1989

Deadlock analysis in networks of communicating processes

S. D.(Stephen D.) Brookes
Carnegie Mellon University

A. W. Roscoe

Deadlock Analysis in Networks of Communicating Processes

S. D. Brookes A. W. Roscoe †

June 1989

CMU-CS-89-161

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

†Programming Research Group
Oxford University
Oxford, England

Expanded version of a paper that appeared in
Logics and Models of Concurrent Systems, Springer Verlag 1985
and as CMU-CS-85-111

Abstract

We use the failures model of CSP to describe the behaviour of a class of networks of communicating processes. This model is well suited to reasoning about the deadlock potential of networks. We introduce a number of simple conditions on networks which aid deadlock analysis either by localizing the analysis required for a proof of deadlock-freedom or by restricting the circumstances in which deadlock could occur. In particular, we formulate some simple theorems which characterize the states in which deadlock can occur, and use them to prove some theorems on the absence of global deadlock in systems. We identify a special class of unidirectional networks and develop specialized results on their deadlock-freedom. We develop more general methods based on (at most) pairwise local deadlock analysis in networks, applicable to the large class of conflict-free networks. We introduce a methodology for proving deadlock-freedom in a large network by decomposing it into subnetworks which can be analysed separately. A variety of examples is given to show the utility of these results. We compare our work with earlier work by several other authors, and make some suggestions for future research.

This research was supported in part by funds from the Computer Science Department of Carnegie Mellon University, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-87-C-1499. A. W. Roscoe gratefully acknowledges support by ONR Grant N00014-87-G-0242.The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the offical policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

DEADLOCK ANALYSIS IN NETWORKS OF COMMUNICATING PROCESSES

S. D. Brookes
Carnegie Mellon University
Pittsburgh, PA 15213

A. W. Roscoe
Programming Research Group
Oxford University
Oxford, England

0. Abstract.

We use the failures model of CSP to describe the behaviour of a class of networks of communicating processes. This model is well suited to reasoning about the deadlock potential of networks. We introduce a number of simple conditions on networks which aid deadlock analysis either by localizing the analysis required for a proof of deadlock-freedom or by restricting the circumstances in which deadlock could occur. In particular, we formulate some simple theorems which characterize the states in which deadlock can occur, and use them to prove some theorems on the absence of global deadlock in systems. We identify a special class of unidirectional networks and develop specialized results on their deadlock-freedom. We develop more general methods based on (at most) pairwise local deadlock analysis in networks, applicable to the large class of conflict-free networks. We introduce a methodology for proving deadlock-freedom in a large network by decomposing it into subnetworks which can be analysed separately. A variety of examples is given to show the utility of these results. We compare our work with earlier work by several other authors, and make some suggestions for future research.

1. Introduction.

In [4,5] we described the failures model of communicating processes and used it to describe some interesting parallel programming examples. The simple mathematical structure of this model lends itself to clean formulation of deadlock properties and to formal manipulation of process behaviour. The model is well suited, by its very construction, to reasoning about the potential or the absence of deadlock in systems of processes. In this paper we elaborate this point in some detail, developing some ideas which originated in Roscoe's thesis [17], discovering various conditions on networks which make deadlock analysis easier and which enable a more structured approach to the entire problem of deadlock analysis.

Our emphasis is on methods for proving deadlock-freedom that allow localized analysis (by focussing on small subnetworks, such as pairs of processes) and support hierarchical decomposition. There is a good reason for finding such techniques: a straightforward proof of deadlock-freedom based on the above definition must take into account all possible traces of a system; in a network with many processes the trace set may be very large, and the combinatorial explosion inherent in such an analysis may be prohibitively expensive. We provide some simple yet useful theorems which may be used to analyse networks for the potential of deadlock. We demonstrate the utility of these results by examining a variety of examples, some well known and some novel.

Outline of Paper.

After this introduction, the paper begins (Section 2) by summarizing some background material and placing this work in context. We summarize relevant notation and basic terminology on CSP and the failures semantic model, and we give a formal definition of deadlock-freedom.

Next, in Section 3, we introduce *networks* and their static *communication graphs*, and we discuss behavioural properties of networks. We define an appropriate notion of *state* for networks, and we provide a simple characterization of states in which deadlock occurs. We introduce *snapshot graphs*, which provide instantaneous pictures of the dynamic state of a network and help in visualising and analysing deadlock. The arcs in a snapshot graph are determined by the *requests* for communication currently being made among the nodes of the network. A network deadlocks if and only if all of its processes are *blocked*, in that all existing requests are *ungranted*. Some examples are given to illustrate the definitions and to show the close connections between deadlock and cycles of ungranted requests in snapshot graphs.

In Section 4 we identify several special types of deadlock-freedom property of networks, taking the topology of a network into consideration. We motivate our desire to develop methods for *hierarchical* analysis of networks, allowing the treatment of networks whose nodes are themselves built up as networks. This leads us to restrict attention to the class of *busy* networks, all of whose nodes are themselves free of deadlock. In all cases our purpose is to find methods of proving deadlock-freedom that require only local analysis, such as analysis of individual nodes or pairs of adjacent nodes in a network.

Section 5 gives elementary results on deadlock analysis, including simple but very useful techniques involving the interaction of the CSP hiding operation with deadlock. The results of this section are applied throughout the paper.

In Section 6 we restrict attention to a class of *unidirectional* networks, with the important subclass of unidirectional tree-structured networks treated as a special case. An example is worked out in detail.

In Section 7 we introduce notions of *conflict*, and develop results that are applicable to networks which are free of conflict, not just to unidirectional systems. Conflict-freedom is a pairwise checkable property, and we argue that it is sufficiently general that most interesting systems which are deadlock-free can be presented as conflict-free networks, so that our methodology is widely applicable. Again we address some examples.

In Section 8 we show how to decompose a network into regions (*essential components*) that can be treated independently in deadlock analysis, provided the interactions between regions are well behaved. This type of decomposition can be very useful in reducing the amount of combinatorial analysis required in deadlock analysis, but is only of practical benefit when a network has non-trivial essential components. Again we tackle an example. We also propose a network design rule based on these ideas, which guarantees deadlock-freedom provided a network can be built in such a way that it adheres to the design rule.

Finally, we discuss related work and point to directions for future research.

2. Background.

This paper evolved from a preliminary draft (with the same title) which was published in [2] as [6]. In this evolution some of the definitions and results have been replaced, notably by the inclusion of some simpler and sharper material on "conflicts". Closely related works applying some of the results of this paper are [8,18,19]. We use basic terminology and notation that is (for the most part) consistent with the usage in these related papers; in a few cases our notation improves slightly on that of [6], but readers familiar with the earlier version should find it easy to relate the new terminology to the old. We have tried to include enough background material to make this paper self-contained, even though this causes some overlap with the contents of [19] in particular. Although this paper is intended to be a companion to [19], there is no requirement to read that paper first.

We assume some familiarity with the material of [13], [4] or [5], where details were given of the syntax for processes in an abstract version of Hoare's language CSP (Communicating Sequential Processes), and of the mathematical construction of the failures model. Here we will provide a brief summary of terminology; the reader should consult the references for more detailed explanations.

Events, alphabets, and processes.

CSP is a language of non-deterministic communicating processes. Communication and parallel composition are taken as primitive notions. Our abstract version of the language is sometimes referred to in the literature as TCSP, or "Theoretical CSP", to distinguish it from the more concrete language introduced as CSP by Hoare in [12]. However, since Hoare himself continues to refer to our abstract version as CSP, as in [13], we will do so too.

We use P, Q, R, etc., to range over the set of CSP processes. The basic actions performed by processes are called *events*, which may be regarded as representing communications. A process may also be able to make nondeterministic choices which affect its ability to perform events. Each process is associated with an alphabet: a set of events (usually, though not necessarily, the set of events mentioned in the syntax of the process).

Two methods have been used in the literature for introducing process alphabets. In [4,5,6] for example, alphabets were introduced explicitly into the (syntax of the) parallel operator: thus a parallel composition of P and Q, using alphabets B and C respectively, would be denoted $P_B\|_C Q$. In [13], however, *all* processes are defined in such a way that they automatically have an associated alphabet: the alphabet of P is denoted αP. In this approach, which we will adopt here for consistency with [13] and [19], there is no need to introduce explicit alphabets in the syntax for parallel composition. Instead we use the syntax $P\|Q$. The alphabet of $P\|Q$ is simply $\alpha P \cup \alpha Q$. The two methods are closely related: $P\|Q$ is semantically equivalent to the explicitly alphabetized version $P_{\alpha P}\|_{\alpha Q} Q$. Parallel composition is commutative and associative, so that we may use notation such as $\|_{i=1}^{n} P_i$ without ambiguity.

In a parallel composition of processes, each process performs events from its own alphabet, with the constraint that events in the alphabets of two processes require their cooperation. As in the original CSP language of [12], we focus on two-way communication, and hence we restrict attention to parallel compositions which are *triple-disjoint*, in that no event is common to the alphabets of more than two processes.

The observable behaviour of a process is explained entirely in terms of the events (from its alphabet) it may or may not perform when placed in an environment which is trying to interact with it. We are particularly concerned with deadlock: the inability to perform any event (or, equivalently, the ability to refuse all events) in the relevant alphabet.

Traces, refusals, and failures.

If A is a set of events, we write A^* for the set of finite sequences, or *traces* over A. We let a, b, c range over events, s, t, u range over traces, and X, Y, A, B, C range over sets of events. We write $\langle\rangle$ for the empty trace, $\langle a \rangle$ for the trace consisting of the single event a, and st for the concatenation of s and t. We write $s \restriction A$ for the trace obtained from s by deleting all events not in A.

A trace of a process is a finite sequence of events which the process may be able to perform in sequence; a *refusal* of a process is a set of events all of which it may be unable to perform; a *failure* of a process is a pair (s, X) consisting of a trace s and a refusal set X. If (s, X) is a failure of a process P, we interpret this as saying that the process may refuse all of the events in X immediately after having performed the sequence s; thus, if the process is placed in an environment which only wants to perform events from this set next at that stage, deadlock is possible. The traces, refusals and failures of P are all composed exclusively of elements of αP. If a process refuses its entire alphabet it is deadlocked (in any environment).

As in [4], we identify (the semantics of) a process P with its *failure set*, which we denote $\mathcal{F}[\![P]\!]$. This is a subset of $(\alpha P)^* \times \wp(\alpha P)$. We use traces$(P) \subseteq (\alpha P)^*$ for the set of traces of P; initials$(P) \subseteq \alpha P$ is the set of initial events of traces of P, i.e., the set of events c which begin a trace of P; refusals$(P) \subseteq \wp(\alpha P)$ is the set of P's initial refusals. All of these sets can be extracted from the failure set of a process: for instance, s is a trace of P if and only if $(s, \emptyset)$ is a failure of P; i.e., traces$(P) = \{s \mid (s, \emptyset) \in \mathcal{F}[\![P]\!]\}$. When s is a trace of P we write $P \underline{\text{after}}\, s$ for the process whose behaviour describes P's subsequent behaviour after first performing the sequence s. Its defining property is that $\mathcal{F}[\![P \underline{\text{after}}\, s]\!] = \{(t, X) \mid (st, X) \in \mathcal{F}[\![P]\!]\}$.

The failure set of a process is closed under certain natural conditions: in particular, trace sets are prefix-closed, refusal sets are subset-closed, and impossible events can be included in refusal sets. A denotational description of the failures semantic function $\mathcal{F}$ is given in [4,5]. A natural ordering based on nondeterminism makes the failures model into a complete partial order, with respect to which all CSP constructs are continuous; thus, recursive process definitions can be treated in the usual way as denoting least fixed points.

Divergence.

The failures model of processes as described in [4] is adequate for analysing deadlock potential, but less well suited to a proper treatment of *divergence*, which occurs when a process is able to perform an unbounded number of hidden internal actions without communicating to its environment. The *improved failures model* of [5] was developed to allow a more satisfactory treatment of divergence. In this model a process is described by a pair (F, D) containing a failure set F and a divergence set D. Divergence was treated pessimistically, in the sense that we treated any possibility of divergence in a process as catastrophic. In such a pessimistic view, it is useless to try to prove absence of deadlock if there is a possibility of divergence.

In this paper we will again adopt this view of deadlock and divergence: we are only interested in proving deadlock-freedom in divergence-free processes. Therefore, we will generally assume that all processes are divergence-free (i.e., have empty divergence set), so that a process is fully described by its failure set. All examples discussed in this paper are divergence-free, and all of the results apply to divergence-free processes. We will be careful to state the necessary assumptions on divergence-freedom when dealing with operators which may introduce divergence (in particular, with the hiding operator).

Infinite refusal sets.

In [5] we also allowed for the possibility of infinite refusal sets when processes were able to use infinite alphabets (for example, if a process can input an arbitrary natural number). This is important for the present paper, since it allows cleaner statements and easier proofs for several results. It is only a minor technical issue, since even in [5] we take the view that every infinite refusal set of a process is determined by the fact that all of its finite subsets are refusal sets; the main point is that we make infinite refusals explicit in this treatment instead of implicitly characterizing them as finitely generable in this way. By doing this we avoid having to resort to annoyingly verbose phraseology when we want to say (for instance, below, in defining deadlock) that a process may refuse its entire alphabet: in order to find a form of words that works both for finite and infinite alphabets we do not have to paraphrase and say that the process may refuse all finite subsets of its alphabet. Thus, in this paper, we focus on a failures model in which refusal sets can, where necessary, be infinite.

Deadlock.

To match our earlier informal description of deadlock with the failures semantics of processes we now supply formal definitions. As remarked above, these are very simple.

Definition 1. The process P *can deadlock* after the trace s if $(s, \alpha P) \in \mathcal{F}[\![P]\!]$. ∎

Definition 2. The process P is *free of deadlock* (or *deadlock-free*) if

$$\forall s \in (\alpha P)^*.\ (s, \alpha P) \notin \mathcal{F}[\![P]\!].$$ ∎

3. Networks of Communicating Processes.

A network is a parallel combination of processes (and, implicitly, alphabets). We will use an indexed tuple notation $\langle P_i \mid 1 \leq i \leq n \rangle$ for a network of n processes, with each P_i using alphabet αP_i. The processes in a network may be themselves built by parallel composition. By presenting a collection of processes as a network we have a means of imposing topological or hierarchical structure on deadlock analysis: it may prove advantageous to group several processes as a single node in a network for the purposes of proving deadlock-freedom.

The networks we consider will be *static*, in the sense that number of processes and their alphabets are fixed throughout the execution of the system. The problem of treating dynamically changing networks will be addressed briefly in the conclusions section of this paper.

Graphical representations of networks of processes have been used extensively in the literature, for instance by Milne and Milner [15]. First we introduce a graph representing the static communication topology of a network.

Definition 3. The *communication graph* of a network $\langle P_i \mid 1 \leq i \leq n \rangle$ is an undirected graph whose nodes represent the processes P_i, and whose arcs are uniquely determined by the alphabets: there is an arc between P_i and P_j iff $\alpha P_i \cap \alpha P_j \neq \emptyset$ and $i \neq j$. ∎

Thus two processes are linked in a communication graph if and only if there is an event common to their alphabets, representing a communication between them. Since CSP treats communication in a symmetric fashion, we do not assign directions to the arcs. The existence of an arc linking process P_i with P_j in this graph, of course, says nothing about whether or not such a communication will ever take place dynamically as the network operates.

Definition 4. The *vocabulary* of the network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ is the set

$$\bigcup \{\alpha P_i \cap \alpha P_j \mid 1 \leq i < j \leq n\}.$$ ∎

The vocabulary of a network consists of the events common to the alphabets of two processes; we will refer to these as *internal communications*. Since we restrict attention to two-way communications (all networks in this paper are assumed to be triple-disjoint), no event is common to the alphabets of more than two processes, and we do not need a more complicated notion of vocabulary.

We will use the obvious notion of subnetwork: W is a *subnetwork of* V if it arises from V by removing some (or none) of its processes. The communication graph of a subnetwork of V will be the subgraph of the communication graph of V obtained by removing the corresponding nodes and arcs involving those nodes. Note that if W is a subnetwork of V, then the vocabulary of W is a subset of the vocabulary of V.

Many interesting networks have tree-structured communication graphs. For example, trees arise as communication graphs of networks built with the master-slave operator (called subordination in [13])

$$[P \| \mathrm{m}_1{:}Q_1 \| \ldots \| \mathrm{m}_n{:}Q_n],$$

in which for each i the process $\mathrm{m}_i{:}Q_i$ is said to be a slave of P because its alphabet is a subset of αP. This alphabetic constraint implies that in this parallel context each action of a slave process can occur only if P also performs it, i.e. that slave processes can only communicate with their master. In a general tree network, the only internal communications are between processes and their sons. Formally, we define the class of tree networks as follows:

Definition 5. A network V is a *tree* when its communication graph has no cycles, or equivalently when it has one more vertex than arcs. ∎

Note that a tree network is necessarily triple-disjoint, for if $\alpha P_i \cap \alpha P_j \cap \alpha P_k \neq \emptyset$, with i, j, k all distinct, there would be a cycle of edges through these three nodes.

Network Behaviour.

The behaviour of a network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ is that of (the process representing) its parallel composition $\mathrm{PAR}(V)$, defined

$$\mathrm{PAR}(V) \;=\; \|_{i=1}^{n} P_i.$$

The alphabet of this network is defined to be $\alpha V = \bigcup \{\alpha P_i \mid 1 \leq i \leq n\}$. Under our assumption that all the processes involved are divergence-free, the failures of V are given (as in [5,19]) by

$$\mathcal{F}[\![\mathrm{PAR}(V)]\!] \;=\; \{(s, \bigcup_{i=1}^{n} X_i) \mid \forall i (1 \leq i \leq n \Rightarrow (s \upharpoonright \alpha P_i, X_i) \in \mathcal{F}[\![P_i]\!])\}.$$

Intuitively, each process in the network is responsible for performing or refusing events in its own alphabet, with the constraint that an event in the vocabulary of the network requires cooperation of two nodes. Correspondingly the behaviour of a network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ after the trace s will be that of the network V after s defined by:

$$V \text{ after } s \;=\; \langle P_i \text{ after } (s \restriction \alpha P_i) \mid 1 \leq i \leq n \rangle,$$

because at this stage the process at node i has performed the sequence $s \restriction \alpha P_i$, obtained by including only the events in s which belong to the set αP_i. This is shown by the law

$$\mathrm{PAR}(V) \text{ after } s = \mathrm{PAR}(V \text{ after } s) \; .$$

The definition of deadlock-freedom for a process P generalizes in the obvious way to a network V: the network is deadlock-free if and only if the process representing its parallel composition is. For convenience, we repeat here the obvious adaptations of Definitions 1 and 2:

Definition 6. A network V *can deadlock after* s if $(s, \alpha V) \in \mathcal{F}[\![\mathrm{PAR}(V)]\!]$. ∎

Definition 7. A network V is *free of deadlock* if $\mathrm{PAR}(V)$ is free of deadlock, i.e., if

$$\forall s \in (\alpha V)^* .\ (s, \alpha V) \notin \mathcal{F}[\![\mathrm{PAR}(V)]\!].$$ ∎

The vocabulary of a network is an important set from the point of view of deadlock analysis because it is the set of events for whose performance the agreement of two node processes is necessary. Failure to reach agreement is a typical cause of deadlock. At any time when V is deadlocked it is clear that no P_i can be willing to perform any event outside the vocabulary of V: such an event would be under the control of P_i alone, and by definition of parallel composition $\mathrm{PAR}(V)$ would also be willing to perform it.

Since parallel composition is commutative, networks differing only in the order in which we list the nodes have the same behaviour, and of course will also have identical communication graphs (up to isomorphism). But parallel composition is also associative, so that combining several nodes of a network into a single node (whose process is defined to be the obvious parallel composition) does not affect the behaviour. However, a network reorganization like this will produce a *different* communication graph. We wish to allow ourselves the freedom to choose the most suitable network topology for proving deadlock-freedom, so it is important to remember that network reorganizations involving grouping of nodes leave deadlock properties invariant.

We will restrict attention in this paper to networks whose communication graphs are *connected*. This causes no loss of generality when trying to prove absence of deadlock, since one may prove absence of deadlock in a general network by analysing the connected components of its communication graph separately, as explained by the following observation:

Remark. If the connected components of a network V are $V_1, \ldots, V_k$, then V can deadlock after s if and only if for each i the subnetwork V_i can deadlock after $s \restriction \alpha V_i$.

This follows easily from the definition of parallel composition, since the connected components necessarily have disjoint alphabets. Dually, V is deadlock-free if and only if at least one of its connected componets is deadlock-free.

States of a Network.

Execution of a communication by a node will generally change the process at that node; nevertheless, the communication graph of the network remains the same. To account for the dynamic effect of communication events, we now introduce a notion of *state*. A state is simply a cross-section of the network giving the local information about what each process in the system has done so far and is refusing to do on the next step.

Definition 8. A *state* of a network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ is a trace s of V together with an indexed tuple $\langle X_1, \ldots, X_n \rangle$ of refusal sets X_i such that for each i,

$$(s \restriction \alpha P_i, X_i) \in \mathcal{F}[\![P_i]\!].$$

A state is *maximal* if each of its refusal sets is maximal, i.e., if

$$\forall Y.\ (s \restriction \alpha P_i, Y) \in \mathcal{F}[\![P_i]\!] \Rightarrow Y \not\supseteq X_i. \qquad \blacksquare$$

When V is in the state $(s, \langle X_1, \ldots, X_n \rangle)$ each node P_i has so far done the sequence $s \restriction \alpha P_i$ and is currently capable of doing any event from $\alpha P_i - X_i$ on the next step.

The structure of the failures model (specifically, the closure conditions on refusal sets) guarantees that each state may be extended to a maximal state. For the purposes of deadlock analysis it is sufficient to focus attention on maximal states: the more events each individual process refuses, the more likely deadlock becomes. Therefore, throughout this paper, we will assume for convenience that all states have this form. We will denote the maximal failures of a process P, in the sense above, by $\hat{\mathcal{F}}[\![P]\!]$. It will be convenient also to use σ to range over states, and $\underline{X}$ to range over indexed tuples of refusal sets. Thus, a typical state may be written $\sigma = (s, \underline{X})$.

A simple characterization of states in which deadlock occurs is provided by:

Lemma 1. *A network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ can deadlock after s iff there is a state $(s, \langle X_1, \ldots, X_n \rangle)$ for which*

$$\bigcup_{i=1}^{n} \alpha P_i = \bigcup_{i=1}^{n} X_i.$$

Proof. By definition of $\mathcal{F}[\![\mathrm{PAR}(V)]\!]$ and Definition 6. $\blacksquare$

We will refer to such a state as a *deadlock state*. Note that, in accordance with the remarks above, any deadlock state extends to a maximal deadlock state. Henceforth, when we refer to a deadlock state, maximality will be assumed implicitly.

Requests and Snapshot Graphs.

Next we introduce the notion of a "request" in a state. Our choice of terminology is intended to be suggestive. A pair of indices $\langle i, j \rangle$ is a request when P_i is trying to communicate with P_j, i.e. when there is an event common to the alphabets of P_i and P_j that is not in P_i's refusal set; $\langle i, j \rangle$ is a strong request if P_i can *only* communicate with P_j, because all events available to P_i on the next step belong also to the alphabet of P_j (and P_i is not itself deadlocked, so that there is at least one event possible for its next step). A request $\langle i, j \rangle$ is ungranted if the target P_j of the request is unwilling to respond to the source P_i, i.e., P_j is currently refusing all of the events relevant to P_i in this state. The formal definition is:

Definition 9. Let $\sigma = (s, \underline{X})$ be a state of the network $V = \langle P_i \mid 1 \leq i \leq n \rangle$. A pair of indices $\langle i, j \rangle$ (with $i \neq j$) is:

- a *request* if $(\alpha P_i - X_i) \cap \alpha P_j \neq \emptyset$;
- a *strong request* if $\emptyset \neq (\alpha P_i - X_i) \subseteq \alpha P_j$;
- *ungranted* if in addition $\alpha P_i \cap \alpha P_j \subseteq X_i \cup X_j$. ∎

An alternative and equivalent formulation of the condition for ungrantedness is that

$$(\alpha P_i - X_i) \cap (\alpha P_j - X_j) = \emptyset.$$

Strictly speaking, the notions of request, strong request, and ungrantedness refer to a specific state. In practice, the state will be clear from the context and we will often omit explicit reference to it. Clearly, by definition, every strong request is also a request.

Ungranted requests can be regarded as the basic building blocks of deadlock. Sometimes we need only be interested in ungranted requests when neither process is able to communicate outside some set A; an especially common case is when A is the vocabulary of the network, since events outside of the vocabulary do not require cooperation between processes. This motivates the following definition.

Definition 10. The pair $\langle i, j \rangle$ is a request (or strong request) *with respect to* A if, in addition to the above requirements, we also have:

$$(\alpha P_i - X_i) \cup (\alpha P_j - X_j) \subseteq A.$$ ∎

Using the notation of [19] we will write

$$P_i \xrightarrow{\sigma} P_j \quad \text{or} \quad P_i \stackrel{\sigma}{\Longrightarrow} P_j$$

when $\langle i, j \rangle$ is a request or strong request of σ. Similarly we will write

$$P_i \xrightarrow{\sigma} \bullet P_j \quad \text{or} \quad P_i \stackrel{\sigma}{\Longrightarrow} \bullet P_j$$

when $\langle i, j \rangle$ is a ungranted request or strong request of σ, and

$$P_i \xrightarrow{\sigma, A} \bullet P_j \quad \text{or} \quad P_i \stackrel{\sigma, A}{\Longrightarrow} \bullet P_j$$

when $\langle i, j \rangle$ is an ungranted request with respect to A.

As a trivial consequence of the definition, any request $\langle i, j \rangle$ is a request with respect to $\alpha P_i \cup \alpha P_j$. Thus, for example,

$$P_i \xrightarrow{\sigma, A} \bullet P_j \Leftrightarrow P_i \xrightarrow{\sigma} \bullet P_j \quad \text{when} \quad \alpha P_i \cup \alpha P_j \subseteq A.$$

It is also obvious that only events belonging to the alphabets of the two processes matter: $\langle i, j \rangle$ is a request with respect to A iff it is a request with respect to $A \cap (\alpha P_i \cup \alpha P_j)$. An ungranted request with respect to A is still ungranted in any superset of A; that is,

$$\text{if } A \subseteq A', \text{ then } P_i \xrightarrow{\sigma, A} \bullet P_j \Rightarrow P_j \xrightarrow{\sigma, A'} \bullet P_j.$$

Of course, similar observations are true of strong requests.

In a network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ the process P_i is said to be *blocked* in the state σ when it is the source of a request, it can only perform events internal to the network, but all of its requests are ungranted: i.e., when

- $P_i \xrightarrow{\sigma} P_j$ for some j,
- $P_i \xrightarrow{\sigma, A} \bullet P_k$ whenever $P_i \xrightarrow{\sigma} P_k$,

where A is the vocabulary of the network.

There is an obvious relationship between blocking and the existence of deadlock: a state σ of a network V is a deadlock state if and only if every process of V is blocked in σ. This follows easily from the definitions.

To aid in the visualisation of deadlock, we next introduce a graphical representation for the collection of requests being made in a state: we call this a "snapshot" graph. This gives an instantaneous picture of the dynamically evolving behaviour of a network, and summarizes the information we need to know in order to determine what the next actions (if any) of the network are in a particular state.

Definition 11. The *snapshot graph* of a network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ in a state $(s, \langle X_1, \ldots, X_n \rangle)$ is the directed graph on the nodes of V in which there is a directed arc from node i to node j iff $\langle i, j \rangle$ is a request in this state, i.e., if $(\alpha P_i - X_i) \cap \alpha P_j \neq \emptyset$. ∎

Definition 12. Let σ be a state of the network $V = \langle P_i \mid 1 \leq i \leq n \rangle$. A sequence of indices $\langle i_0, \ldots, i_{r-1} \rangle$ with $r \geq 3$ is termed a *cycle of requests* in σ if, for each j, $\langle i_j, i_{j+1} \rangle$ is a request in σ (where addition is modulo r). ∎

Cycles of requests correspond precisely to cycles in snapshot graphs. A cycle of requests is *proper* if all of its indices are distinct. It will be termed a cycle of ungranted, or of strong etc., requests in case each of the requests has the appropriate property. The length of this cycle is r, which for a proper cycle is the number of distinct nodes involved. We restrict attention here to cycles of length at least 3; the case of a cycle of length 2 is sufficiently different to merit special treatment later in the paper. In fact, a pair of ungranted requests $\langle i, j \rangle$ and $\langle j, i \rangle$ will be called a *conflict*, and we will devote considerable attention to notions of conflict later.

We will develop techniques for proving deadlock-freedom that rely on establishing a connection between the presence of deadlock and the existence of cycles of ungranted requests. The analogy is not exact, however: there are deadlock-free networks with states in which there are cycles of ungranted requests; there are even deadlocking networks in which no cycles exist (for example, a trivial network with a unique, deadlocked, node). Nevertheless, we will see that for certain general classes of network (ruling out trivial cases like this) deadlock can only be caused by cycles of ungranted requests; this will enable us to focus attention on regions of a network in which such cycles might exist.

To end this section we provide some examples of network definitions and use them to demonstrate and elaborate upon the terminology we have just introduced. These examples also demonstrate the prominent role of ungranted requests in deadlock.

Example 1: A deadlocked chain.

A chain of processes in which each one communicates solely with its immediate neighbours is represented as a particularly simple form of tree network: a single branch. Here is an interesting family of chains, parameterized by the number of processes.

Define a chain of $n+1$ processes for any $n > 1$ as follows. The processes will be $P_0, \ldots, P_n$, with alphabets αP_i given by

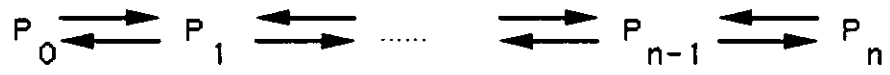
$$\begin{aligned} \alpha P_0 &= \{\text{1.a}, \text{1.b}\}, \\ \alpha P_i &= \{\text{i.a}, \text{i.b}, \text{i+1.a}, \text{i+1.b}\}, \qquad 1 \le i < n, \\ \alpha P_n &= \{\text{n.a}, \text{n.b}\}. \end{aligned}$$

Events have suggestive names comprising a "channel" number and a "message". For simplicity, the only possible messages are a and b, and the channels are numbered 1 to n. We specify the node processes informally as follows. The left-hand end process P_0 can send message b along channel 1 to process P_1 in response to receiving a from it. The right-hand end process P_n can send message a to P_{n-1} along channel n in response to message b. Each of the intermediate processes $P_1, \ldots, P_{n-1}$ can transmit a from its right to its left, and b from its left to its right. The (recursive) process definitions are ($i = 1, \ldots, n-1$):

$$\begin{aligned} P_0 &= (\text{1.a} \rightarrow \text{1.b} \rightarrow P_0) \\ P_n &= (\text{n.b} \rightarrow \text{n.a} \rightarrow P_n) \\ P_i &= (\text{i+1.a} \rightarrow P_i^a) \,\Box\, (\text{i.b} \rightarrow P_i^b), \\ P_i^a &= (\text{i.a} \rightarrow P_i) \,\Box\, (\text{i.b} \rightarrow P_i^{ab}), \\ P_i^b &= (\text{i+1.a} \rightarrow P_i^{ab}) \,\Box\, (\text{i+1.b} \rightarrow P_i), \\ P_i^{ab} &= (\text{i.a} \rightarrow P_i^b) \,\Box\, (\text{i+1.b} \rightarrow P_i^a). \end{aligned}$$

The superscripts on the auxiliary processes (e.g. in P_i^a) indicate the messages which the process is ready to transmit on the next step.

The vocabulary of this network is its entire alphabet, so that every event requires the participation of two processes. However, the process definitions do not allow any single process to *initiate* either an a or a b signal, and no pair of processes can agree initially on a communication, so the chain deadlocks immediately. That is, there is a deadlock state with the empty trace. In this state there are requests $\langle i-1, i\rangle$ and $\langle i, i-1\rangle$ for $1 \le i \le n$. All of these requests are ungranted, and only $\langle 0, 1\rangle$ and $\langle n, n-1\rangle$ are strong. This state yields the following snapshot graph:



Snapshot of Deadlocked Chain

DIAGRAM 1

The astute reader may notice that despite the fact that the entire network may deadlock, every non-empty subnetwork is deadlock-free! Informally, this is because in every non-empty proper subnetwork some event is no longer in the vocabulary and can therefore be initiated by a single process. We will prove that every non-empty subnetwork is deadlock-free later in the paper, as a consequence of a more general result.

Example 2: Dining Philosophers.

In this example, attributed to Dijkstra and Scholten by Hoare [12], there are five "philosopher" processes, five "fork" processes, and a "butler" process. The deadlock properties of this system are well known. The process definitions are:

$$\begin{aligned} \mathrm{PHIL}_i &= (\mathrm{i.enters} \rightarrow \mathrm{i.picks.i} \rightarrow \mathrm{i.picks.i{+}1} \rightarrow \\ &\qquad \mathrm{i.eats} \rightarrow \mathrm{i.puts.i} \rightarrow \mathrm{i.puts.i{+}1} \rightarrow \mathrm{i.leaves} \rightarrow \mathrm{PHIL}_i), \\ \mathrm{FORK}_i &= (\mathrm{i.picks.i} \rightarrow \mathrm{i.puts.i} \rightarrow \mathrm{FORK}_i) \,\Box\, (\mathrm{i{-}1.picks.i} \rightarrow \mathrm{i{-}1.puts.i} \rightarrow \mathrm{FORK}_i), \end{aligned}$$

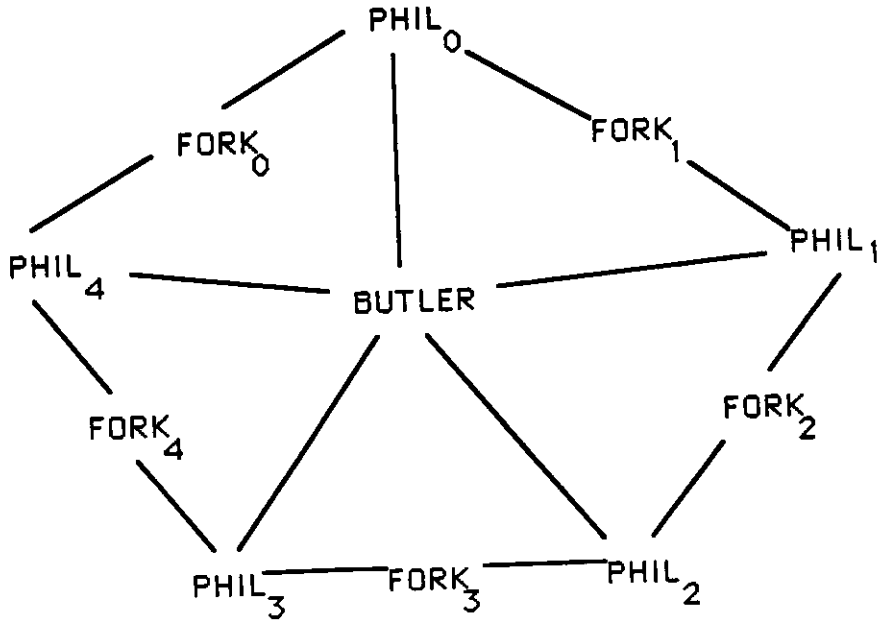
for $i = 0, \ldots, 4$, and

$$\begin{aligned} \mathrm{BUTLER} &= \mathrm{ADMIT} \,|||\, \mathrm{ADMIT} \,|||\, \mathrm{ADMIT} \,|||\, \mathrm{ADMIT}, \\ \text{where} \quad \mathrm{ADMIT} &= \Box_{i=0}^{4} (\mathrm{i.enters} \rightarrow \mathrm{i.leaves} \rightarrow \mathrm{ADMIT}). \end{aligned}$$

Addition and subtraction of indices is modulo 5. Each philosopher wants to enter, pick up the fork on his right, pick up the fork on his left, eat, then put down the two forks, then leave, and resume his cyclic pattern of behaviour. Each fork will initially allow itself to be picked up by either of its neighbouring philosophers, after which it must wait to be put down again before resuming its initial configuration. The butler is an interleaving of four copies of a process which repeatedly allows the entering and subsequent leaving of a philosopher. The alphabets of these processes are:

$$\begin{aligned} \alpha\mathrm{PHIL}_i &= \{\mathrm{i.picks.i}, \mathrm{i.puts.i}, \mathrm{i.eats}, \mathrm{i.enters}, \mathrm{i.leaves}, \mathrm{i.picks.i{+}1}, \mathrm{i.puts.i{+}1}\}, \\ \alpha\mathrm{FORK}_i &= \{\mathrm{i.picks.i}, \mathrm{i{-}1.picks.i}, \mathrm{i.puts.i}, \mathrm{i{-}1.puts.i}\}, \qquad (i = 0, \ldots, 4), \\ \alpha\mathrm{BUTLER} &= \{\mathrm{i.enters}, \mathrm{i.leaves} \mid 0 \leq i \leq 4\}, \end{aligned}$$

Diagram 2 shows the communication graph for a network with nodes for each of the philosophers, forks, and the butler.



Communication Graph of Dining Philosophers

DIAGRAM 2

It is possible in this system for four philosophers to enter and each to pick up one fork, for instance as described by the trace

$$\langle 1.\text{enters}, 2.\text{enters}, 3.\text{enters}, 4.\text{enters}, 1.\text{picks.1}, 2.\text{picks.2}, 3.\text{picks.3}, 4.\text{picks.4}\rangle.$$

At this point, for $i = 1 \ldots 4$ the future behaviour of the i^{th} philosopher is described by:

$$\text{PHIL}_i \text{ after } (\text{i.enters}, \text{i.picks.i}) = (\text{i.picks.i+1} \rightarrow \cdots).$$

In other words, each of these four philosophers now refuses $\alpha\text{PHIL}_i - \{\text{i.picks.i+1}\}$. The other philosopher (PHIL_0) is still trying to enter, so he refuses $\alpha\text{PHIL}_0 - \{0.\text{enters}\}$. The future behaviour of the i^{th} fork ($i = 1 \ldots 4$) is that of

$$\text{FORK}_i \text{ after } (\text{i.picks.i}) = (\text{i.puts.i} \rightarrow \text{FORK}_i),$$

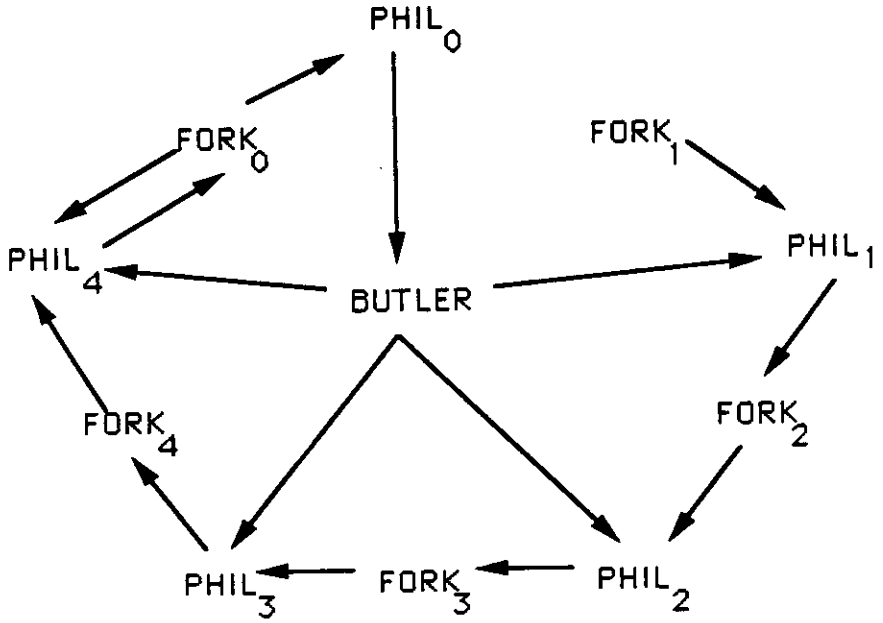
so that each is refusing $\alpha\text{FORK}_i - \{\text{i.puts.i}\}$. The fork numbered 0 is still waiting to be picked up, refusing $\alpha\text{FORK}_0 - \{0.\text{picks.0}, 4.\text{picks.0}\}$.

The butler is described at this point by

$$\text{BUTLER} \text{ after } \langle 1.\text{enters}, 2.\text{enters}, 3.\text{enters}, 4.\text{enters}\rangle = |||_{i=1}^{4} (\text{i.leaves} \rightarrow \text{ADMIT}).$$

Thus, the butler is refusing $\alpha\text{BUTLER} - \{\text{i.leaves} \mid 1 \leq i \leq 4\}$.

We have now described all of the information for a particular state of the system: a trace, and corresponding (maximal) refusals for each process. This is *not* a deadlock state, because the union of these refusal sets does not contain the event 4.picks.0 (in which fork 0 is picked up by philosopher 4); the network is able to perform this event when in this state. Diagram 3 shows the snapshot graph of this network in this state. The requests are all ungranted except for the requests involving the pair PHIL_4 and FORK_0.

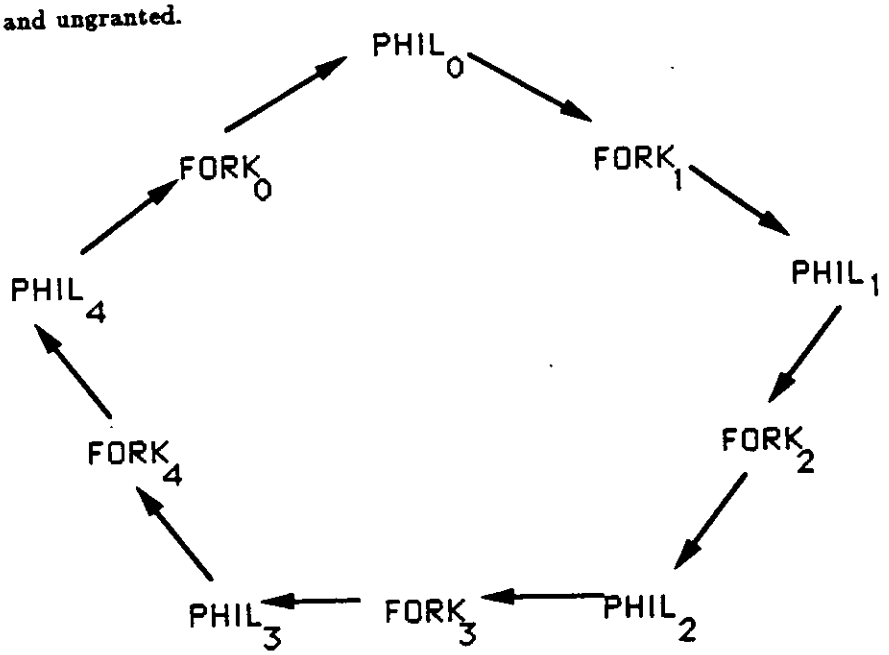


Snapshot Graph of Dining Philosophers

DIAGRAM 3

Example 3: Deadlocked Philosophers.

If the dining philosophers are allowed to operate without the guidance of the butler there is a potential deadlock (when all five philosophers enter and pick up a single fork each). This is summarized in the snapshot graph of Diagram 4. All of these requests are strong and ungranted.



Snapshot of Deadlocked Philosophers

DIAGRAM 4

Example 4: Variants of the Dining Philosophers.

If we regroup the nodes of the dining philosophers network (Example 2) by combining the philosophers into a single node and combining the forks into a single node, we get the network

$$\langle \text{BUTLER}, \|_{i=0}^{4}\text{PHIL}_i,\ \|_{i=0}^{4}\text{FORK}_i \rangle,$$

with identical behaviour to the original system but the following communication graph:

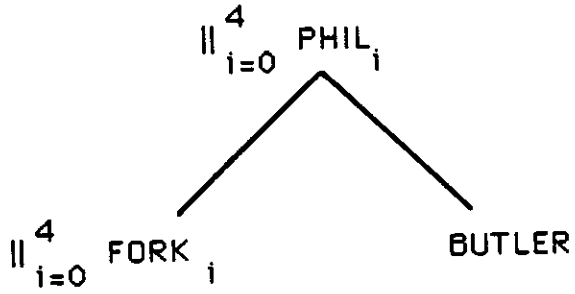


DIAGRAM 5

Each node in this network is deadlock-free. However, if we combine all philosophers and forks into a single node, we get a network with just two nodes, one of which (as discussed above) can deadlock. Nevertheless, again we have the same overall behaviour, so that the system is still deadlock-free.

4. Deadlock Properties of Networks.

We have already defined deadlock-freedom as a global property of a network, involving the behaviour of the process PAR(V). We have stated that deadlock-freedom is invariant under network reorganisations like permutation of nodes and grouping nodes together. We want to be able to take advantage of well chosen network presentations for CSP processes: to use network topology and graph structure as an aid in structuring proofs of deadlock-freedom. For non-trivial networks there are several interesting variations on the theme of deadlock, which take into account the network topology.

Firstly, we will say that a network has a property *hereditarily* if it and all of its non-empty subnetworks have it. A property (of networks) is hereditary if and only if whenever it holds of an entire network it also holds of all non-empty subnetworks. Deadlock-freedom is *not* an hereditary property; equivalently, a network can be free of deadlock without having that property hereditarily. This has already been shown by the Dining Philosophers network (Example 2): the subnetwork obtained by removing the butler (Example 3) fails to be deadlock-free. Hence, it is worthwhile making the following definition.

Definition 13. A network V is *hereditarily deadlock-free* if (it and) each of its non-empty subnetworks is deadlock-free. ∎

It may even be possible (with care, although perhaps misguidedly) to design a deadlock-free network from nodes which may deadlock: one needs in such a case to ensure that the synchronisation requirements forced by the rest of the network prevent each node from reaching deadlock. We believe that this situation is rather irrelevant from the point of view of developing general methods for establishing deadlock-freedom. Since we are specifically interested in developing techniques based on local analysis it is hard to imagine a general method (as opposed to *ad hoc* techniques) in which deadlock-freedom of individual nodes is not crucial. For example, it is easier to reason about dining philosophers when presented with the original network structure, in which all nodes are deadlock-free, rather than using the two-node variant version. Hence, we will concentrate on networks built from individual nodes which are themselves deadlock-free: these we call *busy* networks.

Definition 14. A network V is *busy* if all of its node processes are deadlock-free. ∎

Another advantage of this type of restriction is that it fits well with our desire to develop methods which support hierarchical analysis of a network: if a network is built from nodes which are themselves networks, we will be able to use our techniques for proving deadlock-freedom first for the individual nodes, and then to analyse the entire network we need no longer take into account the network structure of its nodes, since all we require to know about them is that they are deadlock-free.

Note that the properties of triple-disjointness and busyness are obviously hereditary. The property of being a tree is almost hereditary, in the sense that whenever W is a non-empty subnetwork of a tree network V, each of its connected components is again a tree. Given this fact, we will abuse notation slightly and say that treehood is hereditary.

We should remark on the relationships between these various notions of deadlock-freedom. Trivially, hereditary freedom from deadlock implies freedom from deadlock, and also implies busyness. Deadlock-freedom neither implies nor is implied by busyness.

In the rest of the paper we will develop some general techniques for proving deadlock-freedom that use busyness as a hypothesis. The main aim is to develop deadlock-freedom proofs which require only local analysis: busyness (involving single nodes) and pairwise analysis.

5. Proving Deadlock Properties of Networks.

We begin with some very elementary results. The first gives us a base case in beginning hierarchical proofs of deadlock-freedom: when all the parallelism in a network is at the outermost level, so that none of the node processes involve parallel composition, it is very easy to prove busyness. The second pair of results allows us freedom to disregard uses of hiding, or to introduce carefully selected hiding operations to simplify deadlock analysis.

Busy Networks.

To show that a network is busy in general requires a proof of deadlock-freedom for all individual processes. The following simple rule is useful as a basis for establishing deadlock-freedom for CSP processes built without parallel composition; it can be used to prove busyness in a network whose node processes conform to a simple subset of CSP (in particular, no node is itself a parallel composition, and no node can ever terminate successfully). Node processes are allowed to be built by prefixing, by nondeterministic choice (internal and external forms, $\sqcap$ and $\Box$ respectively), by renaming (with an alphabet transformation f), by recursion ($\mu p.P$), and by sequential composition ($P;Q$). The successfully terminating process SKIP may be used in building up node processes, but only in limited contexts to prevent termination of the node process (every occurrence of SKIP must be directly or indirectly followed by a sequential composition). The reason for this constraint should be obvious: a terminated process (like a deadlocked process) cannot perform any event. Note that it is possible to define divergent processes using these constructs (e.g., by unguarded uses of recursion such as $\mu p.p$ or by slightly less obvious cases such as $\mu p.\mathrm{SKIP};p$). Consequently, as remarked earlier, we need to check for divergence-freedom before attempting deadlock analysis.

It is easy to prove (by induction on syntactic structure) that a divergence-free process built with these constructs alone and obeying this constraint on SKIP can never refuse its entire alphabet, and is therefore deadlock-free. Hence the following rule (called D1 in [19]):

Lemma 2. *Suppose the definition of the process P uses only the following syntax:*

$$P ::= SKIP \mid a \rightarrow P \mid P;Q \mid P \,\Box\, Q \mid P \sqcap Q \mid f(P) \mid p \mid \mu p.P$$

(where p denotes a process variable), and P contains no free process variables, is divergence-free, and every occurrence of SKIP in P is directly or indirectly followed by a ";" . Then P is deadlock-free. ∎

Thus, trivially, any network in which the component processes satisfy Lemma 2 will be busy.

Hiding and deadlock analysis.

In many applications (as for instance in occam [14]), uses of parallel composition are accompanied by the hiding of internal communications. These are often regarded as uninteresting to the external observer and in practice outside his control. However, when considering the possibility of deadlock, it is usually vital to keep a full record of the internal events of a network; therefore the networks we consider do not as a rule have internal events hidden. Indeed, the operator PAR defined above does not involve any hiding. Nevertheless, since node processes may be arbitrary CSP processes and may thus involve uses of hiding, we do need to be able to deal with hiding in deadlock analysis. Since hiding may introduce divergent behaviour (if arbitrarily long sequences of the hidden action were possible) we must be careful to ensure divergence-freedom when we apply the hiding operator to

processes. Fortunately, in considering hiding and deadlock analysis the following two laws (called D2 and D3 in [19]) are particularly helpful:

Lemma 3. *If $P\backslash C$ is divergence-free, then it is deadlock-free if and only if P is.* ∎

Lemma 4. *If $C \cap \alpha Q = \emptyset$, then $(P\backslash C \| Q) = (P \| Q)\backslash C$.* ∎

Given any process definition built by parallel composition and hiding, Lemma 4 permits one to move all the hiding to the outermost level, provided any relevant internal communications are renamed to make the condition $C \cap \alpha Q = \emptyset$ true. One thus obtains a behaviourally equivalent process involving an outermost application of hiding. Thus this law says that as far as behavioural analysis is concerned it does not matter whether hiding is all done at the outermost syntactic level or is done in various stages as a network is put together. Once Lemma 4 has been used in this way and the system has been proved free of divergence (not necessarily in that order), Lemma 3 simply observes that, the presence of hiding does not affect the presence of deadlock so we may, for the purpose of proving absence of deadlock, remove the hiding operator altogether.

The above argument permits us, with care, to ignore applications of hiding. It is also possible to *introduce* hiding carefully, and this idea may be very useful in reducing the number of events in a network's alphabet and hence reducing the complexity of its deadlock analysis. If $C \subseteq \alpha P - \alpha Q$ is a set of events such that $P\backslash C$ is divergence-free, then Lemma 3 and Lemma 4 tell us that $P \| Q$ is deadlock-free if and only if $(P\backslash C) \| Q$ is. This concealment of "irrelevant" communication in P can substantially reduce simplify deadlock analysis: a suitable choice of C may greatly diminish the number of states one needs to consider. Examples later in the paper will illustrate this type of reasoning.

6. Deadlock Analysis in Unidirectional Networks.

Many interesting networks have the especially simple property that at all times each process is prepared to communicate with at most one other process; the choice of communication partner may vary during execution of the network. For instance, Dijkstra [9] discusses networks in which each process attempts to communicate with its neighbours in cyclic order. The general property, which we term unidirectionality, is formalized as follows. It is clearly an hereditary property.

In this section of the paper we will develop some results based on pairwise analysis for establishing deadlock-freedom properties of unidirectional systems. First, the formal definition.

Definition 15. A network $V = \langle P_i \mid 1 \leq i \leq n \rangle$ is *unidirectional* if for each trace s of V and each i there is at most one $j \neq i$ such that

$$\text{initials}(P_i \underline{\text{after}}\, s \backslash \alpha P_i) \cap \alpha P_j \neq \emptyset.$$ ∎

Unidirectionality of a network obviously implies that in a deadlock state any request is also a strong request. It also clearly implies that in any state the snapshot graph can have at most one arc leading from any node. Of course, processes in a unidirectional network may still be able to perform events outside of the network's vocabulary.

A connection between cycles of requests and deadlock is made by the following result. It gives a simple characterization of the snapshot graph of a unidirectional system in a deadlock state: if the system satisfies the conditions of the theorem then deadlock corresponds to a cycle in the snapshot graph involving at least three distinct nodes, each request being ungranted.

Theorem 1. *Let $V = \langle P_i \mid 1 \le i \le n \rangle$ be a busy unidirectional network of processes. If each pair $[P_i \| P_j]$ is free of deadlock, then any deadlock state of V contains a proper cycle of ungranted strong requests.*

Proof.

Let $P = \mathrm{PAR}(V)$ and let $(s, \langle Y_1, \ldots, Y_n \rangle)$ be a deadlock state of P. Then by definition

$$\forall i.\ (s \upharpoonright \alpha P_i, Y_i) \in \bar{\mathcal{F}}[\![P_i]\!], \tag{a}$$

and by Lemma 1,

$$\bigcup_{i=1}^{n} \alpha P_i \ = \ \bigcup_{i=1}^{n} Y_i. \tag{b}$$

It follows from this and the fact that the network is triple-disjoint that, whenever $i \neq j$,

$$\alpha P_i \cap \alpha P_j \subseteq Y_i \cup Y_j, \tag{c}$$

By assumption, the Y_i are maximal refusal sets in (a). For each i let Q_i be P $\underline{\text{after}}$ $s \upharpoonright \alpha P_i$, so that $P \ \underline{\text{after}}\ s \ = \ \|_{i=1}^{n} Q_i$. We argue as follows, letting i be an arbitrary index.

- Since P_i was assumed to be deadlock-free, we have $Y_i \not\supseteq \alpha P_i$. From (b) we see that for each i,

$$\alpha P_i - (\bigcup_{j \neq i} \alpha P_j) \ \subseteq \ Y_i, \tag{1}$$

so that in this state of the network each process is refusing all events unique to its own alphabet. Hence,

$$\emptyset \neq \alpha P_i - Y_i \subseteq \bigcup_{j \neq i} \alpha P_j. \tag{2}$$

- By maximality of Y_i we know that Y_i contains all of the *impossible* events, those in the set $\alpha P_i - \mathrm{initials}(Q_i)$:

$$\alpha P_i - \mathrm{initials}(Q_i) \subseteq Y_i.$$

Hence,

$$\alpha P_i - Y_i \subseteq \mathrm{initials}(Q_i).$$

But there is at most one $j \neq i$ with

$$\mathrm{initials}(Q_i) \cap \alpha P_j \neq \emptyset,$$

since the system is unidirectional. Hence there is at most one $j \neq i$ for which

$$(\alpha P_i - Y_i) \cap \alpha P_j \neq \emptyset. \tag{3}$$

Putting these facts together, we see that there is a unique j (depending on i) such that $i \neq j$ and

$$\emptyset \neq \alpha P_i - Y_i \subseteq \alpha P_j.$$

In the above analysis, i was arbitrary, and clearly the unique j satisfying (3) depends on i. Now consider this j as a function of i, mapping indices to indices. Note that $j(i) \neq i$, and it also happens that $j(j(i)) \neq i$, because if this were to happen we would have a pair of indices $i, j = j(i)$ with

$$\alpha P_i - Y_i \subseteq \alpha P_j, \qquad \alpha P_j - Y_j \subseteq \alpha P_i.$$

But by (c) we would then have

$$\alpha P_i \cap \alpha P_j \subseteq Y_i \cup Y_j,$$

which would in turn imply that

$$\alpha P_i - Y_i \subseteq Y_j, \qquad \alpha P_j - Y_j \subseteq Y_i.$$

Hence, we would get

$$Y_i \cup Y_j = \alpha P_i \cup \alpha P_j,$$

contradicting the assumption that the pair $[P_i \| P_j]$ was deadlock-free.

The sequence

$$1, j(1), j^2(1), \ldots$$

must contain a first repetition, say $j^m(1) = j^{m+r}(1)$, since there are only finitely many indices. Define $i_k = j^{m+k}(1)$, for $k = 0 \ldots r-1$. Then $\langle i_0, \ldots, i_{r-1} \rangle$ is a proper cycle of strong ungranted requests. ∎

An intuitive interpretation of this theorem is that global deadlock (i.e., deadlock of the entire system) can only be caused in a unidirectional system by local deadlock (involving at most two processes) or else by a cycle of at least three distinct nodes each demanding to communicate with its successor and refusing to communicate with its predecessor.

Theorem 1 thus gives us a way to focus on specific parts of a unidirectional network (cycles in its communication graph) if we can first establish busyness and pairwise deadlock-freedom. If a network has only a small number of cycles, this type of approach may be advantageous. An important special case is when we have a unidirectional tree network.

Corollary (i). *If a tree network is busy, unidirectional, and pairwise deadlock-free then it is hereditarily free of deadlock.*

Proof. A tree has no proper cycles, and all of the hypotheses of the theorem are hereditary properties. ∎

Corollary (ii). *In a busy unidirectional tree network, pairwise deadlock-freedom implies absence of global deadlock.* ∎

Thus we have a simple method requiring only pairwise deadlock analysis for establishing deadlock properties in unidirectional tree networks.

Another special case where the number of cycles is very small is in a unidirectional *ring* of processes: there are only two possible cycles to consider: clockwise and anticlockwise. To satisfy the preconditions of Theorem 1 we still need to prove pairwise freedom from deadlock. This may also often be possible by a simple case analysis based on the traces and refusals of the two processes in question, and the amount of work involved in the analysis can often be reduced substantially by making further use of Lemma 3 and Lemma 4 above. Here is an example to illustrate this type of reasoning.

Example 5: A Token Ring.

This example is based on [10]. We consider a ring of n processes ($n \geq 3$) each of which wants to keep entering a "critical section"; to maintain mutual exclusion, a process is only allowed to enter its critical section when it has obtained a "privilege" token, which is passed around the ring. When a process wants to begin its critical section it first requests the token from its neighbour; when it is granted the privilege (i.e., when the token reaches it), the process performs its critical section (represented here by a single event) and then releases

the token. Using mutual recursion, and with mnemonic event names, we may define the individual processes P_i $(i < n)$ by

$$\begin{aligned} P_i &= (\text{i.get} \rightarrow \text{i+1.find} \rightarrow \text{i.priv} \rightarrow \text{i.crit} \rightarrow \text{i.rel} \rightarrow Q_i) \\ &\quad \Box\ (\text{i.find} \rightarrow \text{i+1.find} \rightarrow \text{i.priv} \rightarrow \text{i-1.priv} \rightarrow P_i), \\ Q_i &= (\text{i.get} \rightarrow \text{i.crit} \rightarrow \text{i.rel} \rightarrow Q_i) \,\Box (\text{i.find} \rightarrow \text{i-1.priv} \rightarrow P_i). \end{aligned}$$

All arithmetic here is modulo n. The neighbours of process i are $i-1$ and $i+1$. P_i represents a node without the token and Q_i represents a node with the token. Thus, if P_i wants to get the token it must put in a request first to its successor, and wait for that process to find the token and pass it back; if P_i is asked to find the token it passes the request on to its neighbour, and will later relay the token towards the requester. A Q_i process with the token may either allow the critical action or pass the token on to its predecessor.

For each i, let $\alpha P_i = \alpha Q_i$ be the obvious alphabet consisting of all events appearing in the syntactic description of processes above. For the network $V = \langle Q_0, P_1, \ldots, P_{n-1} \rangle$, in which initially the process with index 0 has the token, and these alphabets are used, we would like to prove freedom from deadlock. The communication graph of V is the obvious cycle.

Each of the node processes is obviously deadlock-free, by Lemma 2, since they are built by prefixing, conditional choice, and recursion. It is easy to see from the process definitions that the system is (triple-disjoint and) unidirectional. We wish to use Theorem 1. First we prove pairwise freedom from deadlock. Since non-adjacent pairs of processes are trivially deadlock-free (their alphabets are disjoint and each individual is deadlock-free), it is only necessary to show that each of the adjacent pairs

$$\begin{aligned} &Q_0 \| P_1, \\ &P_i \| P_{i+1} \qquad (0 < i < n), \\ &P_{n-1} \| Q_0 \end{aligned}$$

is deadlock-free. These analyses are simplified by judicious use of hiding, as follows.

For each i, let $L_i = \{\text{i.get}, \text{i.rel}, \text{i.find}, \text{i-1.priv}\}$ and $R_i = \{\text{i.crit}, \text{i.rel}, \text{i+1.find}, \text{i.priv}\}$. Clearly, $L_i \subseteq \alpha P_i - \alpha P_{i+1}$ and $R_i \subseteq \alpha P_{i+1} - \alpha P_i$. We can hide the events from L_i in P_i (or in Q_i) without introducing divergence, because at all stages P_i cannot perform arbitrarily long traces consisting only of events from this set. The same is true of R_i and P_i or Q_i. Let $P_i^L = P_i \backslash L_i$ and $Q_i^L = Q_i \backslash L_i$, with similar notation P_i^R and Q_i^R for $P_i \backslash R_i$ and $Q_i \backslash R_i$. By Lemmas 3 and 4, the original network is pairwise free of deadlock if and only if the pairs

$$Q_0^L \| P_1^R, \tag{1}$$

$$P_i^L \| P_{i+1}^R \qquad (0 < i < n), \tag{2}$$

$$P_{n-1}^L \| Q_0^R. \tag{3}$$

are deadlock-free. We have, by definition, and using standard properties of the hiding operation [5],

$$\begin{aligned} P_i^L &= \text{i+1.find} \rightarrow \text{i.priv} \rightarrow (Q_i^L \sqcap P_i^L), \\ Q_i^L &= (\text{i.crit} \rightarrow Q_i^L) \sqcap P_i^L, \\ Q_i^R = P_i^R &= (\text{i.find} \rightarrow \text{i-1.priv} \rightarrow P_i^R) \,\Box (\text{i.get} \rightarrow Q_i^R). \end{aligned}$$

Since $\sqcap$ is associative and idempotent, it follows easily that P_i^L satisfies the simpler equation

$$P_i^L = \text{i+1.find} \rightarrow \text{i.priv} \rightarrow Q_i^L.$$

Since (3) is in fact case $i = n - 1$ of (2), we need only consider (1) and (2). These pairwise deadlock analyses may be done by a fairly straightforward analysis based on the process definitions. To illustrate the type of reasoning necessary here, consider a typical pair in case (2), $P_i^L \| P_{i+1}^R$ for an i in the range $1 \ldots n-1$. The alphabets of these two processes are

$$\begin{aligned} \alpha P_i^L &= \{\text{i+1.find, i.priv, i.crit}\}, \\ \alpha P_{i+1}^R &= \{\text{i+1.find, i.priv, i+1.get}\}. \end{aligned}$$

Hence, deadlock is only possible if at some point P_i^L refuses i.crit, P_{i+1}^R refuses i+1.get, and one of the processes refuses i+1.find, and one of the processes refuses i.priv. Let $\#a$ denote the number of occurrences of the event a in the current trace of this pair of processes. From the process definitions it is clear that in all stages where P_i^L refuses {i.crit, i.priv} we have #i+1.find = #i.priv, whereas whenever P_{i+1}^R is refusing {i+1.get, i+1.find} we have #i.priv = #i+1.find − 1. This combination of refusals is therefore impossible. The only remaining possibility for deadlock would thus be if P_i^L refuses {i.crit, i+1.find} and P_{i+1}^R refuses {i+1.get, i.priv}. A similar counting argument disposes of this case, and we have thus shown that the pair $P_i^L \| P_{i+1}^R$ is deadlock-free.

We leave it to the reader to fill in the details, and to use similar techniques for case (1), the pair $Q_0^L \| P_1^R$. The conclusion at this stage in the analysis is that the original network is pairwise free of deadlock. We now return to the original network structure.

Now we can use Theorem 1 to deduce the existence, in any deadlock state, of a cycle. Thus, deadlock is possible only if either each process is waiting for its successor or each process is waiting for its predecessor. In order to show that these cycles are impossible, we first prove that the property that there is exactly one process with the token is an invariant for the network.

Now let $\#a$ denote the number of occurrences of event a in the current trace of the entire network. Clearly, process 0 has the token when #n−1.priv = #0.priv, and does not have the token when #n−1.priv = #0.priv + 1. For $i \neq 0$, process i has the token when #i−1.priv = #i.priv − 1, and does not have it when #i−1.priv = #i.priv. It is easy to prove from the process definitions that, for all i, one of these two possibilities always occurs. Initially one process (numbered 0) has the token; and every communication of the form i.priv affects the two adjacent processes in whose alphabets it is. Therefore, there is always exactly one process with the token.

To rule out a "clockwise" cycle in which (for each i) process i is waiting for process $i+1$, note that this can only occur if each process i does not have the token. This violates the invariant property, showing that no such cycle can arise.

To rule out the "anticlockwise" cycle, note that process $i+1$ can only have an ungranted request to process i when it can also communicate outside the alphabet of the network (i.e., in the initial state of P_i or Q_i). This is not a *strong* request. Hence, there can never be a cycle of strong ungranted requests with each process waiting for its predecessor. ∎

The proof given for this example assumed that there is exactly one token. A modification of this proof goes through whenever the network is started with *at least* one token in the ring. Of course, in the degenerate case where there is *no* token, deadlock must occur.

7. Deadlock Analysis in Arbitrary Networks.

Theorem 1 is only applicable to unidirectional networks. The token ring described above served to illustrate this class of networks. More general results are needed to tackle non-unidirectional systems such as the Dining Philosophers. We seek results which allow us to deduce that deadlock can only be caused by some sort of global misbehaviour (for instance, proper cycles of ungranted requests), and which are general enough at least to eliminate deadlock from trees. We can expect the preconditions of such theorems to involve a certain amount of local analysis; in the unidirectional case this amounted to a check that all individual processes and all pairs of processes were deadlock-free.

It is clearly of practical importance to keep the amount of local checking as small as possible: any more than pairwise checking could easily prove prohibitively expensive in calculation. Unfortunately, even simple types of network can sometimes require much more than pairwise analysis. We have already seen that there exist networks in which deadlock is a global property: for example, the deadlocked chain network, all of whose non-empty proper subnetworks were deadlock-free. That example also demonstrates that even for simple communication graphs there is no simple bound on the size (or even on the diameter, to use the graph-theoretical term) of the local regions requiring analysis: it might be necessary to analyse the entire graph at once.

For any particular network it is possible to identify a collection of local regions (called "competition sets" in [6]) which can form the basis of a deadlock analysis, but this may not break down the problem into significantly smaller subnetworks. We will return to the problem of decomposing a general network into separable regions later. For now we will concentrate on types of network which require only pairwise checking. And instead of competition sets we introduce some sharper material on "conflict".

In a non-unidirectional network we cannot expect that proper cycles of *strong* requests will again correspond precisely to deadlock. Nevertheless, an analysis based on proper cycles of requests will clearly be enough to exclude deadlock in trees, since (as we said earlier) trees have no cycles. Our task is therefore to find extensions of the unidirectional condition which are still pairwise checkable and which are strong enough to generate proper cycles of requests in deadlock states. The following definitions, formalizing notions of conflict between pairs of processes, are motivated by this aim.

Conflicts.

Basically, a conflict is a degenerate cycle of two ungranted requests. We give a general definition of a Γ-conflict, or conflict relative to a set Γ of events: a Γ-conflict is a cycle of two ungranted requests with respect to Γ. We will normally be concerned with the case where Γ is the vocabulary of the global network containing the two conflicting processes, since if any individual process can perform an event outside of the network's vocabulary, then the network cannot be deadlocked. Since the emphasis here is on pairwise analysis, we find it convenient to use the abbreviation *pair* for a subnetwork with two nodes.

Definition 16. A state $\sigma = (s, \langle X, Y \rangle)$ of the pair $\langle P, Q \rangle$ is a Γ-conflict if each has an ungranted request to the other (with respect to Γ), i.e., if:

$$P \xrightarrow{\sigma,\Gamma}\bullet Q \quad \text{and} \quad Q \xrightarrow{\sigma,\Gamma}\bullet P.$$

The state is a *strong* Γ-conflict if (at least) one of these ungranted requests is strong i.e., if additionally

$$P \xRightarrow{\sigma,\Gamma}\bullet Q \quad \text{or} \quad Q \xRightarrow{\sigma,\Gamma}\bullet P.$$ ∎

A Γ-conflict is a state where each of P and Q wants to communicate with the other, neither can communicate outside Γ, and they cannot agree on a joint communication. The conflict is strong if one of the two processes is completely blocked by the other one.

Definition 17. A pair $\langle P, Q\rangle$ is *free of* Γ*-conflict* if none of its states is a Γ-conflict. A process is *free of strong* Γ*-conflict* if none of its states is a strong Γ-conflict. ∎

Informally, $\langle P, Q\rangle$ is conflict-free with respect to Γ if P and Q can never simultaneously be offering to communicate with each other without either agreeing on some action or one of them being able to communicate outside Γ. The pair is free of strong Γ-conflict if it can never get into a state where one process can only proceed by communication with the other, which is offering it only inappropriate communications and cannot communicate outside of Γ. Clearly, each pair which is free of Γ-conflict is also free of strong Γ-conflict.

We extend these notions of conflict-freedom to a general network as follows.

Definition 18. A network $V = \langle P_i \mid 1 \leq i \leq n\rangle$ is conflict-free iff each pair $\langle P_i, P_j\rangle$ is conflict-free with respect to the vocabulary A of V. The network is free of strong conflict iff each pair is free of strong A-conflict. ∎

We note that, if $\Gamma' \subseteq \Gamma$, then freedom from Γ-conflict (or strong Γ-conflict) implies freedom from Γ'-conflict (or strong Γ'-conflict). Since the vocabulary of a subnetwork is a subset of that of whole network, it follows that both of these properties are hereditary. Thus, conflict-freedom and strong conflict-freedom can be proved by purely local analysis.

Here are three elementary results on conflict-freedom, giving some simple criteria which guarantee freedom from conflict. In each case we assume that $\alpha P \cap \alpha Q \subseteq \Gamma$, which will certainly be the case when Γ is the vocabulary of a network containing P and Q. We also assume that P and Q are deadlock-free, which will be true when the network containing P and Q is busy.

The first result is almost trivial:

Lemma 5. *The pair* $\langle P, Q\rangle$ *is free of* Γ*-conflict whenever* $|\alpha P \cap \alpha Q| \leq 1$.

Proof. If two processes have no event in common, they never try to communicate with each other and the question of conflict is vacuous. Conflicts never arise between deadlock-free processes with a unique event in common: if each is offering to communicate with the other, they must be agreeing on this event. ∎

The second result applies if at all stages, whenever P and Q are trying to communicate their choice is restricted to a unique event.

Lemma 6. *The pair* $\langle P, Q\rangle$ *is free of* Γ*-conflict if there is an infinite sequence of events common to the alphabets of* P *and* Q*, say* $u \in (\alpha P \cap \alpha Q)^\omega$*, such that in every trace of* P *and in every trace of* Q *the communications between* P *and* Q *form a prefix of* u*, i.e.*

$$\forall s \in \mathit{traces}(P) \cup \mathit{traces}(Q), \quad s\restriction(\alpha P \cap \alpha Q) \leq u.$$

Remark. When u has the special form t^ω for some finite trace t this is essentially a cyclic communication condition.

Proof. Let w be a trace of $P\|Q$, so that $s_1 = w\restriction\alpha P$ is the corresponding trace of P and $s_2 = w\restriction\alpha Q$ that of Q. Clearly, $s_1\restriction(\alpha P \cap \alpha Q) = s_2\restriction(\alpha P \cap \alpha Q)$. By hypothesis this trace is a prefix of u. If neither of the processes is able to communicate outside of Γ on the next step, the event in u immediately following this prefix must be a communication on which the two processes can agree. ∎

The third result applies in case the behaviour of P and Q is such that in all relevant states one of them is *acquiescent*, in the sense that it cannot refuse anything the other one may offer. The states to which this condition must apply are those in which P and Q are refusing to do any external event, but neither is refusing the entire alphabet of the other. It is easy to see that this condition prevents conflict when P and Q are known to be deadlock-free. Hence:

Lemma 7. *The pair $\langle P, Q\rangle$ is free of Γ-conflict if for every trace s of $P\|Q$, whenever $(s\restriction\alpha P, X) \in \bar{\mathcal{F}}\llbracket P\rrbracket$ and $(s\restriction\alpha Q, Y) \in \bar{\mathcal{F}}\llbracket Q\rrbracket$ satisfy*

$$X \supseteq \alpha P - \Gamma,\ Y \supseteq \alpha Q - \Gamma\ , X \not\supseteq \alpha Q,\ Y \not\supseteq \alpha P,$$

it follows that either

$$X \cap \mathit{initials}(Q\ \underline{\mathit{after}}\ s\restriction\alpha Q) = \emptyset$$
$$\mathit{or}\quad Y \cap \mathit{initials}(P\ \underline{\mathit{after}}\ s\restriction\alpha P) = \emptyset.$$

∎

Each of the above criteria in Lemmas 5, 6 and 7 is more general (but more complex) than the previous one. Of course, each also implies freedom from strong Γ-conflict. It is also easy to find yet weaker conditions than those of Lemma 7 which imply freedom from strong conflict; an obvious one is that no process ever makes a strong request.

To illustrate these concepts of conflict-freedom we return to some of our earlier examples.

Example 1: Deadlocked Chain.

Here, no pair of adjacent processes is conflict-free, and the pairs $\langle P_0, P_1\rangle$ and $\langle P_{n-1}, P_n\rangle$ are not even strong conflict-free. It is easy to see, however, that the other pairs of adjacent processes are strong conflict-free: each P_i $(0 < i < n)$ is always in a position where it can either talk to both of its neighbours or it can refuse all communications with the only neighbour it can talk to. In neither of these cases can it be the blocked process in a strong conflict. ∎

Example 2: Dining Philosophers.

This network is conflict-free. For the philosopher–fork combinations observe that there is a pattern of cyclic communication; for example, the communications between PHIL_i and FORK_i always form a prefix of the sequence $\langle$i.picks.i, i.puts.i$\rangle^\omega$. Hence, by Lemma 6, each philosopher–fork pair is conflict-free. For the philosopher–butler combinations, note that whenever the butler can talk to a philosopher he cannot refuse anything the philosopher might offer, so that (in the terminology used above), the butler is acquiescent. Hence, by Lemma 7, each philosopher–butler pair is conflict-free.

Example 5: Token Ring.

Here we have $\alpha P_i \cap \alpha P_{i+1} = \{$i+1.find, i.priv$\}$, and it is easy to see that the interactions between P_i and P_{i+1} follow the cyclic communication pattern $\langle$i+1.find, i.priv$\rangle^\omega$. Hence the network is conflict-free (and also free of strong conflict) by Lemma 6. ∎

Of course, our reason for the invention of the conflict-freedom conditions is that they enable us to establish some useful results on deadlock.

Deadlock Analysis in Conflict-free Networks.

Theorem 2. *Let $V = \langle P_i \mid 1 \leq i \leq n \rangle$ be a busy network with vocabulary Λ. If V is free of strong Λ-conflict, any deadlock state of the network contains a proper cycle of ungranted requests with respect to Λ. If V is conflict-free then any deadlock state contains a proper cycle of ungranted requests $\langle i_0, \ldots, i_{r-1} \rangle$ with respect to Λ $(r \geq 3)$, such that the only requests being made in this state by the processes involved in the cycle are the requests recorded in the cycle.*

Proof. Similar to that of Theorem 1, using the fact that freedom from strong Λ-conflict implies that in any deadlock state, whenever there is a request from P_i to P_j and P_j is not itself deadlocked, there must be a request from P_j to some other process P_k with $k \neq i$. ∎

Note that Theorem 1 is a corollary to Theorem 2, since in a unidirectional network every request in a deadlock state must be a strong request.

It is not hard to improve this result slightly to allow for one pair of processes in the network to fail to be free of strong conflict, if instead the pair is deadlock-free. That we cannot go further and allow two pairs of processes to be deadlock-free but not free of strong conflict is shown by the deadlocked chain example.

Theorem 3. *The conclusions of Theorem 2 remain valid, even if we allow one pair of processes $\langle P_i, P_j \rangle$ to be deadlock-free but not free of strong conflict.* ∎

Corollary. *If a tree network satisfies the conditions of Theorem 2 or Theorem 3 it is hereditarily deadlock-free.* ∎

Again, we return to the examples to demonstrate the uses of these results.

Example 1: Deadlocked Chain.

Even though the pairs $\langle P_0, P_1 \rangle$ and $\langle P_{n-1}, P_n \rangle$ have strong conflicts, it is easy to prove (using Lemma 3 and Lemma 4) that they are both deadlock-free. We already know that all other pairs are free of strong conflict. Since the removal of any number of processes (strictly between 1 and n) leaves us with a collection of chains, none of which contains both of these pairs of processes, we conclude by the above corollary to Theorem 3 that every non-empty proper subnetwork is deadlock-free, confirming our earlier prediction to this effect. ∎

Example 2: Dining Philosophers.

Since we have already shown that this network is busy and conflict-free, Theorem 2 implies that any deadlock state contains a proper cycle of ungranted requests. We sketch a proof demonstrating the impossibility of such a cycle as follows.

- The network structure implies that such a cycle must involve at least one fork process and therefore must contain an edge from a philosopher to a fork.
- No philosopher can have an ungranted request to a fork unless that fork has been picked up by the other adjacent philosopher. Because of its position in the cycle, this fork must have an ungranted request to the philosopher who is currently holding it.
- While a philosopher holds a fork he cannot communicate with the butler. Therefore, the only process to which this second philosopher can have an ungranted request is his other fork.
- We can clearly continue this argument to show that the cycle must run through all of the philosophers and forks (either clockwise or anticlockwise).
- Further, we can deduce that each of the philosophers holds exactly one fork. This means that so far each philosopher has communicated one more 'enters' events than 'leaves'.

However, the butler process was designed to prevent this state arising in more than four philosophers at once. Hence, this contradiction proves that the network is free of deadlock. ∎

Example 5: Token Ring.

By applying Lemma 6 and Theorem 2 we can prove deadlock-freedom of the token ring more easily than by our earlier techniques. We have already shown (easily) that the ring is free of strong conflict. By Theorem 2, this means that any deadlock must be caused by a cycle of ungranted requests, which in this particular network means that either each process is waiting for its successor or each process is waiting for its predecessor. In the previous proof for this network we had to go through a much more involved pairwise analysis to reach this stage in the argument. The remainder of the proof is the same. ∎

On achieving conflict-freedom.

Networks in which the design of processes and the potential communication patterns are rather symmetric may fail to be conflict-free; a good example is provided by the deadlocked chain with its two end processes removed. Typically this happens where there are messages which P might wish to send to Q and vice-versa. Conflict typically appears when P and Q are both waiting for a message from the other, but neither is ready to send one.

It is hard, however, to imagine a reasonable example (of a deadlock-free network) which has conflict and yet cannot be redesigned to achieve freedom from strong conflict. A communication which is being offered to the blocked process in a conflict state can never occur, and it is therefore quite likely that this event can be removed from the design of the blocked process without changing the behaviour of the network as a whole. For example, in the typical conflict described above there must be some mechanism, either a message from some other process or the external environment, which could generate a message from P to Q, or else there is no point in Q waiting for it. We illustrate this potential need to redesign processes so that networks become conflict-free with yet another variant of the Dining Philosophers.

Example 6: Conflicting Philosophers.

If we replace the butler process of Example 2 with the following process, which has different traces from those of the original butler, but the same alphabet, the resulting network would fail to be conflict-free:

$$\begin{aligned}
\text{BUTLER}' &= \text{ADMIT}_4, \\
\text{ADMIT}_4 &= \Box_{i=0}^{4}(\text{i.enters} \rightarrow \text{ADMIT}_3), \\
\text{ADMIT}_k &= \Box_{i=0}^{4}((\text{i.enters} \rightarrow \text{ADMIT}_{k-1}) \,\Box\, (\text{i.leaves} \rightarrow \text{ADMIT}_{k+1})), \quad (k = 1, 2, 3) \\
\text{ADMIT}_0 &= \Box_{i=0}^{4}(\text{i.leaves} \rightarrow \text{ADMIT}_1).
\end{aligned}$$

In this network the pairs consisting of a philosopher and butler are not even free of strong conflict, since when the butler has admitted the other four philosophers he is quite happy to let the remaining philosopher "leave" even though he is blocking him by preventing him from "entering". The snapshot graph describing this state is the same as for the earlier version (Diagram *), except that there is an additional edge, from BUTLER to PHIL_0.

However, the behaviour of the network as a whole is unaltered if we replace the old butler by this one (because the remaining philosopher of course is not even trying to leave at this point). There is a sense in which the network's correctness (i.e. deadlock-freedom) depends more on the overall structure of the network than it did with the original definition of the butler. ∎

In summary, then, we believe that conflict-freedom is a widely applicable condition, since we are aware of no natural and correct network which fails to meet this condition and cannot be redesigned to yield a behaviourally equivalent network that is indeed free of strong conflict.

8. Network Decomposition.

In this section we introduce a general method for decomposition of a network into sub-networks which may be treated largely independently for the purposes of deadlock analysis. The role of conflict-freedom of a pair of processes is crucial in this method.

If V is a network, we define the *disconnecting edges* of V to be the edges of the communication graph whose removal would increase the number of connected components. The disconnecting edges are precisely the edges which cannot be part of any cycle in the graph. We also define the *essential components* of V to be the connected components of the graph that remains after all disconnecting edges are removed. (In graph-theoretic terms, the essential components are the maximal edge bi-connected subgraphs.) The essential components of a tree are its individual processes, and every edge is a disconnecting edge in a tree. Even in a general network the essential components themselves always form a tree when an edge is drawn between a pair of essential components if and only if the alphabets of two of their members intersect non-trivially.

In any busy network free of strong conflict, Theorem 2 showed that deadlock can only be caused by a cycle of ungranted requests. It is not difficult to see that such a deadlock-causing cycle necessarily lies in one of the essential components of the network. It follows that if none of the essential components of such a network can contain such a cycle, the whole network is deadlock-free. Note, however, that it is necessary to prove the absence of cycles of ungranted requests in an essential component with respect to the alphabet of the whole network, not just with respect to that of the essential component.

These facts, and analysis of conflict-freedom, help to establish the following result.

Theorem 4. *Suppose V is a network with essential components $V_1, \ldots, V_k$ where the pair of processes joined by each disconnecting edge are conflict-free with respect to A, the vocabulary of V. Then if each of the V_i is deadlock-free, so is V.*

Proof. This result follows from the associative law of PAR, for the behaviour of V is the same as that of the network whose nodes are the parallel compositions of V's essential components.

$$\mathrm{PAR}(V) = \mathrm{PAR}(\langle \mathrm{PAR}(V_i) \mid 1 \leq i \leq k \rangle)$$

The communication graph of this new network is a tree as observed above. It is busy by assumption that each V_i is deadlock-free. To show conflict-freedom we argue as follows.

- If the pair $[\mathrm{PAR}(V_i) \| \mathrm{PAR}(V_j)]$ were in conflict with respect to the vocabulary of V, their alphabets would intersect and so there would be a (necessarily unique) pair of processes $P \in V_i$ and $Q \in V_j$ such that $\alpha P \cap \alpha Q \neq \emptyset$.
- If $(s, \langle X, Y \rangle)$ were a conflict of $[\mathrm{PAR}(V_i) \| \mathrm{PAR}(V_j)]$ then, by definition of $\mathrm{PAR}(V_i)$ and $\mathrm{PAR}(V_j)$ there would be states $(s \upharpoonright \alpha V_i, \underline{X})$ and $(s \upharpoonright \alpha V_j, \underline{Y})$ of V_i and V_j corresponding to the failures $(s \upharpoonright \alpha V_i, X) \in \mathcal{F}[\![\mathrm{PAR}(V_i)]\!]$ and $(s \upharpoonright \alpha V_j, Y) \in \mathcal{F}[\![\mathrm{PAR}(V_j)]\!]$.
- If X' and Y' are the components of $\underline{X}$ and $\underline{Y}$ corresponding to P and Q, the conditions ensure that $X \cap \alpha V_j = X' \cap \alpha Q$ and that $Y \cap \alpha V_i = Y' \cap \alpha P$.
- It follows that $(s \upharpoonright (\alpha P \cup \alpha Q), \langle X', Y' \rangle)$ is a A-conflict of $[P \| Q]$, contrary to assumption, giving the desired contradiction.

∎

Thus we have some results identifying parts of networks which can, from the point of view of deadlock analysis, be regarded as independent. Of course, this is only useful in practice if the network decomposes into significantly smaller or simpler subnetworks: we can reasonably expect a small network to be much easier to analyse than a big one.

To illustrate the use of the type of network decomposition we propose, and to demonstrate the use of Theorem 4, here is another example.

Example 7: Interconnected Token Rings.

Suppose that, instead of the single ring which we saw earlier, it is for some reason desired to implement a system of interconnected rings. Provided the connection structure between the rings is a (connected) tree, it is straightforward to develop a deadlock-free system from our earlier work and Theorem 4. The rings will consist of the processes P_i and Q_i as before (P_i for processes with no token initially, Q_i for processes having one) and a new type of process L_i for linking two rings. Each link will have one of these L processes at each end, rather than having one process sitting in both rings, for the latter would not put the two rings in different essential components. These link processes never initially contain a token and always pass one on immediately after having received one. They do, however, remember how many tokens there are in each of the two components that would be created were its link to be cut. Provided it is correctly initialized such a process can always know these numbers since changes can only come about when the process itself effects the transfer. We give here the definition of a link process for the case where there is only one token.

The definition given here assumes that the link process is to replace process i in a ring where all the P_j and Q_j have the same alphabets as before except that, to keep the internal alphabets of distinct rings disjoint, the events of each ring are tagged with a label (e.g. ρ, ν). If a link process is to connect ring ρ to ring ν and is to be placed at position i in ring ρ we will use the notation $L_i(\rho, \nu)$ to denote it. Such a link process will have alphabet

$$\alpha L_i(\rho,\nu) = \{\rho.\text{i.find}, \rho.\text{i+1.find}, \rho.\text{i.priv}, \rho.\text{i+1.priv}, \nu.\rho.\text{req}, \rho.\nu.\text{req}, \rho.\nu.\text{pass}, \nu.\rho.\text{pass}\}.$$

There is an obvious directionality associated with a link process, and we may correspondingly refer to the two sides of a linking arc as the partition of the network which would occur if the linking arc were cut. Somewhat loosely (but, we hope, in accordance with intuition) we refer, when describing the behaviour of the link process, to these two subnetworks as "its" side and "the other" side. Informally, we specify the behaviour of a link process as follows.

When the token is on its side of the linking arc, the link process is prepared to accept either a request for the token from the ring (the event ρ.i.find), or a request from the other side of the link ($\nu.\rho$.req). In either case since the token is to be found on its side of the link the process then makes a request to its neighbour (the event ρ.i+1.find). Once the token has been found and reaches the link (ρ.i.priv) the link process will respond appropriately to the request that began this activity: either pass the token on to the next process in the ring (ρ.i-1.priv) or pass it over the to the other side ($\rho.\nu$.pass).

When the token is on the other side of the link, the link process is prepared only to accept a request for the token (ρ.i.find) and to pass this request on over to the other side ($\rho.\nu$.req). When the token arrives and is passed over ($\rho.\nu$.pass), the link process hands it over to the neighbour who requested it (ρ.i.priv).

Formally, we describe the behaviour of a link process by means of two auxiliary process definitions: when the token is on its side the link process is denoted $L_i^+(\rho, \nu)$, and when

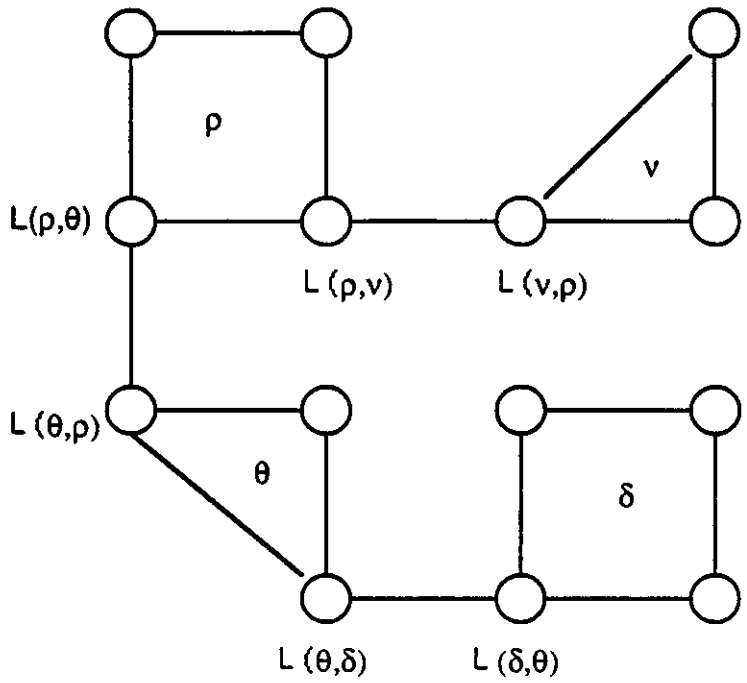
the token is on the other side the process is denoted $L_i^-(\rho,\nu)$; these auxiliary processes are defined (omitting the ring names) by:

$$\begin{aligned} L_i^+ &= (\nu.\rho.\text{req} \rightarrow \rho.\text{i+1.find} \rightarrow \rho.\text{i.priv} \rightarrow \rho.\nu.\text{pass} \rightarrow L_i^-) \\ &\quad \Box\ (\rho.\text{i.find} \rightarrow \rho.\text{i+1.find} \rightarrow \rho.\text{i.priv} \rightarrow \rho.\text{i-1.priv} \rightarrow L_i^+) \\ L_i^- &= \rho.\text{i.find} \rightarrow \rho.\nu.\text{req} \rightarrow \nu.\rho.\text{pass} \rightarrow \rho.\text{i.priv} \rightarrow L_i^+ \end{aligned}$$

This formal description is intended to correspond to the informal remarks above; for instance, the passage of the token across a linking arc results in a change of "sign".

Note that we allow networks in which some of the rings consist only of link processes. Except in the trivial case where there is only one ring, all rings must contain at least one link process. It should be obvious that the essential components of a multiple ring system like this are just the rings; the disconnecting edges are the links between rings.

When the network is set up all pairs of link processes are in opposite states (one +, one −), since the token is on one side or the other. Here is an example of such a network:



Linked Rings
DIAGRAM 6

To prove the link pairs conflict-free it is sufficient to prove that any adjacent pair with opposite signs, say $L_i^+(\nu,\rho) \| L_j^-(\rho,\nu)$, are. This follows from Lemma 6 since the communications between this pair are cyclic, repeating $\langle \rho.\nu.\text{req}, \nu.\rho.\text{pass}, \nu.\rho.\text{req}, \rho.\nu.\text{pass} \rangle$.

The proof that the individual rings are deadlock-free is essentially the same as in the earlier example. (Note that each ring is still unidirectional, even though the whole network is not.) The invariant which prevents there being a cycle of strong requests is now that in

each ring the number of nodes holding a token plus the number of "negative" link nodes (for which the token is on the other side) always equals one.

There are several interesting ways to extend this idea to deal with multiple tokens, but we will not discuss these here. ∎

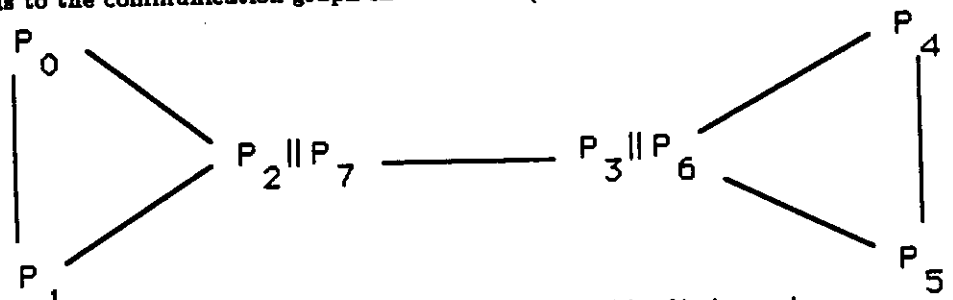
The methodology based on Theorem 4 relies on showing first that the essential components of a network are deadlock-free, and then that the links between essential components are well behaved (conflict-free). One could, of course, relax the condition that the links between essential components are conflict-free if there were some other means of showing the interactions between the whole essential components to be *strong* conflict-free. One assumption that is *not* in general strong enough for this is that the network V is strong conflict-free; the reader might like to confirm this by considering the following example.

Example 8: A Bowtie Network.

Let P_i $(i = 0 \ldots 7)$ be the following simple processes:

$$P_i = i \rightarrow i+1 \rightarrow P_i,$$

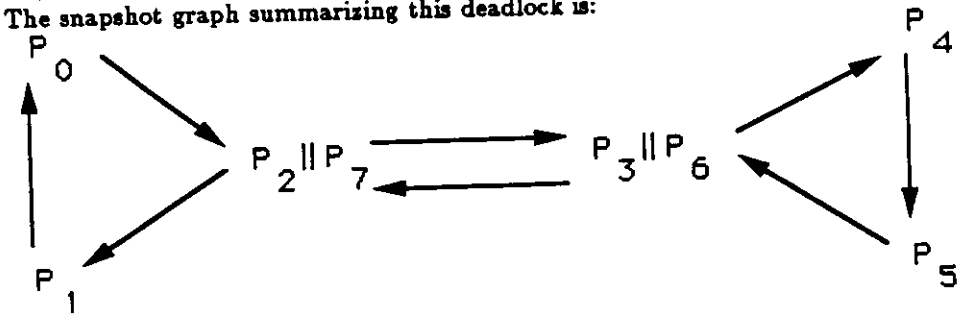
where arithmetic is done modulo 8 and where processes have the obvious alphabets. Consider the network formed by the six processes $P_0, P_1, P_4, P_5, [P_2 \| P_7], [P_3 \| P_6]$. This corresponds to the communication graph shown below (and hence the name "bowtie"):



Communication Graph of Bowtie Network

DIAGRAM 7

This network is deadlocked, even though its essential components $\langle P_0, P_1, [P_2 \| P_7] \rangle$ and $\langle P_4, P_5, [P_3 \| P_6] \rangle$ are each deadlock-free and the one disconnecting edge is strong conflict-free. The snapshot graph summarizing this deadlock is:



Snapshot Graph of Bowtie Network

DIAGRAM 8

The deadlock state has two cycles of ungranted requests, one in each of the two essential components. ∎

A Design Rule Guaranteeing Deadlock-Freedom.

We have so far proven some results which show that, under certain circumstances (such as unidirectionality, or conflict-freedom) deadlock can only be caused by cycles of ungranted requests. This allowed us to tackle deadlock analysis by proving the non-existence of such cycles. For tree networks this is sufficient to prove deadlock-freedom directly. However, for general networks the problem remains of establishing that cycles of ungranted requests are impossible. So far, our methods for doing this have been rather *ad hoc*: we have relied largely on case analysis of traces and refusals, and finding of invariant properties that are false in all potential deadlocks. This type of case analysis was made simpler by selective use of hiding. Nevertheless, we have so far not introduced any general results which can themselves *directly* prove the deadlock-freedom of a network whose communication graph has cycles. All we have managed to do for those is to get a better understanding of the ways deadlock can arise.

It is our intention that the work of this paper should serve as the foundations for the development of more specific techniques for proving deadlock-freedom. A wide class of such techniques, mainly based on the concept of *variants*, have already been described in [19]. In this section we give another example: a theorem stating some simple (though admittedly rather curious) conditions under which deadlock cannot arise. One may regard these conditions as imposing a *design rule* which, if adhered to, guarantees absence of deadlock directly, without need for detailed investigation into cycles of requests. The utility of this particular design rule is demonstrated by applying it to a particular example network which does meet these conditions: a mail system involving a ring of user processes.

Theorem 5. *If, in the busy, strong conflict-free network, whenever a process P has an ungranted request to another process Q in the same essential component, P was the last process in that component with whom Q communicated, the network is deadlock-free.*

Proof. By Theorem 2, any deadlock state has a cycle of ungranted requests necessarily lying in a single essential component. Suppose that the most recent communication between two consecutive elements of this cycle was between P and Q, with P now waiting for Q. Now P must be blocking some process other than Q in the cycle, but this is impossible by assumption. ∎

Example 9: Message-passing ring.

Consider a message-passing ring in which a number of users can send mail to one another. Each user is associated with a node; the nodes are connected in a ring; a node may

(i) accept a message from its user and pass it to its clockwise neighbour, or

(ii) accept a message from its anticlockwise neighbour and give it to its own user or pass it clockwise as appropriate.

Clearly if each node has the capacity to store only one message at a time, the system may deadlock. (When all users simultaneously decide to output a message, none of these messages can ever leave its source node.)

However, if each node has capacity bigger than one, and if also each node has a non-zero limit strictly smaller than its capacity such that, when the node contains the limit or more items it will only accept a message from the ring (i.e., not from its own user), the network is deadlock-free. Intuitively this is because the network can never become "full" (the last message entered would need to be into a node with only one slot left, but this is not allowed).

Of course, the simplest example of such a network is where each node has capacity two and will only accept input from the environment when empty. However, the parameters of a practical implementation would be more generous. ∎

9. Comparison and Conclusions.

We have shown how to use the failures model of CSP to provide a succinct and mathematically tractable representation of deadlock. We have been able to use the model in proofs of some interesting and useful results on the analysis of deadlock in networks, and then to prove absence of deadlock in a variety of examples. We have focussed firstly on results pertaining to unidirectional systems (e.g. Theorem 1) and also on results applicable to general networks (Theorems 2, 3, 4, 5). All of our methods support hierarchical analysis of networks, assuming that deadlock-freedom of individual nodes has already been established. We singled out for special attention the class of tree networks (Corollaries (i) and (ii) to Theorem 1, and the Corollary to Theorem 3.) We demonstrated the power of our techniques by applying them to a collection of example networks. We argued that the concepts developed in this paper (such as cycles of requests, conflict-freedom, essential components) provide the basis for a battery of theorems on deadlock analysis that are widely applicable.

The theorems of this paper are only a sample of a large class of general results which we and others have derived for analysing the deadlock properties of networks. As stated earlier, the results of the current paper have already been used extensively in [19], where more specialized techniques were developed such as those based on "variants". A variant is a function from the states of the components of a network into a partial order. These simple combinatorial techniques often allow localized proofs that no cycle of ungranted requests can arise, even in networks whose communication graphs have many cycles. These techniques seem to be widely applicable but are certainly not complete, since they cannot handle all networks.

We analysed in Example 10 a simple message-passing network. A far more complex and sophisticated message-passing network has been developed in [18] (as an occam program) based on this one, with analysis again based on Theorem 5. It makes use of the general topology of a network to pass messages efficient routes, but uses the ring behaviour as a last resort to avoid deadlock. It would be interesting to see if this theorem is applicable to any other types of example.

Again, Theorem 5 was specialised to networks which have been designed to meet a simple "design rule". Adhering to the design rules will then guarantee deadlock-freedom. There is much to be said for such design rules, especially if they are chosen to be easy to apply in practice. The discovery of more of them should therefore be regarded as a priority.

Related Work.

Dathi's thesis [8] will give a broad survey of techniques both new and old which can be brought into this framework. He also gives a very thorough comparative survey of the relative power and applicability of various deadlock-proving techniques.

Dijkstra [9] proved some theorems on the absence of deadlock in unidirectional networks for the special case in which the patterns of communication were cyclic: each process rotated its communication requests in cyclic order through its immediate neighbours. Dijkstra stated that his results were applicable in a more general setting, and [19] has demonstrated that this is indeed the case.

Chandy and Misra have developed a method for proving deadlock-freedom using priorities [7]. Roughly speaking, they proposed that, for every state of a network, one should

assign priorities to the edges of the communication graph in such a way that every process can always communicate over its adjacent edge of highest priority. The existence of such a prioritization is equivalent to a certain strengthened version of our deadlock-freedom condition. In fact, their condition implies that every non-empty subnetwork can make progress in future. To use such a technique requires a global analysis of all states of the network, which may in practice be expensive because of the exponential growth in the size of the state space of a network as a function of the number of processes. It is for reasons such as these that we advocate localized analysis wherever possible. Dathi [8 represents the method of [7] in our setting.

In [16] Reisig discusses proof rules for deciding if deadlocks can occur in distributed systems of sequential processes which communicate deterministically by means of buffers. He uses a Petri net model and gives a characterization of deadlock states.

The work of Apt *et al.* [1,2,3] on reasoning about CSP programs includes some methods for analysing deadlock. Essentially, this work is based on a rather different approach from ours: a CSP program is first transformed syntactically into a program in a *guarded command* language [11] which no longer involves communication. In this transformation syntactically matching pairs of communications are combined into assignment statements to mimic the affect of synchronizing an input with a matching output. Then one reasons about the absence of deadlock by finding a global invariant which guarantees that no deadlock state can be reached, because it is false in deadlock states.

Dynamic Networks.

In this paper we have focussed entirely on static networks in which both the number of processes and their topology remain unchanged throughout the execution history of the network. Dynamically changing networks may arise in practice, for instance when recursion is used inside a parallel composition: it may be possible for a process to spawn one or more recursive parallel instances of itself, or indeed of other processes, during execution. The most difficult problem arising when trying to extend techniques such as ours to cover dynamically evolving networks is that of keeping track of the network's structure during execution. This means that a lot of specialized concepts and notations have to be introduced to deal with network structure. Of course deadlock only arises in a static situation, so that the ideas of this paper carry over to the analysis of dynamic networks more or less wholesale. We plan to develop the theory of dynamic networks, including deadlock analysis, in a future paper.

Acknowledgements.

The authors would like to thank C. A. R. Hoare for his many helpful suggestions and discussions, and for his encouragement and guidance during the development of the failures model for CSP. Discussions on deadlock analysis with Krzysztof Apt, Naiem Dathi, Jay Misra, Ernst-Rudiger Olderog, David Reed and Wolfgang Reisig have been very useful.

10. References.

[1] Apt, K. R., *A Static Analysis of CSP Programs*, in: Logics of Programs, Proceedings, Springer LNCS Vol. 164, pp. 1-17 (1983).

[2] Apt, K. R., (editor), Logics and Models of Concurrent Systems, Springer Verlag NATO ASI Series, Series F, Vol. 13 (1985).

[3] Apt, K. R., Frances, N., and de Roever, W. P., *A Proof System for Communicating Sequential Processes*, ACM TOPLAS, vol. 2, no. 3, pp. 359-385 (1980).

[4] Brookes, S. D., Hoare, C. A. R., and Roscoe, A. W., *A Theory of Communicating Sequential Processes*, JACM (July 1984).

[5] Brookes, S. D., and Roscoe, A. W., *An Improved Failures Model for Communicating Processes*, Proc. NSF-SERC Seminar on Concurrency, Springer LNCS Vol. 197, pp. 281-305 (1985).

[6] Brookes, S. D., and Roscoe, A. W., *Deadlock Analysis in Networks of Processes*, pp. 305-323 in [2].

[7] Chandy, K. M., and Misra, J., *Deadlock Absence Proofs for Networks of Communicating Processes*, Information Processing Letters, Vol. 9, no. 4, Nov. 1979.

[8] Dathi, N., D. Phil. thesis, Oxford University (forthcoming, 1989).

[9] Dijkstra, E. W., *A Class of Simple Communication Patterns*, EWD643, in: Selected Writings on Computing, Springer Verlag (1982).

[10] Dijkstra, E. W., *Invariance and non-determinacy*, in: Mathematical Logic and Programming Languages, Prentice-Hall.

[11] Dijkstra, E. W., *Guarded Commands, Non-determinacy, and Formal Derivation of Programs*, CACM Vol. 18, No. 8 (August 1975).

[12] Hoare, C. A. R., *Communicating Sequential Processes*, CACM 1978.

[13] Hoare, C. A. R., Communicating Sequential Processes, Prentice-Hall (1985).

[14] INMOS Ltd., The occam programming manual, Prentice-Hall (1984).

[15] Milne G., and Milner R., *Concurrent Processes and their Syntax*, JACM 26, No. 2, pp. 302-321.

[16] Reisig, W., *Deterministic Buffer Synchronization of Sequential Processes*, Acta Informatica 18, pp. 117-134 (1982).

[17] Roscoe, A. W., *A Mathematical Theory of Communicating Processes*, D. Phil. thesis, Oxford University (1982).

[18] Roscoe, A. W., *Routing messages through networks: an exercise in deadlock avoidance*, Proceedings of OUGTM7, Grenoble 1987, published by IMAG.

[19] Roscoe, A. W., and Dathi, N., *The Pursuit of Deadlock Freedom*, Information and Computation, Vol. 75, No. 3 (December 1987).