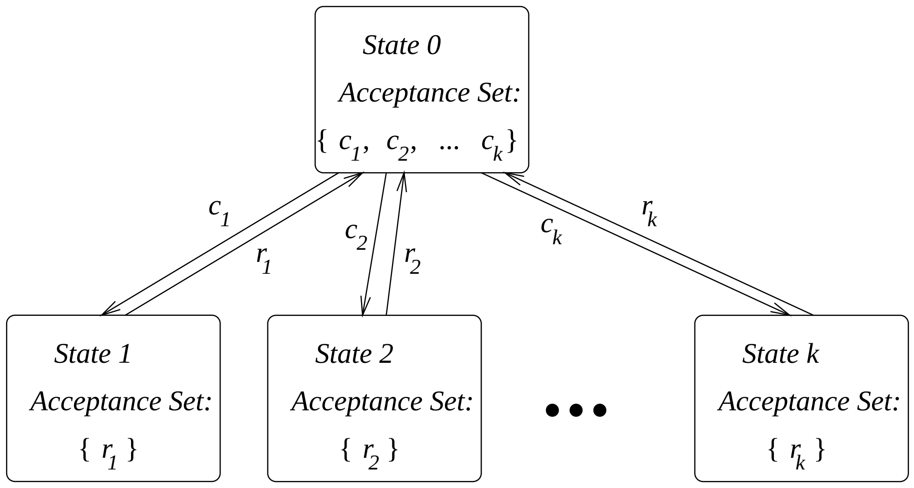
The analysis proceeds in two stages. The first stage is to start from the end of the list and work backwards to see how many processes behave as resources.

Checking that a process P behaves as a resource relies on the fact that the normal-form transition system of a resource process has a very specific form. Consider a general resource process

$$R = \Box_{i=1}^{k} \; c_i \rightarrow r_i \rightarrow R$$

The normal-form transition system for this process is shown in figure 3.6. It has an initial state representing the situation where the resource has not been 'claimed', plus one state for each claim channel c_i, representing the state of having been claimed on that channel.

Figure 3.6: Normal Form Transition System for General Resource Process



To establish whether a given process P is of this form involves firstly attempting to split its alphabet into a set of *claim-release* pairs $\{(c_1, r_1), ..(c_k, r_k)\}$. The initial state of P should have a single minimal acceptance set $\{c_1, .., c_k\}$ equal to the set of initial events of P. Then for each c_i there should be a transition to a state S_i with a single minimal acceptance set $\{r_i\}$ and a single transition back to the initial state of P. Each of the r_i must be distinct and different from all the c_i.

If this splitting of αP proves successful, it must then be checked that each of the claim-release pairs consists of events from the alphabet of a process before P in the network list. Also no two event pairs should match the same process. If this is so P is taken to be a valid resource process. At the same time a list of claim-release pairs, *cr_list*(U), is constructed for each user process U, consisting of records of the form $((c, r), n)$, where (c, r) is a claim-release pair and n is a resource number (taken as the numeric order of the resource in the network). (Note that we have relaxed the condition that each resource needs to make itself available to every user process. A resource may be private to a particular subset of users.)

Performing this check on the normal-form transition system for process *FORK*(4) (see figure 3.4) results in splitting up its alphabet into two pairs

$$\{(\mathit{takes}.4.4, \mathit{drops}.4.4), (\mathit{takes}.0.4, \mathit{drops}.0.4)\}$$

It is then found that

$$\begin{array}{rcl} \{\mathit{takes}.4.4, \mathit{drops}.4.4\} & \subseteq & \alpha \mathit{PHIL}(4) \\ \{\mathit{takes}.0.4, \mathit{drops}.0.4\} & \subseteq & \alpha \mathit{PHIL}(0) \end{array}$$

So it is concluded that *FORK*(4) is a resource.

As soon as a process is discovered which does not behave as a resource it is taken to be a user process, along with all the processes which precede it in the network ordering. In the case of the Dining Philosophers, the first non-resource process discovered is *PHIL*(4). Each user process must then be checked for adherence to the Extended Resource Allocation protocol. This protocol was defined formally using failures specifications on page 58. We need to check that each user process U communicates with its resources in alternating sequence on each (c, r) pair in *cr_list*(U). Also that it attempts only to claim resources ordered below those that it already holds, and never attempts to communicate with another user while holding a resource. This is achieved by casting the specification in terms of incremental trace functions and then using the technique described on page 71, as follows.

Let

$$\mathit{cr\_list}(U) = \langle((c_1, r_1), n_1), .., ((c_k, r_k), n_k)\rangle$$

Then, for each $i \in \{1, .., k\}$ define

$$f_i(s) = s \downarrow c_i - s \downarrow r_i$$

Incrementally, this is written

$$\begin{array}{rcl} f_i(\langle\rangle) & = & 0 \\ f_i(s^\frown\langle x\rangle) & = & \left\{ \begin{array}{lll} f_i(s) + 1 & \text{if} & x = c_i \\ f_i(s) - 1 & \text{if} & x = r_i \\ f_i(s) & \text{otherwise} & \end{array} \right. \end{array}$$

Function $f_i(s)$ should take the value 1 whenever user U is holding resource n_i and otherwise take the value 0. In this case, rather than examining the minimal acceptance sets of U after trace s, we need to look at its initial events I. These are available as the transition events that are possible from the normal form state of U that corresponds to s. We define

$$PRED(f_1(s), .., f_k(s), I) = \left(\begin{array}{c} \forall j : \{1, .., k\}.(1 \geq f_j(s) \geq 0) \wedge \\ \left(f_j(s) = 1 \Longrightarrow \left(\begin{array}{c} (\forall i.c_i \in I \Longrightarrow n_i < n_j) \wedge \\ (\forall U' \neq U.I \cap \alpha U' = \{\}) \end{array} \right) \right) \end{array} \right)$$

If this specification check succeeds for each user process then the deadlock analysis is reduced to the subnetwork $\langle U_1, .. U_M \rangle$ which must be analysed by other means. It may well be that the user processes have disjoint alphabets, in which case no further analysis is required.

It is important to note a minor flaw in the part of the algorithm which identifies resource processes. It is possible that a network could contain one or more processes which are intended to be treated as users, but which never actually use any resources and appear to behave like resources themselves. These could be identified as such in the searching process described above, which could then lead to a valid deadlock-free network being rejected. This is very unlikely to occur in practice. The problem could be avoided by modifying Deadlock Checker to insist that resource processes be explicitly labelled as such, but as this would cause unnecessary inconvenience in the vast majority of cases it has not been done.

Complexity

We shall continue to assume that as the number of processes in a network, n, increases, the number of states and events of each process remains approximately fixed. This means that the time taken to perform any local analysis of an individual process, or pairs of neighbouring processes, can be assumed to be independent of the size of the network.

Let us consider the algorithm for checking the Resource Allocation Protocol. We assume that the proportion of user processes to resource processes remains fixed as n grows. Starting at the end of the network list, the claim-release channels pairs for each resource process discovered need to be matched with the alphabet of a process which precedes it. Each matching operation can be done in constant time by making use of the two arrays *first* and *second*, indexed by events in αV, which were set up in order to verify triple-disjointedness (page 70). So the entire matching process is $O(n)$. All the other checking performed is local to a process, and so $O(n)$ for the network as a whole (by the above assumptions). This gives us an overall complexity of $O(n)$.

Cyclic Processes

To analyse a network purporting to be cyclic-PO, we need to check that each process communicates cyclically on its channels according to some partial order, for which we construct the Hasse digraph. This is the minimal representation of a partial order; it has a vertex for each element of the partial order and an arc xy whenever element y is *directly* below x, *i.e.*

$$x > y \quad \wedge \quad \nexists\, z . x > z > y$$

Then, in order to prove deadlock-freedom, we must show that the union of the Hasse digraphs, which we call the *channel dependence digraph*, contains no circuit.

Recall that we formally defined the cyclic-po process *CYCLIC-PO*$(X, >)$, which communicates on the set of channels X, partially ordered by $>$, as follows.

$$\begin{array}{rcl}
\mathit{CYCLIC\text{-}PO}(X, >) & = & C2(X, \{\}, >) \\
C2(X, \mathit{DONE}, >) & = & C2(X, \{\}, >) \\
 & & \lhd \mathit{DONE} = X \rhd \\
 & & \Box_{x : \mathit{mins}(X - \mathit{DONE}, >)}\; x \rightarrow C2(X, \mathit{DONE} \cup \{x\}, >)
\end{array}$$

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

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

It can be shown that this definition is unchanged when $(X, >)$ is replaced with its Hasse digraph.

For verifying that a process P is cyclic and extracting its Hasse channel ordering a two pass algorithm is employed as follows. The first pass tries to extract a Hasse digraph on the assumption that the process is indeed cyclic. In each state s of the normal form transition system of P we look at every transition (e, s') that does not take us back to the the initial state of P. If an event e' is possible in state s' that was not possible in state s we assume that $e' > e$. When this first stage is complete we will have constructed a relation $>$ on the channels of P. If P is cyclic-PO this will actually be the Hasse digraph of its channel ordering. This is because whenever a cyclic-PO process performs an event e and then immediately becomes ready to perform event e', without having completed a cycle, e' must be directly above e in the channel ordering. If P is not cyclic-po the relation that we have constructed will be meaningless.

If the $>$ relation contains a cycle $c_1 > .. > c_k > c_1$ we can eliminate P straight away. Otherwise we must now check whether the behaviour of P adheres exactly to *CYCLIC-PO*$(\alpha P, >)$. This relies on the normal-form transition system of the latter having a very specific form. Each state corresponds to the process $C2(\alpha P, \mathit{DONE}, >)$ for a particular set of events *DONE*. We perform the check using a depth-first search (see appendix B) starting from the initial state of P. For each state of P that we visit we maintain a record of the events that have been performed to arrive there, and call this set *DONE*. We then check that the immediate behaviour at each state, as given by its acceptance sets and transition events, conforms to that of $C2(X, DONE, >)$. We also check that *DONE* is consistent when a state is visited more than once. Whenever the initial state is revisited, *DONE* should be equal to αP. This second pass can only succeed if P is cyclic-PO with ordering $>$.

If every process in the network is found to be cyclic-PO, the Hasse digraphs of their channel orderings are aggregated into a global channel dependence digraph. We know from theorem 7, page 40, that the network is deadlock-free if, and only if, there is no circuit in the $\rhd$ relation

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

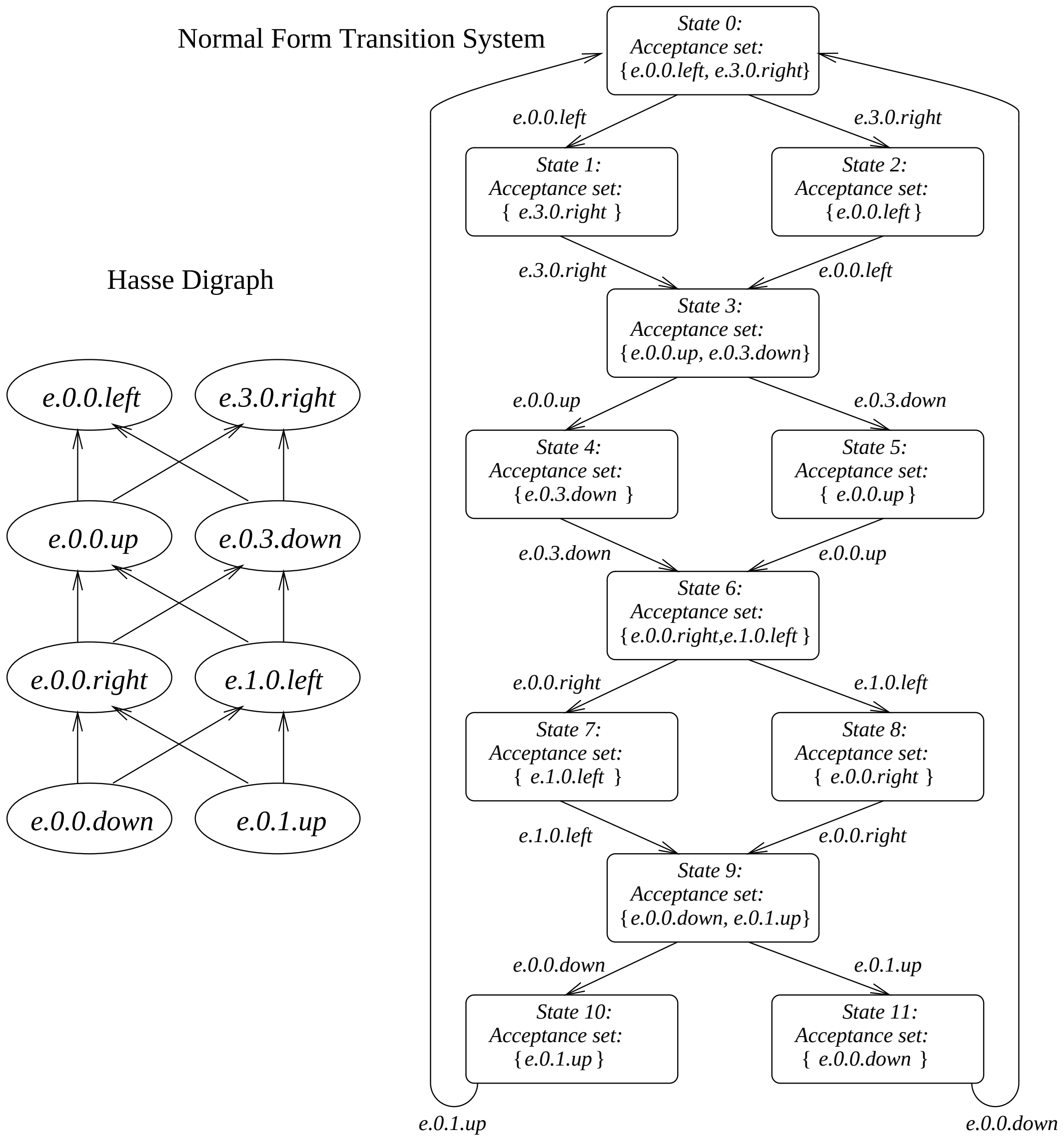
Now $\rhd$ is the union of the full channel orderings of each process in the network and so the channel dependency digraph is a subset of $\rhd$. However it is a subset which carries all the vital information and it may easily be shown that the channel dependency digraph contains a circuit if, and only if, $\rhd$ contains a cycle. It follows that the network is deadlock-free if, and only if, there is no circuit in the channel dependency digraph. This is checked using the DFS algorithm.

To demonstrate the use of this tool we recall the toroidal cellular array. This is coded in machine-readable CSP as follows.

```
n=4
indices = {0,1,2,3}
pragma channel e:indices.indices.{left,up,right,down}

CELL(i,j) = if ((i+j)%2==0) then LEFT(i,j) else RIGHT(i,j)

LEFT(i,j) = e.i.j.left -> e.((i-1)%n).j.right -> UP(i,j) []
            e.((i-1)%n).j.right -> e.i.j.left -> UP(i,j)

UP(i,j) = e.i.j.up -> e.i.((j-1)%n).down -> RIGHT(i,j) []
          e.i.((j-1)%n).down -> e.i.j.up -> RIGHT(i,j)

RIGHT(i,j) = e.i.j.right -> e.((i+1)%n).j.left -> DOWN(i,j) []
             e.((i+1)%n).j.left -> e.i.j.right -> DOWN(i,j)

DOWN(i,j) = e.i.j.down -> e.i.((j+1)%n).up -> LEFT(i,j) []
            e.i.((j+1)%n).up -> e.i.j.down -> LEFT(i,j)

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

Each process is cyclic and communicates with each of its neighbours in turn. (Note that the interleaving construct has been algebraically transformed into external choice. This is due to a syntax restriction placed on CSP by FDR 1.4.) Deadlock should be avoided because alternate cells commence with different orientations. The Hasse digraph and normal-form state transition system for process $CELL(0, 0)$ are illustrated in figure 3.7.

Figure 3.7: Hasse Digraph and Normal Form Transition System for *CELL*$(0, 0)$


Normal Form Transition System
State 0: Acceptance set: {e.0.0.left, e.3.0.right}
e.0.0.left
e.3.0.right
State 1: Acceptance set: { e.3.0.right }
State 2: Acceptance set: {e.0.0.left }
e.3.0.right
e.0.0.left
State 3: Acceptance set: {e.0.0.up, e.0.3.down}
e.0.0.up
e.0.3.down
State 4: Acceptance set: {e.0.3.down }
State 5: Acceptance set: { e.0.0.up }
e.0.3.down
e.0.0.up
State 6: Acceptance set: {e.0.0.right,e.1.0.left }
e.0.0.right
e.1.0.left
State 7: Acceptance set: { e.1.0.left }
State 8: Acceptance set: { e.0.0.right }
e.1.0.left
e.0.0.right
State 9: Acceptance set: {e.0.0.down, e.0.1.up}
e.0.0.down
e.0.1.up
State 10: Acceptance set: {e.0.1.up }
State 11: Acceptance set: { e.0.0.down }
e.0.1.up
e.0.0.down
Hasse Digraph
e.0.0.left
e.3.0.right
e.0.0.up
e.0.3.down
e.0.0.right
e.1.0.left
e.0.0.down
e.0.1.up

We load the compiled network definitions, and check for adherence to the cyclic-PO protocol.

```
Command (h for help, q to quit):l torus.net
Command (h for help, q to quit):o
```

For each process in the network a report like this one is returned

```
Process CELL(0,0) is cyclic-po:
  (e.0.0.up    > e.3.0.right),  (e.0.3.down  > e.3.0.right),
  (e.0.0.up    > e.0.0.left ),  (e.0.3.down  > e.0.0.left ),
  (e.0.0.right > e.0.3.down ),  (e.1.0.left  > e.0.3.down ),
  (e.0.0.right > e.0.0.up   ),  (e.1.0.left  > e.0.0.up   ),
  (e.0.0.down  > e.1.0.left ),  (e.0.1.up    > e.1.0.left ),
  (e.0.0.down  > e.0.0.right),  (e.0.1.up    > e.0.0.right)
```

The program then checks for circuits in the channel dependency digraph, and finding none reports

```
Network torus.net is deadlock-free
```

If we change the dimensions of the toroidal array to 5×5, it turns out that the network will deadlock, as is revealed by Deadlock Checker in the following way.

```
Found closed trail of dependent channels:
<e.4.4.right,e.4.4.up,e.4.3.right,e.0.4.up,e.4.4.right>
Network torus5.net deadlocks
```

When deadlock has been identified the reason behind it is always reported.

The algorithm for checking cyclic-PO networks involves local checking of each process to establish its channel ordering, which is $O(n)$, plus a check for circuits in the channel dependence digraph. We can assume that the number of edges in this graph grows proportionally to n by taking the number of edges in the Hasse digraph of each process to be independent of n. Checking for circuits can be performed in linear time with the DFS algorithm. So the cyclic-PO network check can be done with $O(n)$ complexity.

Client-Server Protocol

Deadlock Checker contains a tool for verifying that a network has been implemented according to the basic client-server protocol (described on page 45). There are two phases to the method employed. Firstly the program attempts to identify the client and server channels bundles of each process in the network. For this to be feasible, the order in which the processes are supplied in the network is significant. A process should communicate with those before it as a server and those after it as a client. This would guarantee that the client-server digraph would be free of circuits. Secondly the program checks for conformance to the basic CSP specifications using the channel bundles that have just been calculated.

The first part of the algorithm, that which calculates the channel bundles of each process, has limitations. It will not succeed in correctly identifying client and server channel bundles for certain valid basic client-server networks. There are two possible reasons for this. The first is that it is assumed that there is no *polling* on client or server channels. (By polling we mean a process communicating on a channel when in an unstable state, for instance if it is waiting for some concealed time-out event.) The second, which is less important, is only likely to arise due to a coding error and is described below.

However the method for verifying that a process with *given* client and server channel bundles obeys the basic protocol is precise, and will work for any basic client-server network. It is a simple application of the specification checking technique described on page 71.

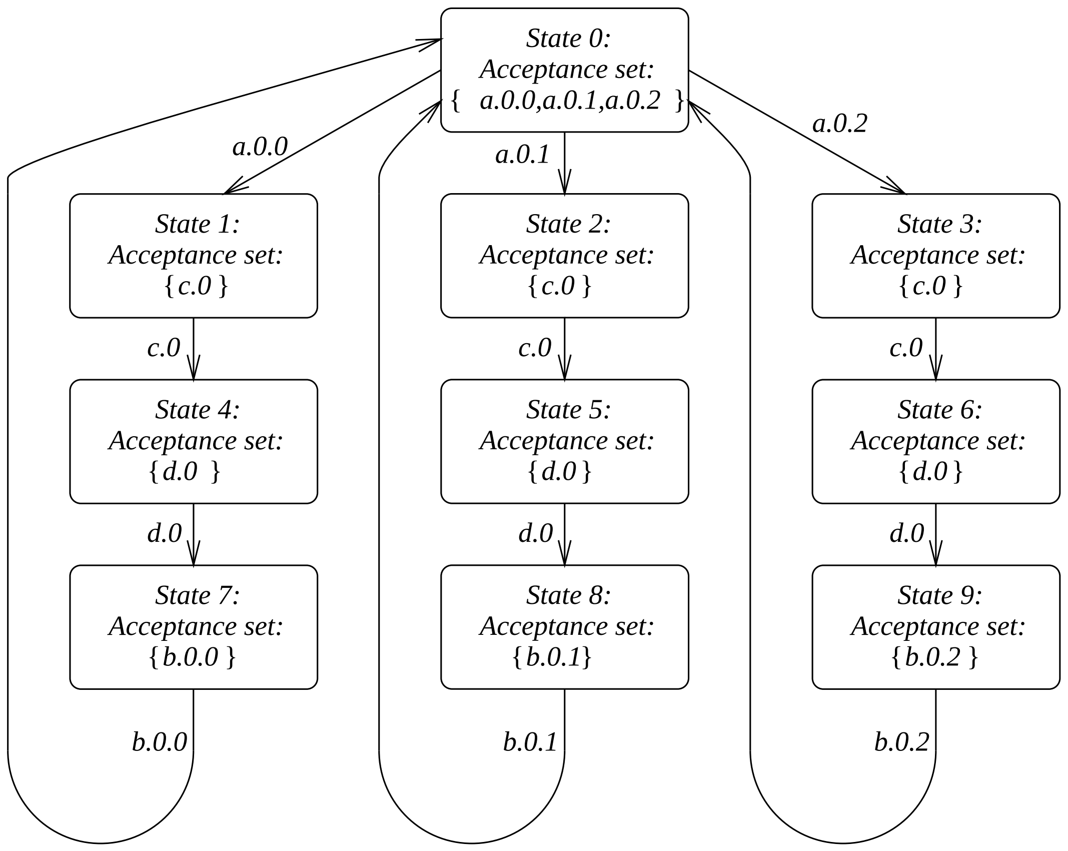
To assist with explaining this algorithm, we shall consider its application to the simple process farm described in chapter 2. The machine readable description of this network is as follows.

```
iset = {0,1,2,3,4}
jset = {0,1,2}
pragma channel a,b: iset.jset
pragma channel c,d: iset

WORKER(i,j) = a.i.j -> b.i.j -> WORKER(i,j)

FOREMAN(i) = [] j:jset @ (a.i.j -> c.i ->
             d.i -> b.i.j -> FOREMAN(i))

FARMER = [] i:iset @ (c.i -> d.i -> FARMER)

--+ WORKER(0,0),WORKER(0,1),WORKER(0,2),
--+ WORKER(1,0),WORKER(1,1),WORKER(1,2),
--+ WORKER(2,0),WORKER(2,1),WORKER(2,2),
--+ WORKER(3,0),WORKER(3,1),WORKER(3,2),
--+ WORKER(4,0),WORKER(4,1),WORKER(4,2),
--+ FOREMAN(0),FOREMAN(1),FOREMAN(2),FOREMAN(3),FOREMAN(4)
--+ FARMER
```

The normal form transition system for process *FOREMAN*(0) is illustrated in figure 3.8.

In order to try to establish the client and server bundles of a network the following steps are performed

Figure 3.8: Normal Form Transition System for *FOREMAN*(*0*)



1. For each process P in the network list, the set of channels which it uses to communicate with predecessors in the list is compiled: $B(P)$. This should represent the union of channels in P's server bundles, which must be disjoint, *i.e.* there is no channel shared by two server bundles.

   For process *FOREMAN*$(\mathit{0})$ we find that

   $$B(\textit{FOREMAN}(\mathit{0})) = \{a.\mathit{0}.\mathit{0}, b.\mathit{0}.\mathit{0}, a.\mathit{0}.\mathit{1}, b.\mathit{0}.\mathit{1}, a.\mathit{0}.\mathit{2}, b.\mathit{0}.\mathit{2}\}$$

2. For each process P, we start at its initial state and perform a depth-first search until we find a state S where P can accept communication on a server channel, *i.e.* there is a minimal acceptance set A which intersects with $B(P)$. By rule **(b)** of the basic client-server definition, $A \cap B(P)$ should consist of all the server requisition and drip channels of P. (This assumes that there is no communication by polling, in which case a server requisition or drip might have already occurred without having appeared in a minimal acceptance set.)

   Process *FOREMAN*$(\mathit{0})$ accepts communication on server channels while in its initial state, where it has a minimal acceptance set $A = \{a.\mathit{0}.\mathit{0}, a.\mathit{0}.\mathit{1}, a.\mathit{0}.\mathit{2}\}$.

3. For each channel c in $A \cap B(P)$ we take the corresponding transition from state S to a new state S'. We then construct a server bundle from c by performing a DFS, rooted at S', to find a successor state where P has a transition on some server channel c'. If c' lies in $A \cap B(P)$, *i.e* it is a requisition or a drip, then c must be a drip, otherwise (c, c') is a requisition-acknowledge bundle. If, however, the DFS terminates without finding another communication on a server channel it means that the process might never be able to communicate on a server channel again after performing event c. In this case we take c to be a drip channel. It is theoretically possible that this is incorrect and that c is actually a requisition channel, but in practice this is most likely to be a coding error in process P.

   Applying this step to *FOREMAN*$(\mathit{0})$ involves performing DFS searches rooted at states 1, 2, and 3 to find the next state where a server event may be performed. In each case a new server event is discovered (in states 7, 8 and 9 respectively) which results in the construction of three requisition-acknowledge bundles for the process, as follows:

   $$\textit{servers}(\textit{FOREMAN}(\mathit{0})) = \{\langle a.\mathit{0}.\mathit{0}, b.\mathit{0}.\mathit{0}\rangle, \langle a.\mathit{0}.\mathit{1}, b.\mathit{0}.\mathit{1}\rangle, \langle a.\mathit{0}.\mathit{2}, b.\mathit{0}.\mathit{2}\rangle\}$$

4. Having calculated the server channel bundles of a process P, we must check that they are disjoint, and that their union is $B(P)$. Both these properties are clearly satisfied for *FOREMAN*$(\mathit{0})$.

5. The next step is to assign each server bundle of P to another process as a client bundle. This is done by checking that the channels which form each server bundle belong to the alphabet of some preceding process in the list. If there is any server bundle which cannot be matched in this way then something is wrong with the network being checked. The three server bundles of *FOREMAN*(0) are allocated as client bundles to *WORKER*$(0, 0)$, *WORKER*$(0, 1)$ and *WORKER*$(0, 2)$ respectively.

If each stage has been successful, then we shall have calculated a set of client and server bundles for each process, which can now be checked against the basic protocol. However it is possible that this procedure might have failed even if the processes were valid, for the two reasons given above. It is important to make clear that this limitation could never result in Deadlock Checker passing a network as being deadlock-free when it actually deadlocks. The restriction could easily be overcome by requiring the client and server bundles to be explicitly defined in the original CSP network script, although this would be inconvenient to the user. Perhaps both options should be offered in a future version of Deadlock Checker. (However the more general SDD algorithm, which will be described below, can correctly identify deadlock-freedom for any basic client-server network, regardless of the order in which the processes are supplied.)

The second phase, which is checking that each process obeys the basic client-server protocol, is a straightforward application of the CSP specification checking technique described on page 71, using the formal definition of the rules of the basic client-server protocol, recast in terms of incremental trace functions.

To demonstrate the tool in action again, here is the analysis of the simple process farm.

```
Command (h for help, q to quit):l farm.net
Command (h for help, q to quit):w
```

For each process in the network a report of the following form is returned

```
Process FOREMAN(0) obeys client-server protocol:
  clients(FOREMAN(0)) = {<c.0,d.0>}
  servers(FOREMAN(0)) = {<a.0.0,b.0.0>,<a.0.1,b.0.1>,<a.0.2,b.0.2>}
```

As each process satisfies the protocol the program concludes that the network will never deadlock.

```
Network farm.net is deadlock-free
```

We shall now estimate the complexity of the algorithm for checking adherence to the basic client-server protocol with the usual assumptions about the number of states and events of each process. Calculating the set of server channels, $B(P_i)$, for each process P_i, can be done in constant time, by making use of arrays *first* and *second* that were set up in the course of testing for triple-disjointedness (page 70). Once this set has been separated into server bundles for P_i, by local analysis, these may each be matched up with a preceding process in the network in the same manner. The act of checking each process for conformance to the protocol, is again purely local to each process and so $O(n)$. Hence the overall complexity is $O(n)$.

Network Decomposition

Deadlock Checker implements the method for factorising deadlock analysis of Brookes and Roscoe (theorem 6, page 32). This involves finding all the *disconnecting edges* of the network communication graph. Any such edge that is shown to be *conflict-free* may be removed. Deadlock analysis is then reduced to checking that each of the remaining network fragments (*essential components*) is deadlock-free. First we need to construct the communication graph for the network, and calculate its *vocabulary* Λ. This is straightforward given the alphabet of each process, which is calculated at the compilation stage.

Finding the disconnecting edges of the graph can be done in linear time, using a variant of the DFS algorithm. This is described in appendix B. It is then required to check that the pair of processes (P, Q) which constitutes each disconnecting edge is conflict-free. This is done by checking that for every state σ of the subnetwork $\langle P, Q\rangle$ the following condition holds.

$$\neg(P \overset{\sigma,\Lambda}{\rightarrow} \bullet Q \wedge Q \overset{\sigma,\Lambda}{\rightarrow} \bullet P)$$

The specification checking technique described on page 71 is applied here. Any disconnecting edge which is found to be conflict-free is removed from the communication graph. When this phase is finished the DFS algorithm is employed once again to assemble the residual components.

The subnetwork that each essential component represents is then assigned a name, and placed on a ‘stack’ of networks. It may then be analysed by other methods. Deadlock Checker maintains a tree-structure on this stack for hierarchical proofs. So, if and when deadlock-freedom has been established for each essential component, the original network will be reported as being deadlock-free.

The following example demonstrates the construction of a hierarchical proof using Deadlock Checker. Consider the Telephoning, Arm-Wrestling, Dining Philosophers. This is a system constructed from two tables of arm-wrestling philosophers, with a telephone link added between the two most senior philosophers. The CSP code is as follows

```
PHILNAMES= {0,1,2,3,4}
FORKNAMES = {0,1,2,3,4}
TABLENAMES = {A,B}
```

```
pragma channel eats:TABLENAMES.PHILNAMES
pragma channel takes,drops:TABLENAMES.PHILNAMES.FORKNAMES
pragma channel wrestles:TABLENAMES.PHILNAMES.PHILNAMES
pragma channel phone

-- Junior philosopher: may challenge any of his seniors to an
-- arm-wrestling contest, between meals. He is left handed for
-- adherence to Resource Allocation Protocol.

JPHIL(x) = takes.x.0.4 -> takes.x.0.0 -> eats.x.0 ->
           drops.x.0.0 -> drops.x.0.4 -> JPHIL(x) |~|
           ( |~| j:{j | j <- PHILNAMES, 0<j} @ wrestles.x.0.j ->
             JPHIL(x) )

-- Intermediate philosopher: may challenge any of his seniors to
-- an arm-wrestling contest or accept a challenge from a junior,
-- between meals.

PHIL(i,x) = ( takes.x.i.i -> takes.x.i.((i-1)%5) -> eats.x.i ->
              drops.x.i.((i-1)%5) -> drops.x.i.i -> PHIL(i,x) |~|
              ( |~| j:{j | j <- PHILNAMES, i<j} @ wrestles.x.i.j ->
                PHIL(i,x) ) ) []
            ( [] j:{j | j <- PHILNAMES, j<i} @ wrestles.x.j.i ->
              PHIL(i,x) )

-- Senior philosopher: accepts arm-wrestling challenges from his
-- juniors between meals; may also telephone senior philosopher
-- on other table or accept a call from him between meals.

SPHIL(x) = ( takes.x.4.4 -> takes.x.4.3 -> eats.x.4 ->
             drops.x.4.3 -> drops.x.4.4 -> SPHIL(x) ) []
           phone -> SPHIL(x) []
           ( [] j:{j | j <- PHILNAMES, j<4} @ wrestles.x.j.4 ->
             SPHIL(x) )

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

--+ JPHIL(A),PHIL(1,A),PHIL(2,A),PHIL(3,A),SPHIL(A)
--+ JPHIL(B),PHIL(1,B),PHIL(2,B),PHIL(3,B),SPHIL(B)
--+ FORK(0,A),FORK(1,A),FORK(2,A),FORK(3,A),FORK(4,A)
--+ FORK(0,B),FORK(1,B),FORK(2,B),FORK(3,B),FORK(4,B)
```

The first stage of proving this network deadlock-free is to separate it into essential components (which in this case are the two tables of philosophers and forks).

```
Command (h for help, q to quit):l armphonephils.net
Command (h for help, q to quit):d
Network armphonephils.net is triple-disjoint
Network armphonephils.net is busy
SPHIL(A) and SPHIL(B) are conflict-free wrt vocab
Deadlock analysis reduced to:
<JPHIL(A), PHIL(1,A), PHIL(2,A), PHIL(3,A), SPHIL(A),
 FORK(0,A), FORK(1,A), FORK(2,A), FORK(3,A), FORK(4,A)>
<JPHIL(B), PHIL(1,B), PHIL(2,B), PHIL(3,B), SPHIL(B),
 FORK(0,B), FORK(1,B), FORK(2,B), FORK(3,B), FORK(4,B)>
```

The two new subnetworks will have now been added to the stack. One of these is selected and then analysed first as a user resource network, and then as a client-server network once its resources have been stripped away.

```
Command (h for help, q to quit):n (list networks)
armphonephils.net (unresolved)
armphonephils.net_0 (unresolved)   essential component
armphonephils.net_1 (unresolved)   essential component
Command (h for help, q to quit):s armphonephils.net_0
Command (h for help, q to quit):a
Network armphonephils.net_0 is busy
Network armphonephils.net_0 is triple-disjoint
Process FORK(4,A) acts as a resource
...
Process SPHIL(A) is not a resource
User process SPHIL(A) obeys resource allocation protocol
...
Deadlock analysis reduces to:
<JPHIL(A), PHIL(1,A), PHIL(2,A), PHIL(3,A), SPHIL(A)>

Command (h for help, q to quit):n
armphonephils.net (unresolved)
armphonephils.net_0 (unresolved)
armphonephils.net_1 (unresolved)
armphonephils.net_0_3 (unresolved)     resources stripped
Command (h for help, q to quit):c (display current network)
armphonephils.net_0_3
Command (h for help, q to quit):w
Network armphonephils.net_0_3 is busy
Network armphonephils.net_0_3 is triple-disjoint
Process JPHIL(A) obeys client-server protocol
clients(JPHIL(A)) =
{<wrestles.A.0.4>, <wrestles.A.0.3>,
 <wrestles.A.0.2>, <wrestles.A.0.1>}
```

```
servers(JPHIL(A)) = {}
...
Network armphonephils.net_0_3 is deadlock-free
Network armphonephils.net_0 is deadlock_free
```

To complete the deadlock analysis the other essential component is analysed in the same manner

```
Command (h for help, q to quit):s armphonephils.net_1
Command (h for help, q to quit):a
...
Command (h for help, q to quit):w
...
Network armphonephils.net_1_4 is deadlock-free
Network armphonephils.net_1 is deadlock_free
Network armphonephils.net is deadlock_free
```

The proof of deadlock-freedom for the network of Telephoning, Arm-wrestling, Dining Philosophers has now been completed.

The algorithm for network decomposition requires the construction of the network communication graph and vocabulary. Using the following algorithm, it is possible to do this with complexity $O(n\mathit{log}(n))$.

1. Start with the two arrays, *first* and *second*, that were constructed in order to establish triple disjointedness. Scan the two arrays to construct the list of pairs of the form $(\mathit{first}(e), \mathit{second}(e))$ such that both elements of the pair have been defined. This list will contain all the edges of the communication graph, but some of them may be duplicated. The set of values of e which contribute to this list is the vocabulary of the network.

2. Purge duplicate pairs from the list by performing a merge-sort (as described in [Paulson 1991]). This will result in a list of the edges in the communication graph.

The first step of this algorithm has complexity $O(n)$, where n is now taken as the number of edges in the communication graph; the second, which involves performing a merge-sort, has complexity $O(n\mathit{log}(n))$.

To complete the network decomposition, local checks of process pairs remain to be done, which is $O(n)$, and also some global graph operations, which can also be done in $O(n)$ using the DFS. So network decomposition can be done with overall complexity $O(n\mathit{log}(n))$.

Restricting a Network to its Vocabulary

Deadlock Checker also has a feature to restrict a network to its vocabulary (only shared events visible). By lemma 5 (page 32) we know that if a network transformed in this way is deadlock-free then so must have been the original network. This is useful, for instance, in the case of a network containing a cyclic-PO essential component, where some of the processes have had extra channels added for communication with processes in other essential components, which are not used according to the cyclic-PO paradigm.

However it is possible for the act of hiding these extra channels to introduce divergence, which renders the resulting network unsuitable for deadlock analysis by our methods. This only happens when an arbitrarily long sequence of communications on the external channels is possible.

The technique that we use to restrict a network to its vocabulary comprises the following steps

1. The vocabulary of the network Λ is calculated (those events which occur in the alphabet of two processes).

2. For each process P_i all events in $\alpha P - \Lambda$ are hidden. In the normal-form transition system for P this involves relabelling with τ those transitions labelled with any of these events and removing acceptance sets which include these events.

3. The resulting transition system then needs to be renormalised. This is performed using Roscoe's algorithm as described in section 3.1.

4. The transformed network is placed on Deadlock Checker's network stack. If it is subsequently proven deadlock-free then so must be the original network.

Checking for Livelock-Freedom

Deadlock Checker does not overlook the important property of livelock-freedom. We implement the proof rule of Roscoe (theorem 5, page 31) which works in many cases. The order in which the processes are supplied is significant here. The intention is to establish divergence-freedom after all internal communications have been hidden. To do this, we need to show that no process can communicate indefinitely with those before it in the network list, as follows.

1. For each process P_i we calculate the subset of its alphabet shared with predecessors in the process list and call this N_i.

2. We then consider the subgraph of the normal-form transition system of P_i containing only those arcs labelled with events which lie in N_i.

3. If this subgraph contains no circuit then P_i cannot communicate indefinitely with its predecessors in the network list.

3.4 Towards a General Purpose Algorithm

The SDD algorithm

The tools described above are useful for proving deadlock-freedom for networks constructed according to rigid design rules. But they do not allow for any improvisation by the creative programmer. The only scope for improvement is the addition of checking code for extra design rules, as and when required.

Despite these limitations, the design rules that are understood by Deadlock Checker enable the automatic proof of deadlock-freedom for networks of an unprecedented size.

In this section we shall describe the development of an alternative algorithm, which has no knowledge of design rules, and yet turns out to be able to do much offered by the above tools, and more besides. A characteristic of deadlock-states of busy, triple-disjoint networks, is that they involve a cycle of ungranted requests (theorem 1, page 29). So if we can prove that a network can never have a cycle of ungranted requests, then it is deadlock-free. This is the fundamental principle which underlies the proof technique of variant functions (theorem 2, page 29).

We now present a closely related alternative to variants, the *SDD* algorithm. This attempts to prove deadlock-freedom by forming a *state-dependence digraph*. This is basically a kind of giant wait-for digraph which instead of having just a single vertex to represent a process has a different vertex for each minimal acceptance set of each normal-form state.

1. Starting with a network of normalised transition systems $\langle P_1, P_2, ...P_N \rangle$ we form the communication graph G, and a digraph, *SDD*, which is initially empty.

2. For each edge (P, P') of G we form the set $D(P, P')$ of all normal-form state pairs (S, S') that processes P and P' can be in simultaneously.

3. For each pair (S, S') in each $D(P, P')$, for each minimal acceptance set A of S and for each minimal acceptance set A' of S', if P has an ungranted request to P', with respect to Λ – the vocabulary of the network, add arc $((P, S, A), (P', S', A'))$ to digraph *SDD*. And if P' has an ungranted request to P, with respect to Λ, add arc $((P', S', A'), (P, S, A))$.

4. We now have constructed a digraph, *SDD*. If this is circuit-free the network is reported as being deadlock-free.

Theorem 10 *A busy, triple-disjoint network, which has a circuit-free state dependence digraph, is deadlock-free*

Proof. Consider a busy, triple-disjoint network $V = \langle P_1, ..P_n \rangle$. Suppose that V has a deadlock state

$$\sigma = (s, \langle X_1, ..X_n \rangle)$$

In this state there is a cycle of ungranted requests

$$P_{i_1} \xrightarrow{\sigma,\Lambda} \bullet P_{i_2} \xrightarrow{\sigma,\Lambda} ..P_{i_k} \xrightarrow{\sigma,\Lambda} \bullet P_{i_1}$$

Where each process P_{i_h} has performed trace $s \upharpoonright \alpha P_{i_h}$, and is refusing set X_{i_h}. Let this trace and refusal set correspond to state and acceptance set (S_{i_h}, A_{i_h}) of the normal form transition system for P_{i_h}. As P_{i_h} has an ungranted request to $P_{i_{h+1}}$ in state σ, the analysis of the two processes will produce an arc from vertex $(P_{i_h}, S_{i_h}, A_{i_h})$ to vertex $(P_{i_{h+1}}, S_{i_{h+1}}, A_{i_{h+1}})$ in the state dependence digraph. Performing this analysis of each pair of consecutive processes in the cycle of ungranted requests will result in a circuit in the state dependence digraph.

So we have shown that if there is a deadlock-state of V, then there is a circuit in its state dependence digraph. This completes the proof□

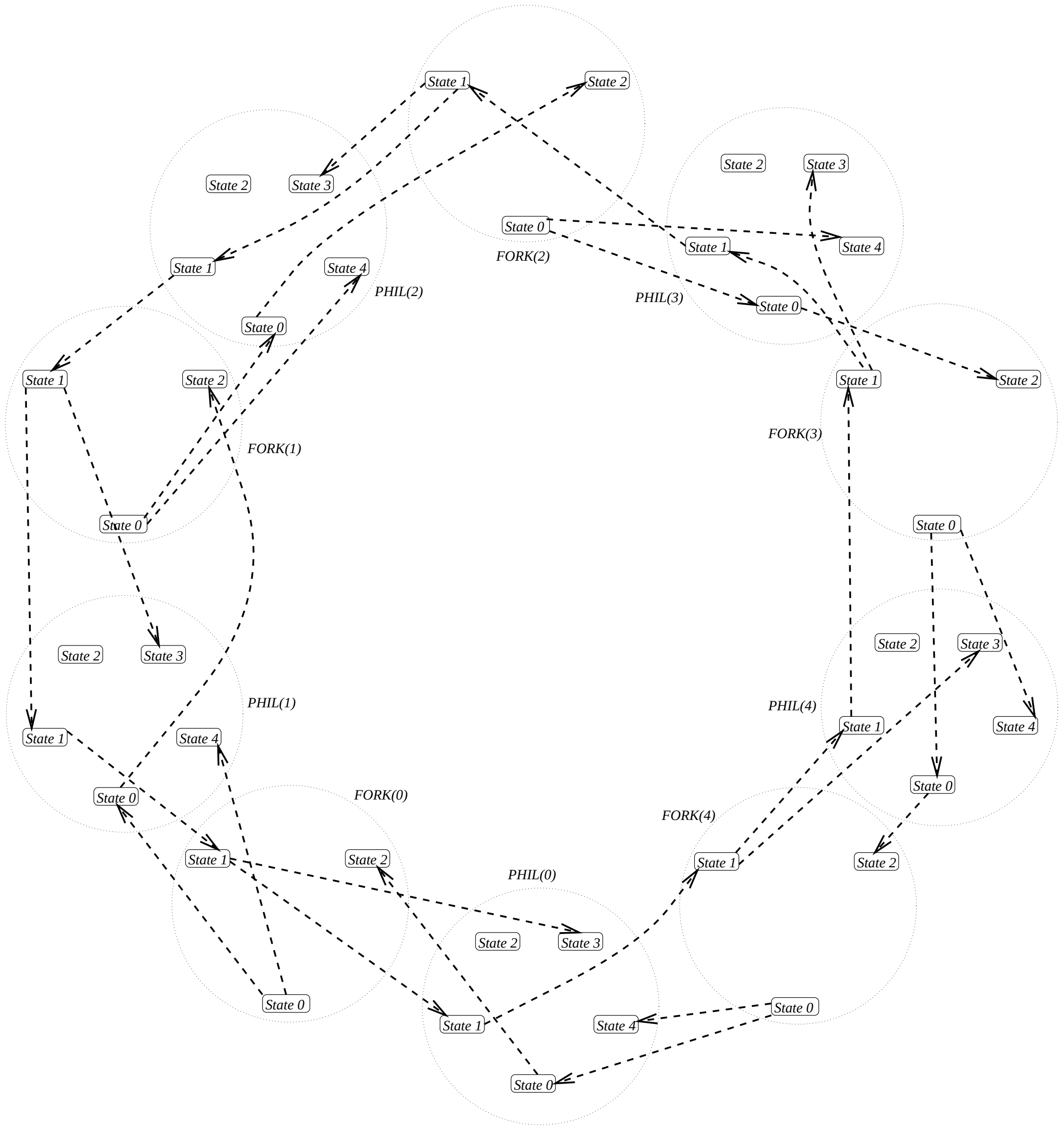
Here is what happens when we apply the SDD algorithm to the Dining Philosophers network.

```
Command (h for help, q to quit):l phils.net
Command (h for help, q to quit):v
Network phils.net is triple-disjoint
Network phils.net is busy
Found possible cycle of ungranted requests:
FORK(0) ready to do drops.0.0 blocked by PHIL(0)
PHIL(0) ready to do takes.0.4 blocked by FORK(4)
FORK(4) ready to do drops.4.4 blocked by PHIL(4)
PHIL(4) ready to do takes.4.3 blocked by FORK(3)
FORK(3) ready to do drops.3.3 blocked by PHIL(3)
PHIL(3) ready to do takes.3.2 blocked by FORK(2)
FORK(2) ready to do drops.2.2 blocked by PHIL(2)
PHIL(2) ready to do takes.2.1 blocked by FORK(1)
FORK(1) ready to do drops.1.1 blocked by PHIL(1)
PHIL(1) ready to do takes.1.0 blocked by FORK(0)
```

The state dependence digraph for the Dining Philosophers is shown in figure 3.9, constructed from the normal form transition systems shown in figure 3.4. (As each process in the network is deterministic there is exactly one minimal acceptance set corresponding to each state. In the case of a non-deterministic system there would need to be more than one vertex to represent certain states in the state-dependence digraph.) It contains a single circuit, representing the situation where each philosopher has picked up his left fork.

Although SDD works in many cases where the variant functions could have been used, it is not quite so powerful, because of the fact that an arbitrary number of maximal failures of a process can be mapped onto a single state in the normal form. One example of this is that the SDD technique will often fail for networks of cyclic processes which

Figure 3.9: Construction of SDD for Dining Philosophers



are amenable to variant function technique. It will sometimes find a 'phantom' cycle of ungranted requests which cannot actually occur. For instance, consider what happens when we apply the algorithm to the deadlock-free toroidal cellular array.

```
Command (h for help, q to quit):l torus.net
Command (h for help, q to quit):v
Found possible cycle of ungranted requests:
CELL(2,3) ready to do e.2.3.right e.3.3.left
          blocked by CELL(3,3)
CELL(3,3) ready to do e.3.2.down e.3.3.up
          blocked by CELL(3,2)
CELL(3,2) ready to do e.0.2.left e.3.2.right
          blocked by CELL(0,2)
CELL(0,2) ready to do e.0.1.down e.0.2.up
          blocked by CELL(0,1)
CELL(0,1) ready to do e.0.1.right e.1.1.left
          blocked by CELL(1,1)
CELL(1,1) ready to do e.1.0.down e.1.1.up
          blocked by CELL(1,0)
CELL(1,0) ready to do e.1.0.right e.2.0.left
          blocked by CELL(2,0)
CELL(2,0) ready to do e.2.0.up e.2.3.down
          blocked by CELL(2,3)
```

The cycle of ungranted requests that has been reported cannot actually occur. Process `CELL(2,3)` can only have an ungranted request to `CELL(3,3)` if the latter has yet to complete its previous communication cycle. Also no cyclic-PO process can ever have an ungranted request to a another one that has completed more cycles. Following the potential cycle of ungranted requests in this way actually takes us back to the original process `CELL(2,3)` in the same state but on an earlier cycle. Clearly a process cannot be on two *I/O* cycles simultaneously, so the potential cycle of ungranted requests is unreal. What it actually represents is a spiral of ungranted requests backwards in time.

We shall address this problem by refining the algorithm later on, but first let us explore the power of this prototype version in relation to some other design rules.

Applications of the SDD algorithm

Theorem 11 *Any circuit-free client-server network composed from finite-state 'basic' processes has a circuit-free state dependence digraph*

Proof. Consider a basic client-server network $V = \langle P_1, .., P_n \rangle$, with a circuit-free topology. This is deadlock-free by rule 7 (page 47). We shall show that the state dependence digraph of V can never have a path of length 2, going through states of processes

P_i, P_j, P_k, such that the relationship between P_i and P_j is client to server and the relationship between P_j and P_k is server to client. Then the circuit-freedom of the state dependence digraph will follow as a direct consequence of the circuit-freedom of the client-server digraph.

So first suppose that there is an arc in the state dependence digraph

$$((P_j, S_j, A_j), (P_k, S_k, A_k))$$

where P_j communicates with P_k as server to client. This arc represents a potential ungranted request in the subnetwork

$$\langle P_j, P_k \rangle$$

and we can deduce, from the definition of the basic client-server protocol, that this can only occur when P_j is waiting for P_k to perform a *requisition* or *drip* event. It also follows from rule **(b)** that P_j is ready to perform all its server *requisition* and *drip* events, *i.e* they are all contained in A_j.

Now suppose that there is another arc in the state dependence digraph

$$((P_i, S_i, A_i), (P_j, S_j, A_j))$$

where P_i communicates with P_j as client to server. This arc represents an ungranted request in the subnetwork

$$\langle P_i, P_j \rangle$$

We already know that A_j contains every server *requisition* and *drip* event of P_j, so P_i must be waiting to communicate with P_j on a client *acknowledge* channel. But this is impossible by rule **(c)** of the protocol.

This contradiction means that there is no path in the state dependence digraph which goes from client to server and then from server to client. Therefore, as the client-server digraph is circuit-free, there can be no circuit in the state dependence digraph, so the network will be reported as being deadlock-free by the SDD algorithm. □

The SDD algorithm is clearly more powerful than the tool for checking deadlock-freedom in basic client-server networks. It will always work and does not require the processes to be supplied in any particular order. However, as it has no intelligence regarding the actual protocol, it will probably be less useful as a debugging aid, especially for analysing networks constructed by teams rather than by individuals.

Theorem 12 *Any finite-state user-resource network which obeys the Resource Allocation Protocol has a circuit-free state-dependence digraph.*

Proof. Consider a finite-state user-resource network which adheres to the Resource Allocation Protocol (page 56). Suppose that there is a circuit in its state-dependence digraph. Due to the bipartite nature of the network, this circuit must run through a sequence of vertices of the form

$$\langle (U_{i_1}, S_{i_1}, A_{i_1}), (R_{j_1}, S_{j_1}, A_{j_1}), .., (U_{i_k}, S_{i_k}.A_{i_k}), (R_{j_k}, S_{j_k}, A_{j_k})\rangle$$

A user process can only have an ungranted request to a resource when it is waiting to claim it. We deduce that for each process R_{j_h}, state S_{j_h} is the normal-form state where it is waiting to be released by the next process in the circuit, $U_{i_{h+1}}$ (addition modulo k), which is in state $S_{i_{h+1}}$. (See figure 3.6.)

Now the arc $((R_{j_h}, S_{j_h}, A_{j_h}), (U_{i_{h+1}}, S_{i_{h+1}}, A_{i_{h+1}}))$ represents an actual ungranted request within the subnetwork $\langle R_{i_h}, U_{i_{h+1}}\rangle$ so it must be possible for $U_{i_{h+1}}$ to be holding resource R_{j_h} in state $S_{i_{h+1}}$. As it also tries to claim resource $R_{j_{h+1}}$ from this state, it follows that $R_{j_h} > R_{j_{h+1}}$, by the terms of the protocol. Applying this result all the way around the circuit leads to the following contradiction

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

From this we conclude that the state-dependence digraph is actually circuit-free, and the network will be reported as being free of deadlock by the SDD algorithm□

In practice, the SDD algorithm also seems to be a useful tool for analysing user-resource networks, where the users communicate with each other, obeying the Extended Resource Allocation Protocol. It has no trouble with proving the Arm-Wrestling Dining Philosophers deadlock-free (or indeed the Telephoning Arm-Wrestling Dining Philosophers). It would be nice if whenever the subnetwork of user processes could be proven deadlock-free by the SDD algorithm so could be the whole network. However this is not always the case, as the following example illustrates.

```
pragma channel a,b,c,c1,r1,c2,r2

U1 = (b -> U1 |~| a -> U1) [] c1 -> r1 -> U1
U2 = (c -> U2 |~| b -> U2) [] c2 -> r2 -> U2
U3 = a -> c -> U3
R = c1 -> r1 -> R [] c2 -> r2 -> R

--+ U1, U2, U3, R
```

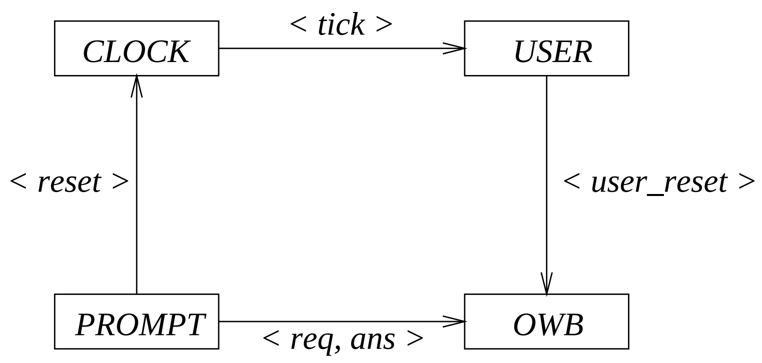
Here network $\langle U1, U2, U3\rangle$ is provably deadlock-free by the SDD algorithm, but $\langle U1, U2, U3, R\rangle$ is not, even though it obeys the Extended Resource Allocation Protocol, and so is, in fact, deadlock-free. In the former case events $c1$, $r1$, $c2$, and $r2$ lie outside the vocabulary, so there are no ungranted requests between $U1$ and $U2$ with respect to the vocabulary. But in the latter case the vocabulary includes all these events and a potential cycle of ungranted requests is reported.

```
U2 ready to do b blocked by U1
U1 ready to do a blocked by U3
U3 ready to do c blocked by U2
```

Analysing Non-Standard Networks with SDD

Welch, Justo and Willcock consider an interesting example of client-server network where the basic protocol has been slightly abused [Welch *et al* 1993]. The system comprises a *USER* process which is stimulated by regular 'ticks' from a *CLOCK* process. The *USER* process may reset the interval between ticks by means of a reset channel. Conceptually, process *USER* communicates as both a client and a server with process *CLOCK*. In order to avoid a circuit of client-server relationships a 'circuit-breaker', consisting of a one-place overwriting buffer *OWB* together with a prompter *PROMPT*, is inserted along the reset channel. The client-server digraph of the resulting system is shown in figure 3.10.

Figure 3.10: Client-Server Digraph for *CLOCK* Network



The machine-readable CSP code for this network is as follows:

```
pragma channel tock,user_reset,req,ans,reset,time_out

USER = tock -> (USER |~| user_reset -> USER)
PROMPT = req -> ans -> reset -> PROMPT
OWB = user_reset -> (req -> ans -> OWB [] OWB)
CLOCK = reset -> CLOCK [] time_out -> tock -> CLOCK

--+ PROMPT,CLOCK,USER,OWB
```

In this definition *CLOCK* has an internal event *time_out*. This event represents a signal from an internal timer process that it is time to send out the next *tock*. Each process behaves according to the basic client-server protocol, apart from *OWB*. This process will shut down service on channel *req* whenever its buffer is empty. This contravenes rule **(b)** of the protocol. Nonetheless Welch, Justo and Willcock claim that the network is deadlock-free. This cannot be shown by the algorithm which tests adherence to the basic client-server protocol, but it is no problem for the SDD algorithm.

```
Command (h for help, q to quit):l clock.net
Command (h for help, q to quit):v
Checking PROMPT with CLOCK
Checking PROMPT with OWB
Checking USER with CLOCK
Checking USER with OWB
Network clock.net is deadlock-free
```

This is a good example of a situation where a programmer's intuition has been automatically confirmed by SDD, avoiding the need for an analytic proof. As the system was designed with an aircraft control system in mind, this could be useful.

Accommodating Cyclic Processes

In general, the SDD is unable to prove networks of cyclic processes deadlock-free. However these are an important ingredient of many parallel algorithms. Fortunately we can remedy the problem as follows. First of all we extend the network analysis to produce a state-dependence digraph with coloured arcs.

Remember that arc $((P, S, A), (P', S', A'))$, represents an ungranted request from process P in state S to process P' in state S'. If this can occur only when P and P' have each visited their initial state exactly the same number of times, we colour the arc *red*. Alternatively if P must have visited its initial state more times than P' we colour the arc *green*. Otherwise the arc is coloured *blue* to represent uncertainty.

The arc colouring is calculated in a similar way to the technique for specification checking described on page 71. First we construct a set of records of the form

$$\langle \sigma_P, \sigma_{P'}, \mathit{count} \rangle$$

Each record contains a pair of states that P and P' may simultaneously be at together with a numeric labelling: *count*. This represents the number of times that the initial state of P has been 'crossed' minus the number of times that the initial state of P' has been crossed. A process is said to have crossed its initial state whenever it performs an event which returns it to its initial state. If the numeric labelling of state pairs is found to be inconsistent, *i.e* two records are found $\langle \sigma_P, \sigma_{P'}, \mathit{count} \rangle$ and $\langle \sigma_P, \sigma_{P'}, \mathit{count}' \rangle$ with $\mathit{count} \neq \mathit{count}'$, then *all* the numbering information regarding states of $\langle P, P' \rangle$ is discarded. Any ungranted request found between the two processes is regarded as being 'uncertain' and coloured blue. If, however, a consistent numbering is discovered it may be used to colour ungranted requests red, green or blue in the manner described above.

To illustrate how the coloured state dependence digraph is constructed, let us return to the example of the two-place buffer from page 73. Recall that this was defined as follows

$$\mathit{LEFT} \quad = \quad \mathit{in} \rightarrow \mathit{mid} \rightarrow \mathit{LEFT}$$

$$\begin{aligned}\alpha \mathit{LEFT} &= \{\mathit{in},\mathit{mid}\}\\[1em] \mathit{RIGHT} &= \mathit{mid} \rightarrow \mathit{out} \rightarrow \mathit{RIGHT}\\ \alpha\mathit{RIGHT} &= \{\mathit{mid},\mathit{out}\}\\[1em] V &= \langle \mathit{LEFT}, \mathit{RIGHT}\rangle\end{aligned}$$

The exhaustive search for records of the form $(\sigma_{LEFT}, \sigma_{RIGHT}, \mathit{count})$ proceeds as follows. First we have

$$\mathit{pending} = \{(0,0,0)\},\ \mathit{done} = \{\}$$

Check $(0,0,0)$; possible transition is *in*; neither initial state is crossed; leads to record: $(1,0,0)$. Now we have

$$\mathit{pending} = \{(1,0,0)\},\ \mathit{done} = \{(0,0,0)\}$$

Check $(1,0,0)$; possible transition is *mid*; initial state of *LEFT* is crossed; leads to record: $(0,1,1)$. Now we have

$$\mathit{pending} = \{(0,1,1)\},\ \mathit{done} = \{(0,0,0),(1,0,0)\}$$

Check $(0,1,1)$; possible transitions are *in* and *out*; if *in* is performed neither initial state is crossed but if *out* is performed initial state of *RIGHT* is crossed; lead to records $(1,1,1)$ and $(0,0,0)$. Now we have

$$\mathit{pending} = \{(1,1,1)\},\ \mathit{done} = \{(0,0,0),(1,0,0),(0,1,1)\}$$

Check $(1,1,1)$; possible transition is *out*; initial state of *RIGHT* is crossed; leads to record $(1,0,0)$. Now we have

$$\mathit{pending} = \{\},\ \mathit{done} = \{(0,0,0),(1,0,0),(0,1,1),(1,1,1)\}$$

Now we have discovered all the state pairs in which processes *LEFT* and *RIGHT* may simultaneously rest. For each pair we have found an invariant property *count* which represents the number of times that *LEFT* has visited its initial state more than *RIGHT*.

Suppose that *LEFT* and *RIGHT* are embedded in some network V' which has a vocabulary Λ containing events *in* and *out*. We find that state pair $(0,0)$ involves an ungranted request from *RIGHT* to *LEFT* with respect to Λ. This is represented as a red arc in the coloured state dependence digraph because the value of *count* is always zero for this state pair. We also find that state pair $(1,1)$ involves an ungranted request from *LEFT* to *RIGHT* with respect to Λ. This is represented as a green arc in the coloured state dependence digraph because the value of *count* is always 1 for this state pair. The other state pairs, $(0,1)$ and $(1,0)$ do not involve any ungranted requests.

So the analysis of *LEFT* and *RIGHT* would result in the addition of the following arcs to the coloured state dependence digraph for V'.

$$\text{Red arc:} \quad \left(\begin{pmatrix} \text{Process:} & \mathit{RIGHT} \\ \text{State:} & \mathit{0} \\ \text{Acceptance set:} & \{\mathit{mid}\} \end{pmatrix}, \begin{pmatrix} \text{Process:} & \mathit{LEFT} \\ \text{State:} & \mathit{0} \\ \text{Acceptance set:} & \{\mathit{in}\} \end{pmatrix}\right)$$

$$\text{Green arc:} \quad \left(\begin{pmatrix} \text{Process:} & \mathit{LEFT} \\ \text{State:} & \mathit{1} \\ \text{Acceptance set:} & \{\mathit{mid}\} \end{pmatrix}, \begin{pmatrix} \text{Process:} & \mathit{RIGHT} \\ \text{State:} & \mathit{1} \\ \text{Acceptance set:} & \{\mathit{out}\} \end{pmatrix}\right)$$

When the same analysis is applied to processes *FORK*$(\mathit{0})$ and *PHIL*$(\mathit{0})$ in the Dining Philosophers network, inconsistencies are found in the *count* variable. This obvious by the fact that process *FORK*$(\mathit{0})$ may cross its initial state any number of times, by cycling on events $\mathit{takes.1.0}$ and $\mathit{drops.1.0}$, before process *FORK*$(\mathit{0})$ has performed any event at all. In this case all the ungranted requests detected between the processes would appear as blue arcs in the coloured state dependence digraph.

Any circuit in the *coloured* state-dependence digraph containing a blue arc remains a potential cause of deadlock, so does any circuit which contains only red arcs. But a circuit containing no blue arcs and at least one green arc does not represent a cycle of ungranted requests in the network, for the ungranted requests cannot all occur simultaneously.

We check for deadlock-freedom as follows. First we use a variant of the DFS to remove all those arcs from the digraph which do not lie on any circuit (described in appendix B). If any blue arc remains then there is potential for deadlock. Otherwise all the remaining arcs must be red or green. The only risk of deadlock in this case is if there is a circuit consisting only of red arcs, so we remove all the greens arcs and then see whether any circuit still remains.

Given that the motivation for the CSDD algorithm was to be able to handle cyclic processes, the following result is not altogether surprising.

Theorem 13 *Take a deadlock-free network of cyclic-LOP processes. Its coloured state-dependence digraph contains neither a blue arc nor a circuit of red arcs.*

Proof. Let $V = \langle P_1, ..P_n \rangle$ be a deadlock-free network of cyclic-LOP processes. Each process is finite-state by definition. We observe that although a cyclic-LOP process does not necessarily visit the same states on each cycle, its initial state is always crossed between cycles. Between any two visits to a particular state, such a process performs every event in its alphabet the same number of times, equal to the number of times that it has crossed its initial state.

Consider a subnetwork of two communicating cyclic-LOP processes, $\langle P_i, P_j \rangle$. Suppose that these processes may simultaneously be in states σ_{P_i} and σ_{P_j}. Between two particular visits to this state pair, suppose that P_i performs m_i cycles of events in αP_i

and P_j performs m_j cycles of events in αP_j. As the processes communicate with each other we have

$$\alpha P_i \cap \alpha P_j \neq \{\}$$

Let c be an event from $\alpha P_i \cap \alpha P_j$. Between the two visits to the state pair, event c has been performed m_i times by P_i and has also been performed m_j times by P_j. So $m_i = m_j$; in other words P_i and P_j must each cross their initial state the same number of times between any two visits to a given pair of states.

This means that when the subnetwork $\langle P_i, P_j \rangle$ is analysed for records of the form

$$(\sigma_{P_i}, \sigma_{P_j}, \mathit{count})$$

where *count* represents the number of times more that P_i has crossed its initial state than P_j, we shall find that *count* is invariant for any pair of states.

A cyclic-LOP process can only have an ungranted request to another process which has performed the same number of cycles or one less cycle. It follows that the coloured state dependence digraph for V contains only red and green arcs, no blue arcs. Suppose that a circuit of red arcs were found. This would correspond to a sequence of processes

$$\langle P_{i_1}, ..P_{i_k}, P_{i_1} \rangle$$

such that each process P_{i_h} would have a state where it could perform some event c_h with its successor in the sequence but not be able to to perform some other event c_{h-1} with its predecessor in the sequence, despite having completed the same number of cycles. This would imply the existence of a circuit in the $\rhd$ relation,

$$c_1 \rhd .. \rhd c_k \rhd c_1$$

which would contradict theorem 8 (page 41), so there can be no circuit of red arcs in the coloured state dependence digraph for V. It follows that V will be passed as deadlock-free by the CSDD algorithm□

Unlike the SDD algorithm, the CSDD algorithm has no problem with the toroidal cellular array.

```
Command (h for help, q to quit):x
Network torus.net is deadlock-free
```

Although the new algorithm can handle cyclic-LOP networks it is not guaranteed to be able to prove deadlock-freedom for cyclic-PO networks in general, as these may have legitimate cycles of ungranted requests at times, despite being deadlock-free.

Note that when it is required to use CSDD to prove deadlock-freedom for hybrid networks including cyclic subnetworks, one has to be careful that the extra communications added to the cyclic processes do not remove the property that they should each cross their initial state exactly once after each cycle.

Each stage in the analysis has $O(n)$ complexity (where n is the number of edges in the communication graph), given our usual assumptions about the number of states and events of each process, except for the construction of the communication graph and vocabulary, which we have shown to be feasible with $O(n\mathit{log}(n))$ complexity. Thus the CSDD algorithm can be performed with complexity $O(n\mathit{log}(n))$.

Allowing for Weak Conflict

Another useful way to extend the SDD algorithm is to incorporate theorem 3 (page 30). Recall that if a network is shown to be free of strong conflict then any deadlock state must contain a cycle of ungranted requests of length at least three. This means that if the state dependence digraph of a strong conflict free network contains no circuits of length three or more then the network is deadlock-free regardless of how many circuits of length two are found.

The property of strong conflict freedom may be checked during the construction of the state dependence digraph at virtually no extra cost. If a strong conflict is found then it is reported and the algorithm terminates.

Searching for circuits of length three or more in a simple digraph may be performed by the following algorithm. For each arc (v, v') of the digraph $D(V, A)$, use the DFS to look for a path from v' to v in the digraph $D(V, A - ((v', v)))$. If no such path is found then the digraph has no circuit of length three or more.

For our coloured digraph, we adapt this algorithm as follows. First we check that there is no circuit of length at least three containing a blue arc. For each blue arc (v, v') we use the DFS to look for a path from v' to v in the digraph $D(V, A - ((v', v)))$. If no such circuit is found then we remove all the blue and green arcs from the digraph and search for a circuit of length three or more in the resulting red digraph. If none is found we have proved deadlock-freedom.

In the prototype version of Deadlock Checker, this improvement has been included as part of the CSDD test, but not the SDD test. Unfortunately the technique that we use to check for circuits of length three or more, increases the complexity to $O(n^2)$, as a DFS search may now be required for each arc in the digraph. However, there may well exist a more efficient technique than this.

Potential for Further Improvement

Despite the improvements that we have made to the original SDD algorithm, the possibility remains of detecting bogus cycles of ungranted requests. One way in which this has been observed in practice has been the detection of a circuit in the state dependence digraph which crosses more than one state of the same process. It is clearly impossible for a process to be in two states at the same time so such a circuit cannot represent a *real* cycle of ungranted requests.

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

A more promising approach involving this vertex colouring is based on the concept of *request selector functions*[Dathi 1990, Roscoe 1995]. Suppose that there is some vertex of the state dependence digraph, $v = (P, S, A)$, which has outgoing arcs to vertices which have several different colours. Now suppose that we choose one particular such colour $C(v)$ and delete every outgoing arc from v that points to a vertex with a different colour from $C(v)$. If the stripped down version of the state dependence digraph which results contains no circuit then it is still the case that the network must be deadlock-free. The result still holds no matter how many vertices v are treated in this manner.

This may be informally justified as follows. What we have actually done is to choose a particular process to which P has a request, when it is accepting the events of A in state S, and to ignore requests to other processes. For any deadlock state of the network, we could still find a cycle of ungranted requests corresponding to a circuit in the stripped down state dependence digraph.

It is thought that this technique should be useful as follows. Suppose that a state-dependence digraph has been constructed and is found to contain circuits. An algorithm is envisaged which would attempt to find a sequence of vertex and colour selections leading to the removal of sufficient arcs to render the digraph circuit-free, and hence prove deadlock-freedom.

We could also extend the power of the checker to embrace the design rule of Brookes and Roscoe (theorem 4, page 30). We might do this by adding an extra dimension to the coloured state dependence digraph: arcs would either be ‘flashing’ or ‘non-flashing’. An ungranted request from a state of process P to a state of process Q would be set to be flashing only if it had been shown that Q must have communicated with P more recently than with any other process in that situation. Then any circuit of flashing arcs, of length greater than two, could not represent a real cycle of ungranted requests in the network, as the ungranted requests could not occur simultaneously. This follows from the reasoning we used to prove theorem 4.

It would also be relatively straightforward to allow for networks where the processes can terminate. Dathi defines a network to be *prudent* if no process ever tries to communicate with one that is only willing to perform event $\surd$ [Dathi 1990]. With a slight adjustment to the definition of deadlock-freedom to allow for termination, the CSDD algorithm could be implemented in exactly the same way for a network containing terminating processes which had been shown to be prudent.

Due to the exponential state explosion as a network grows in size, it seems unlikely that there is an algorithm for deciding deadlock-freedom for finite-state processes which is both efficient and *complete*. There are certain networks for which deadlock-freedom depends on some crucial property of global states. For instance the analysis of a ‘tokenring' system in [Brookes and Roscoe 1991] involves proving that there is exactly one 'token' present in any state of the system. The techniques described above, being based on local analysis, would be inadequate for this particular task. However there is certainly much scope for automatic assistance in performing analyses of this nature. The limitations of proving deadlock-freedom purely by local analysis are further discussed in [Roscoe 1995].

There is clearly potential for expanding the armoury of efficient verification techniques such as CSDD. If these are to be used in anything other than a trial and error fashion they must be backed up with further design rules which will enable networks to be built not only deadlock-free, but that may be easily verified so.