TRANSPUTER COMMUNICATIONS, VOL. 3(3), 1–18 (AUGUST 1997)

A Design Strategy for Deadlock-Free Concurrent Systems

J.M.R.Martin
Oxford University Computing Services, 13 Banbury Road, Oxford OX2 6NN, UK
P.H.Welch
Computing Laboratory, The University, Canterbury, Kent, CT2 7NF, UK

SUMMARY

When building concurrent systems, it would be useful to have a collection of reusable processes to perform standard tasks. However, without knowing certain details of the inner workings of these components, one can never be sure that they will not cause *deadlock* when connected to some particular network.

Here we describe a hierarchical method for designing complex networks of communicating processes which are deadlock-free. We use this to define a safe and simple method for specifying the communication interface to third party software components. This work is presented using the CSP model of concurrency and the occam2.1 programming language.

KEY WORDS deadlock livelock design concurrency reuse components networks CSP occam

1 INTRODUCTION

Code reusability is an important consideration in large-scale system design. It is an issue which modern programming languages are expected to address. In the concurrent programming language, occam2.1 [1], arbitrarily complex subnetworks of processes may be concealed within a single process definition. The internal structure of such a process may be changed independently from any network in which it is embedded. This is extremely useful for a number of reasons. However, in the context of concurrent systems, it is also potentially dangerous. Unless we know their internal details, how can we be sure that a program which incorporates these components is not prone to deadlock?

Deadlock is a major problem with parallel programming. Safety critical systems must be guaranteed deadlock-free, but this is not an easy task. In [2] and [3] it is shown how to tackle this problem using software *design rules*. By placing minor restrictions on the manner in which concurrent systems are constructed, it is possible to remove completely the spectre of deadlock. Proofs of deadlock-freedom for networks of arbitrary size are reduced to simple checks for local conditions of processes, and topological properties of the network configuration.

In this paper we shall build on that work to show how deadlock-free networks may be built hierarchically, incorporating reusable components. This provides an efficient method for modular programming in occam2.1 that offers security guarantees that are very difficult to obtain from traditional approaches to concurrent system design.

CCC 1070–454X/97/030001–18

Received 7 March 1996
Revised 15 January 1997

In section 2 we describe the deadlock analysis terminology of Roscoe and Dathi, which is built upon the firm mathematical foundations of CSP. In section 3 we present a formal definition of the *client-server* protocol which has previously existed only as a collection of informal rules, potentially open to misinterpretation. We then prove a theorem about how this protocol may be used to build deadlock-free networks and illustrate this by two examples: an occam2.1 process farm and a message router. In section 4 we show how to use the client-server protocol to build deadlock-free networks hierarchically from composite sub-components. This is important for the design of large systems.

There are certain other design techniques which are known to yield deadlock-free concurrent systems. In section 5 we describe a technique for integrating subnetworks built by different methods into a client-server system. Section 6 looks at the related issues of divergence-freedom and starvation-freedom and section 7 describes ongoing research.

2 FORMAL ANALYSIS OF DEADLOCK PROPERTIES

The deadlock problem has proved a popular area of research for many years. The early results are largely informal, and suffer from the lack of a suitable underlying mathematical model. This problem was remedied in 1986 by A.W.Roscoe and N.Dathi [4] who provided a powerful suite of tools for deadlock analysis, based on CSP – the mathematical language for reasoning about concurrent systems [5]. This enabled much of the earlier work to be formalised. We shall now describe their terminology.

With each CSP process P is associated an *alphabet*, αP, the set of actions that P may perform. The observable behaviour of P is described by its *failures* and *divergences*. A *failure* of P consists of a pair (s, X) where s is a possible *trace* of events that P may perform, and X is a *refusal set* of P/s – a set of events which if offered to P after it has performed trace s, might be completely refused. A *divergence* of P is a trace s after which P may indulge in an infinite series of concealed actions. A glossary of CSP operators used in this paper is given in Figure 1.

Of particular importance for deadlock analysis are the *maximal* refusal sets of a process, as the more events that are refused the more likely is deadlock to occur. We define the set of maximal failures of P as those failures (s, X) of P where the refusal set X is maximal, *i.e.*

$$maximal\_failures(P) \;=\; \left\{ (s, X) \;\middle|\; \begin{array}{l} (s, X) \in failures(P) \wedge \\ (\forall (t, Y) \in failures(P).\; s = t \;\Rightarrow\; Y \not\supset X) \end{array} \right\}$$

Maximal failures correspond to particular *states* that a process may be in. For instance, consider the process defined by

$$Q \;=\; a \rightarrow (b \rightarrow Q \sqcap c \rightarrow Q), \quad \alpha Q \;=\; \{a, b, c\}$$

After it has performed event a, Q can refuse $\{\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, or $\{a, c\}$. Its maximal refusals at this stage are $\{a, b\}$ and $\{a, c\}$, which correspond to the two states that the process may be in once it has decided which branch of the $\sqcap$ construct to take: *either* it decides to behave like $b \rightarrow Q$ *or* it decides to behave like $c \rightarrow Q$. These are called *stable* states of Q because once they are entered no internal activity is possible, at least until an external event has been performed. Before making the decision as to which branch of the $\sqcap$ construct to take, Q is said to be in an *unstable* state. For the purpose of deadlock analysis, unstable

Processes	
αP	the alphabet of events for process P
$e \rightarrow P$	first event e, then process P
$c ? x \rightarrow P$	first input x from channel c, then process P
$c ! y \rightarrow P$	first output y to channel c, then process P
$P \Box Q$	external choice of P or Q
$P \sqcap Q$	internal choice of P or Q (non-deterministic)
$P \parallel Q$	P in parallel with Q
$P ; Q$	P followed by Q
$P \setminus \{e_1, \ldots, e_n\}$	P with all occurrences of events $\{e_1, \ldots, e_n\}$ hidden
$P \lhd$ *boolean-expression* $\rhd Q$	if *boolean-expression* then P else Q
Special Events	
$\surd$	successful termination
$c.i$	communication of message i on channel c *or* event c_i
$c.i.j$	communication of message (i, j) on channel c *or* communication of message j on channel c_i *or* event c_{i_j}
Traces	
$\langle\rangle$	the empty trace
$\langle a, b, c\rangle$	the trace of events: a then b then c
$traces(P)$	the set of all finite traces of P
$s \downarrow c$	number of communications on channel c in trace s
$\sharp s$	the length of trace s
$s \upharpoonright \{e_1, \ldots, e_n\}$	the trace s with all events removed save those in $\{e_1, \ldots, e_n\}$
P/s	the process describing the behaviour of P after it has performed trace s
Refusals, Failures and Divergences	
$refusals(P)$	the set of sets X which if offered to P might cause it to deadlock immediately
$failures(P)$	the set of pairs (s, X) such that s is a trace of P and X is a refusal of P/s
$divergences(P)$	the set of traces s such that P/s may immediately engage in an infinite series of hidden events

Figure 1. Basic CSP Glossary

states are irrelevant, as a system in such a state is yet to come to rest – when a system is deadlocked there can be no activity at all.

Consider a *network* V of CSP processes, $\langle P_1, \ldots, P_N \rangle$, composed in parallel. Events which lie in the alphabet of two processes of V require their simultaneous participation, and this is the means of communication within the network. It is assumed that the network is *triple-disjoint* – i.e. there is no event which is shared by more than two process alphabets. It is also assumed that the network is *busy* – i.e. each process P_i is non-terminating and individually free of both deadlock and divergence. (It is reasonably straightforward to extend the model to allow for process termination, if required. See [6] for details.)

The behaviour of a network is described in terms of a set of states that it may be in. A state σ of V consists of a trace of events s together with a tuple of refusal sets $\langle X_1, \ldots, X_N \rangle$ such that, for all i, $(s \upharpoonright \alpha P_i, X_i)$ is a maximal failure of P_i. It represents a set of events in which each process may refuse to engage after they have collectively performed the sequence of events in s.

In a *deadlock state* every event is refused by some process, *i.e.*

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

A network which has no deadlock states is said to be *deadlock-free.*

In any state $\sigma = (s, \langle X_1, \ldots, X_N \rangle)$, if there is a process P_i which is ready to perform an event (or events) which lie in the vocabulary of another process P_j (*i.e.* $(\alpha P_i - X_i) \cap \alpha P_j \neq \{\}$), we say that P_i is making a *request* of P_j, and write

$$P_i \xrightarrow{\sigma} P_j$$

If, in addition, P_j is not ready to grant this request (*i.e.* $(\alpha P_i - X_i) \cap (\alpha P_j - X_j) = \{\}$), we say that it is *ungranted* and write

$$P_i \xrightarrow{\sigma} \bullet P_j$$

When a process has all its requests ungranted we say that it is *blocked*. In a deadlock state every process is blocked.

3 CLIENT-SERVER NETWORKS

3.1 Client-Server Protocol

A particularly flexible design rule for deadlock freedom is the *Client-Server Protocol*. This was originally formulated by P. Brinch Hansen [7] in the context of operating systems. It has since been adapted as a means of designing deadlock-free concurrent systems using occam2 (see [3]).

The basic idea is that a process communicates on each of its channels either as a *client* or as a *server*, according to a strict protocol. A network of client-server processes is deadlock-free if it has no cycle of client-server relationships.

The standard definition of the client-server protocol is both informal and intuitive. It runs as follows. If a client is waiting to communicate with a server, the server, ignoring possible indefinite delays whilst waiting to communicate *as a client* with another process, must eventually become ready to participate in that communication.

Unfortunately this is too vague a condition to translate into a local specification of CSP processes. For instance consider a process that acts only as a server. In order to establish the above property we would need to show that any particular client request would eventually be granted. To do this we would need to analyse the way in which the process interacts with each of its other clients and possibly also how they interact with each of their clients in turn. Here we introduce a formal definition which is slightly more restrictive than the informal one, yet with the distinct advantage of being purely a property of an individual process, independent of any network in which it is embedded. This is more suitable for collaborative software projects and makes automatic checking much more feasible.

A *basic client-server* CSP process P has a finite set of external channels partitioned into *bundles*, each of which has a type, in relation to P, which is either *client* or *server*. We write the set of client bundles of P as $clients(P)$, and the server bundles as $servers(P)$.

Each channel bundle consists of *either* a pair of channels, a *request* and an *acknowledgement*, $\langle r, a \rangle$, *or* a single channel (which we call a *drip*) $\langle d \rangle$. This allows client-server conversations to be either one way or two way. In principle there is no reason why a bundle should not contain three or more channels. In practice we have found that two channels are generally sufficient, and have restricted the definition as such.

In the subsequent analysis, a communication event on a channel is represented purely by the channel name, ignoring any data that is passed. The purpose of this is clarity and simplicity. Following this convention, a basic client-server process, P, must obey the following rules:

(a) P is divergence-free, deadlock-free and non-terminating – it can never refuse to perform every event in its alphabet, *i.e.*

$$\forall (s, X) : failures(P). \quad X \neq \alpha P$$

(b) When P is in a stable state, *either* it is ready to communicate on all its *request* and *drip* server channels *or* it is ready to communicate on none of them. In CSP terms, this means that the refusal sets in the *maximal failures* of P include either *none* of its request and drip server channels or *all* of them, *i.e.*

$$\forall (s, X) : maximal\_failures(P).$$

$$\left(\begin{array}{ccc} (\forall \langle d \rangle : servers(P). & d \notin X) & \wedge \\ (\forall \langle r, a \rangle : servers(P). & r \notin X) & \end{array} \right) \vee \left(\begin{array}{ccc} (\forall \langle d \rangle : servers(P). & d \in X) & \wedge \\ (\forall \langle r, a \rangle : servers(P). & r \in X) & \end{array} \right)$$

(c) P always communicates on any request-acknowledgement bundle pair $\langle r, a \rangle$, in alternating sequence $r, a, r, a, \ldots$, *i.e.*

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

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

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

The formal statement of these rules is rather daunting, but it should be possible to design conformant processes using only the informal descriptions which accompany them. Work is in progress to develop software engineering tools (see Section 7) that will check for adherence to the formal model, for total assurance.

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

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

Theorem 1 (Client-Server Theorem) *A client-server network, composed from basic client-server processes, which has a circuit-free*[1] *client-server digraph, is deadlock-free.*

Proof. First we observe that the matching requirements for client and server bundles within a network enforce triple-disjointedness within a client-server network. This, coupled with rule **(a)**, ensures that client-server networks conform to the model of Roscoe and Dathi.

Then we use induction on the number of processes in the network. Suppose that the theorem holds for any network with N processes. Consider a client-server network $V = \langle P_1, \ldots, P_{N+1} \rangle$, which has a circuit-free client-server digraph, yet has a deadlock state σ. Without loss of generality we assume that P_{N+1} is maximal in the client-server ordering of V. Every process is blocked in state σ, and therefore is making an ungranted request to another process. Because of its maximality P_{N+1} can only make ungranted requests on *server* channels. By rules **(c)** and **(d)** these cannot be *acknowledge* channels – they can only be *request* or *drip* channels. By rule **(b)** it follows that P_{N+1} is making an ungranted request on every one of its request and drip channels. So there is no process in the subnetwork $V' = \langle P_1, \ldots, P_N \rangle$ that is ready to communicate with P_{N+1} in state σ. Each process in V' is in fact blocked only by other processes in V', which implies that the subnetwork is itself deadlocked. This contradicts our hypothesis that the theorem holds for networks of size N. It follows that V must be deadlock-free. Clearly the theorem holds in the case when $N = 1$, so by induction it holds for all N. *Q.E.D.*

3.2 Examples

A Simple Process Farm

We consider an application where computing-intensive tasks are performed in parallel using a standard *farm* network configuration. A farmer employs n foremen each of whom is responsible for m workers. The program is coded in occam2.1[2] as shown in Figure 2.

When each `WORKER` (i, j) becomes idle, it reports the result of any work done to its `FOREMAN` (i) using channel `a[i][j]`. `FOREMAN` (i) reports this on channel `c[i]` to the `FARMER` who, in turn, replies with a new task using channel `d[i]`. `FOREMAN` (i) then assigns

[1] A *circuit* in a digraph is a sequence of distinct vertices $\langle v_0, v_1, \ldots, v_n \rangle$ such that each pair of vertices (v_i, v_{i+1}) is connected by an arc, and also (v_n, v_0). A digraph containing no circuits is said to be *circuit-free*.

[2] To make the examples more readable we take advantage of the support for user-defined data types in the recently released occam2.1 language. These types, of course, have no impact on deadlock properties.

```
...  type and constant declarations (TASK, ANSWER, n and m)

PROC FARMER ([n]CHAN OF ANSWER c, [n]CHAN OF TASK d)
  TASK task:
  ANSWER answer:
  WHILE TRUE
    ALT i = 0 FOR n
      c[i] ? answer              -- server (request)
        SEQ
          ...  set up new task
          d[i] ! task            -- server (acknowledge)
:

PROC FOREMAN ([m]CHAN OF ANSWER a.i, [m]CHAN OF TASK b.i,
              CHAN OF ANSWER c.i, CHAN OF TASK d.i)
  TASK task:
  ANSWER answer:
  WHILE TRUE
    ALT j = 0 FOR m
      a.i[j] ? answer            -- server (request)
        SEQ
          c.i ! answer           -- client (request)
          d.i ? task             -- client (acknowledge)
          b.i[j] ! task          -- server (acknowledge)
:

PROC WORKER (CHAN OF ANSWER a.i.j, CHAN OF TASK b.i.j)
  TASK task:
  ANSWER answer:
  SEQ
    ...  set initial value for task
    WHILE TRUE
      SEQ
        ...  work out answer
        a.i.j ! answer           -- client (request)
        b.i.j ? task             -- client (acknowledge)
:

[n][m]CHAN OF ANSWER a:          -- construct the network
[n][m]CHAN OF TASK b:
[n]CHAN OF ANSWER c:
[n]CHAN OF TASK d:

PAR
  FARMER (c, d)
  PAR i = 0 FOR n
    PAR
      FOREMAN (a[i], b[i], c[i], d[i])
      PAR j = 0 FOR m
        WORKER (a[i][j], b[i][j])
```

Figure 2. Farm Network

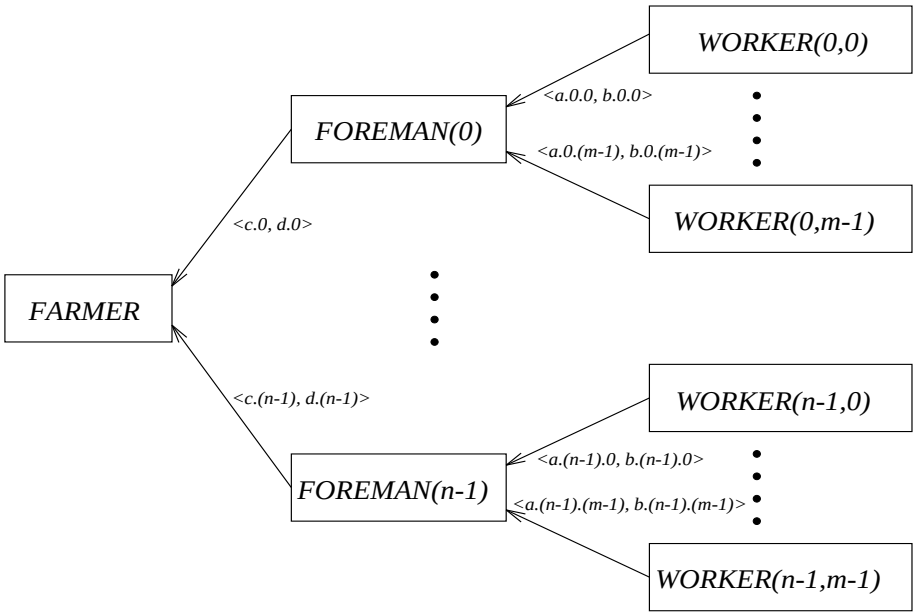

Figure 3. Client-Server Digraph for *FARM*

the new task to `WORKER` (i, j) with channel `b[i][j]`. Here the relationships between each `WORKER` and its `FOREMAN` and between each `FOREMAN` and the `FARMER` are all *client-to-server* (see Figure 3).

The CSP communication patterns for the component processes may be extracted as follows:

$$
\begin{aligned}
FARMER &= \Box_{i=0}^{n-1}\, c.i \rightarrow d.i \rightarrow FARMER \\
clients(FARMER) &= \{\} \\
servers(FARMER) &= \{\langle c.0, d.0\rangle, \ldots, \langle c.(n-1), d.(n-1)\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
FOREMAN(i) &= \Box_{j=0}^{m-1}\, a.i.j \rightarrow c.i \rightarrow d.i \rightarrow b.i.j \rightarrow FOREMAN(i) \\
clients(FOREMAN(i)) &= \{\langle c.i, d.i\rangle\} \\
servers(FOREMAN(i)) &= \{\langle a.i.0, b.i.0\rangle, \ldots, \langle a.i.(m-1), b.i.(m-1)\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
WORKER(i,j) &= a.i.j \rightarrow b.i.j \rightarrow WORKER(i,j) \\
clients(WORKER(i,j)) &= \{\langle a.i.j, b.i.j\rangle\} \\
servers(WORKER(i,j)) &= \{\}
\end{aligned}
$$

It is straightforward to see that each process obeys the basic client-server protocol. Clearly no individual process would ever terminate, deadlock or diverge if run in isolation (rule (a)). Whenever a *FARMER* or *FOREMAN* process attempts to communicate on a

server request channel it offers the complete choice of such channels to its environment (rule (b)). The *WORKER* processes are pure clients so rule (b) does not apply to them. Each process communicates on each of its client request-acknowledgement pairs in alternating sequence (rule (c)). And whenever a *WORKER* or *FOREMAN* process communicates on a client request channel it immediately becomes ready to communicate on the corresponding acknowledgement channel, and remains so until this communication takes place (rule (d)). The *FARMER* is a pure server process for which rule (d) does not apply.

The client-server digraph of this network is illustrated in figure 3. It has no circuits and, hence, the network is guaranteed deadlock-free.

It is usually safe to represent communication events in occam2 purely by their channel names in the CSP specification, as was done here. The one exception is when using a *variant protocol* on a particular channel. If the inputting process is unwilling to accept the type of datum offered by the outputting process, a local deadlock may ensue. However, if an exhaustive case list is offered by the inputting process there is no problem – but this may be impractical. This issue is discussed in more detail in [8].

Note that the client-server ordering does not imply a single direction of dataflow. A client-server bundle may contain both an input channel and an output channel.

A Message Router

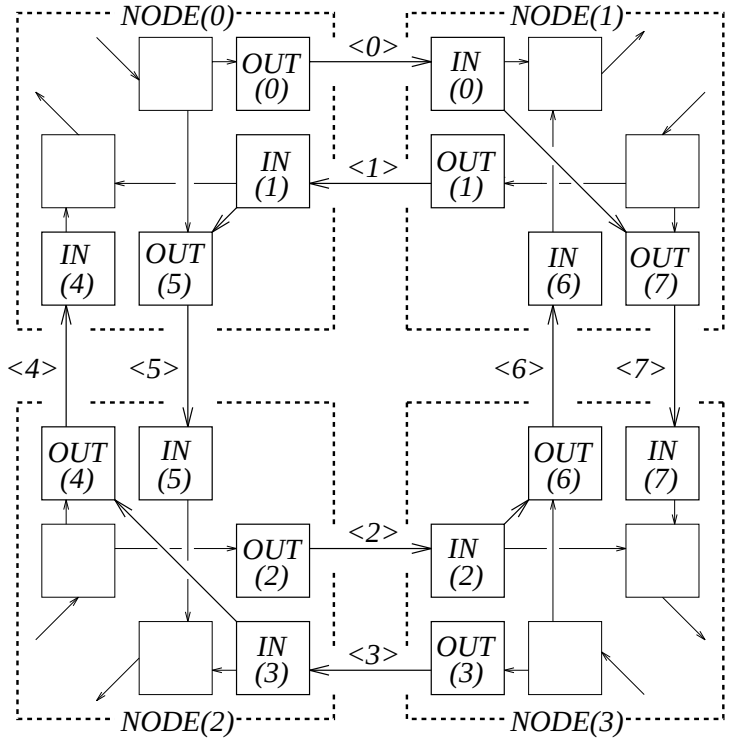

Figure 4. Client-Server Digraph for the Message Router

Figure 4 shows the client-server digraph of a message routing program for a network of processors. The processor topology illustrated is a square. Each processor is connected to its horizontal and vertical neighbours by a link in each direction. Each processor runs a separate process to control each of its input and output links. It also runs two interface processes. One of these collects messages which have arrived at their destination, and passes them to the local application program. The other routes messages from the local application destined for other processors. The following routing strategy is employed. In order to send a message to a processor with grid coordinates (x, y), first move it to a node with the right x coordinate and then, keeping the x coordinate fixed, move it to the right y coordinate. This strategy is easily generalised to an n dimensional grid – see [9].

In this case, each client-server bundle consists of a single *drip* channel, which makes it particularly easy to verify adherence to the protocol – rules **(c)** and **(d)** not being relevant. In fact each arrow in the client-server digraph corresponds to dataflow (by chance). Although the *processor* configuration contains circuits, the client-server digraph of the *processes* contains none, so the router is deadlock-free.

Note: The fact that the routing program is itself deadlock-free does not guarantee that any program that incorporates it will not deadlock. Great care still needs to be taken [10].

4 COMPOSITE PROCESSES

A *composite* client-server process V is a client-server network $\langle P_1, \ldots, P_N \rangle$ composed solely from *basic client-server* processes and whose client-server digraph contains no circuits. We define:

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

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

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

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

Theorem 2 (Client-Server Closure) *A client-server network, composed from composite client-server processes, which has a circuit-free client-server digraph, is deadlock-free.*

Proof. Consider the total client-server digraph formed when each composite process is expanded back into its basic components. Since there are no cycles in the sub-digraphs contributed by the composite processes and there is no cycle in the top-level digraph, there can be no cycles in the total digraph. Theorem 1 allows us to conclude that the composite network is deadlock-free. *Q.E.D.*

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

However this theorem is too weak in some circumstances. We need to find a generalisation.

We define a dependency relationship $\gg$ between the server bundles and client bundles of a composite process V. If $x \in \mathit{servers}(V)$ and $y \in \mathit{clients}(V)$ then $x \gg y$ means that there is a path from the process with server bundle x to that with client bundle y, in the client-server digraph of V.

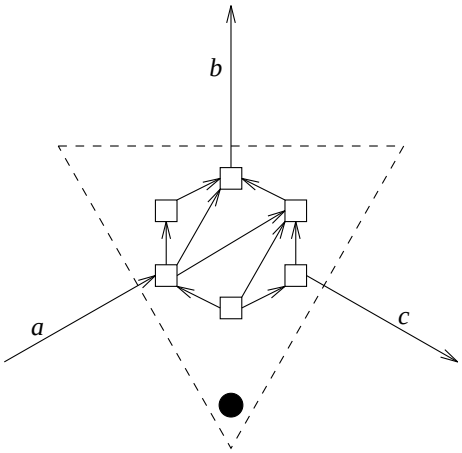

Figure 5. Composite Client-Server Process

For example, figure 5 shows a composite client-server process *TRIANGLE*, with external client-server channel bundles a, b, and c. Here we have:

$$\begin{aligned} \mathit{servers}(\mathit{TRIANGLE}) &= \{a\} \\ \mathit{clients}(\mathit{TRIANGLE}) &= \{b, c\} \end{aligned}$$

but:

$$(a \gg b) \quad \mathit{and} \quad \neg\,(a \gg c)$$

We construct an *exploded* client-server digraph of a network containing composite processes in the following way. A node representing any composite process V is removed. In its stead is placed a set of nodes – one for each server or client bundle of V. Each of these is joined to its corresponding arc (that was formerly incident to the node representing V). Then we draw an arc from the node representing (server) bundle i to that representing (client) bundle j if, and only if, $i \gg j$ in V.

Theorem 3 (Exploded Client-Server Closure) *A client-server network, composed from composite client-server processes, which has a circuit-free exploded client-server digraph, is deadlock-free.*

Proof. As in Theorem 2, consider the *total* client-server digraph formed when each composite process is expanded back into its basic components. As before, there are no cycles in the sub-digraphs contributed by the composite processes. Therefore, if there were a cycle in the *total* digraph, the construction rules for the *exploded* digraph mean that the latter would also contain a cycle. Since the *exploded* digraph has no cycles, neither does the *total* digraph and, by Theorem 1, the whole client-server network is deadlock-free. *Q.E.D.*

Figure 6 displays two representations of a network constructed from six copies of *TRIANGLE* (with suitably relabelled channels): the client-server digraph, and an exploded client-server digraph. The former contains a circuit, so we cannot use theorem 2 to show that the network is deadlock-free. However the latter contains none. So the network *is* deadlock-free by theorem 3.

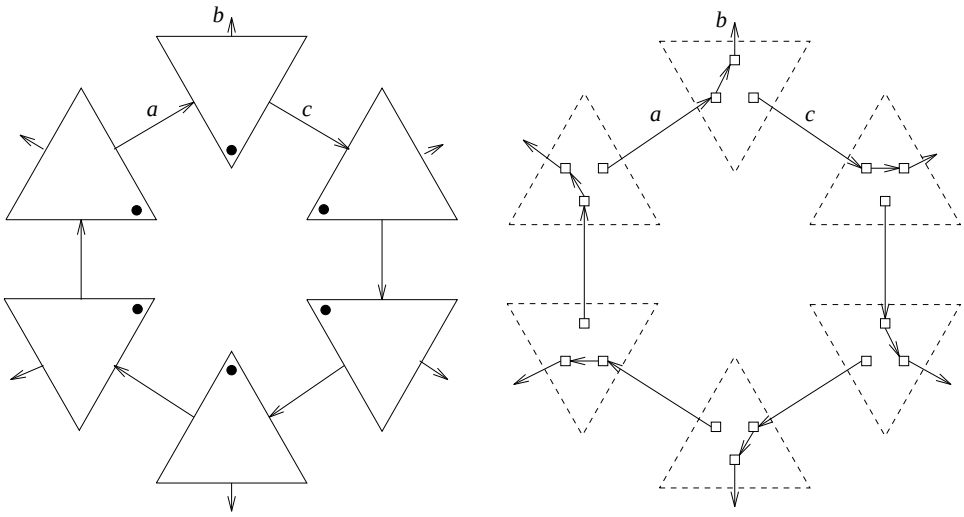

Figure 6. Client-Server Digraph and Exploded Client-Server Digraph

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

Therefore for deadlock-free design with composite client-server processes, we only need to be told their client bundles, server bundles, and the dependency relation $\gg$ between them. The benefit of Theorems 2 and 3 is that we avoid repeating superfluous information in the diagrams we draw to design our programs. Instances of complex subnetworks are reduced *either* to single nodes *or* to simplified representations when theorem 2 is too weak.

5 ADDING A CLIENT-SERVER INTERFACE TO AN ARBITRARY NETWORK

5.1 Hybrid Networks

Theorems 2 and 3 make available a hierarchical approach to deadlock-free software construction, based on multiple layers of the client-server model.

However client-server mechanisms are not appropriate for all types of system or subsystem. It would be nice to be able use other paradigms to design deadlock-free subnetworks (such as described in [2] and [3]), and then wrap them up with a client-server interface for inclusion in a wider context.

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

$$V' = \langle P_1', \ldots, P_n' \rangle$$

and where each process P_i' performs events in the alphabet of P_i as well as (possibly) additional events that are external to the network, *i.e.*

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

The basic rule of thumb is that we may freely add *client* connections to any component process P_i, but we may add *server* connections to at most one such process. The following rules must be obeyed:

1. The additional channels of each process P_i' are partitioned into client and server bundles, and P_i' must obey the basic client-server protocol on these bundles.
2. It is not permitted for more than one process P_i' to have *server* connections.
3. It is necessary to ensure that the new connections added to each process P_i do not interfere with its internal behaviour. If communication on those channels were concealed, P_i' should behave identically to P_i, *i.e.*

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

The client-server bundles of V' are taken to be the disjoint union of those of each component. It is clear that V' will adhere to rules **(c)** and **(d)** of the basic client-server protocol (see section 3), since Rule 1 stipluates that each process P_i' does. Rule 2 ensures that V' obeys rule **(b)** of the basic protocol. Rule 3 guarantees that V' is deadlock-free, divergence-free and non-terminating – rule **(a)** of the basic client-server protocol. Hence, V' may now be treated as a *basic client-server* process for the construction of client-server networks that are automatically deadlock-free (provided the construction obeys the premise of one of the theorems in this paper).

Rule 3 needs a little further explanation in the context of actual programming. In occam2 it means that if each communication on a client or server channel by P_i' were replaced by `SKIP`, then P_i' would follow an identical communication pattern to P_i.

5.2 Polling on a Server Channel

The technique of *polling* on a channel is a means by which a process can attempt to communicate on a channel without the risk of becoming blocked. If the communication fails within a certain time, the process gives up and does something else. In occam2 polling is usually performed using a `PRI ALT` construct which has either `SKIP` or timer input alternatives.

While a process is attempting to poll a channel its state is unstable. If the processes which constitute a network only ever communicate on external server request and drip channels by *polling*, then rule 2 for adding a client-server interface to a network may be overlooked. In this way we may add external server connections to as many processes as we wish in a network, and then safely embed it within a client-server system.

5.3 Example – Bunjee Jump Simulation

We consider an occam2 program for modelling the action of an elastic rope in the simulation of a "bunjee-jump". The rope is modelled by N copies of a process `ROPE`, each simulating the motion of a section of the rope. These processes interact with their neighbours according to the *I/O-PAR* paradigm [3]. This is a very simple design rule which runs as follows. In any network where each process behaves cyclically and communicates on each of its channels exactly once on each cycle *in parallel*, deadlock can never occur.

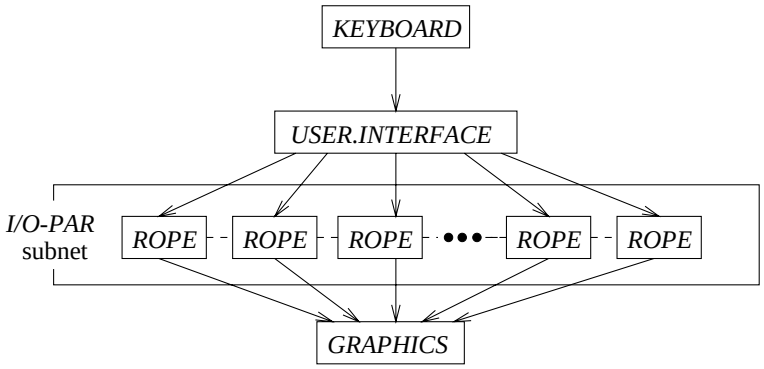

Figure 7. Client-Server Digraph for the Bunjee-Jump Simulation

In order that we may visualise this simulation, a graphics handling process, `GRAPHICS`, is added to the *I/O-PAR* network using client-server connections. This process acts as a server to each `ROPE` process along a single input channel. In this way, the position of each rope section is periodically communicated to the graphics handler.

We also add a process `USER.INTERFACE` which communicates with each of the rope elements as a client. To each rope process is added a corresponding *polled* server connection. This mechanism is used to allow interactive keyboard control of the simulation. `USER.INTERFACE` also has a server connection to a keyboard interface process `KEYBOARD`.

The client and server channels which were added to each component in the *I/O-PAR* subnetwork of rope elements do not effect its internal communication pattern. Although it has multiple disjoint server connections, they are only initiated by polling and so are

```
...  declaration of constants (n = the number of ROPE sections)
...  type declarations (COORDS, PARAMS, ACK and DISPLAY)

PROC ROPE (VAL INT i,
           CHAN OF COORDS to.left, from.left,     -- I/O-PAR
           CHAN OF COORDS to.right, from.right,   -- I/O-PAR
           CHAN OF PARAMS new.parameters,         -- server
           CHAN OF ACK acknowledge,               -- server
           CHAN OF DISPLAY display)               -- client
  ...  local declarations
  SEQ
    ...  initialisation of local state
    WHILE TRUE
      SEQ
        --{{{  service new.parameters/acknowledge bundle (polling)
        PRI ALT
          new.parameters ? parameters
            SEQ
              ...  update local computation parameters
              acknowledge ! ack
          TRUE & SKIP
            SKIP
        --}}}
        --{{{  communicate with neighbours (I/O-PAR)
        PAR
          IF
            i <> 0        -- check not first section of rope
              PAR
                from.left ? x.y.left
                to.left ! x.y.start
            TRUE
              SKIP
          IF
            i <> (n-1)    -- check not last section of rope
              PAR
                from.right ? x.y.right
                to.right ! x.y.finish
            TRUE
              SKIP
        --}}}
        ...  compute new state for this section of rope
        --{{{  report state to display (client transaction)
        IF
          display.cycle
            display ! rope.section
          TRUE
            SKIP
        --}}}
:
```

Figure 8. Bunjee-Jump (ROPE process)

```
...  declaration of KEYBOARD process

PROC USER.INTERFACE (CHAN OF BYTE keyboard,            -- server
                     [n]CHAN OF PARAMS new.parameters, -- client
                     [n]CHAN OF ACK acknowledge)       -- client
  ...  local declarations
  WHILE TRUE
    SEQ
      keyboard ? character
      ...  set up new parameters
      PAR i = 0 FOR n
        SEQ
          new.parameters[i] ! parameters[i]
          acknowledge[i] ? ack[i]
:

PROC GRAPHICS ([n]CHAN OF DISPLAY display)             -- server
  ...  local declarations
  WHILE TRUE
    ALT i = 0 FOR n
      display[i] ? rope.section
        ...  update display
:

-- construct the network

CHAN OF BYTE keyboard:
[n]CHAN OF PARAMS new.parameters:
[n]CHAN OF ACK acknowledge:
[n+1]CHAN OF COORDS to.left, to.right:
[n]CHAN OF DISPLAY display:

PAR
  KEYBOARD (keyboard)
  USER.INTERFACE (keyboard, new.parameters, acknowledge)
  PAR i = 0 FOR n
    ROPE (i, to.left[i], to.right[i],
          to.right[i+1], to.left[i+1],
          new.parameters[i], acknowledge[i],
          display[i])
  GRAPHICS (display)
```

Figure 9. Bunjee-Jump (rest of the network)

non-blocking. It follows that the subnetwork formed from the ROPE processes behaves as a single client-server process in relation to its environment. The client-server digraph for the complete system is given in figure 7 – note that the *I/O-PAR* subnetwork should be regarded as a single vertex. This digraph is free of circuits so the system is deadlock free.

Skeleton code for the resulting network is given in figures 8 and 9. The high-valency and irregular communication topology of this program is not a problem for a network of transputers. For first generation (T2/T4/T8) architectures, the Southampton Virtual Channel Router[10] is part of the standard Toolset. For T9000s, the same idea is implemented by hardware in the Virtual Channel Processor. These mechanisms remove the need for communication design to be constrained to 4-valent topologies. This remains true for occam retargetings to non-transputer multi-processors.

6 RELATED ISSUES

Apart from deadlock, there are two other major forms of pathological behaviour that networks of communicating processes may exhibit: *livelock* and *starvation*.

Livelock is said to occur if the network becomes locked in an everlasting sequence of internal communications, without ever communicating with its environment. It is a particular form of divergence that occurs when internal communications are concealed. This is a problem which may be every bit as catastrophic as deadlock. Fortunately it is a relatively easy one to solve in client-server networks. If every component of a client-server network has server connections and will always communicate on a server channel within a finite number of actions, the network as a whole will be livelock-free – this is easily proved by induction.

Starvation occurs to a particular process within a parallel network when it is held in a perpetual state of waiting to communicate with another process or set of processes. In a client-server network this might occur when two clients are competing for use of the same server, but one of them is always favoured. In a message routing system, this would mean that a message might never be delivered in heavy traffic. There is no way to deal with this possibility of unfairness in the standard CSP model. However it can be tackled in occam2 with careful use of the `PRI ALT` construct [11].

7 AREAS OF ONGOING RESEARCH

This paper has described a methodology for modular parallel programming. We have shown how to develop reusable components that may be incorporated into complex networks guaranteeing deadlock-freedom. The communication interface to third-party software components is specified by their client and server channel bundles. An assurance is also required that any such component adheres to principles outlined above, either as a basic client-server process or a composite client-server process.

A software engineering tool has been developed to check this automatically[12]. This is based on the FDR tool of Formal Systems (Europe) Ltd. [13] which is used for checking refinement in CSP. The deadlock-checking program is presented with a network of CSP processes. Each of these is then compiled into a compact normal form representation (if possible). It is then analysed for client-server conformance using the model-checking techniques of [14]. Work is in progress to improve the power and performance of this tool.

A program has been developed by Formal Systems (Europe) Ltd. which extracts the communication pattern from occam2 programs into CSP format [8]. This means that they too could be checked automatically. It would then be easy to verify that a third party component is correctly implemented, according to the client-server model. The same could be done for other CSP-based languages.

The main advantage of this will be as follows. Grappling with CSP specifications will not appeal to all programmers. Many will prefer to work from the informal statement of the client-server rules using a high-level language directly. The model checker could be used to provide total assurance that their code conforms to the formal requirements.

A different tool, based on the ideas in [2,3], is described in [15]. This allows designers to work just with graphical representations of the system data-flow and to overlay this with client-server digraphs. Currently, the tool checks designs against Theorems 1 and 2 (and a similar result concerning I/O-PAR networks) to ensure that only deadlock-free systems can be created. For each *basic* process, the tool generates template occam (serial) code with a communication structure that satisfies Rules **a**, **b**, **c** and **d** from Section 3. Code for state declaration, initialisation and maintenance has to be added but, so long as no infinite computational loops are introduced, this has no impact on these rules. For *higher-level* components (e.g. those implemented by client-server sub-networks), the full occam parallel code is produced.

Work is in progress to extend this tool to accept the dependency relationship $\gg$ (Section 3), to check designs against the more flexible Theorem 3 and to support the use of hybrid paradigms (as defined in Section 5). The attraction of tools such as this is that deadlock-free designs can be constructed graphically (top-down, bottom-up or both) without the designer needing fluency in either CSP or occam, and with all the parallel and communication servicing code generated automatically. The tool knows the theorems and the designer learns from the tool.

REFERENCES

[1] SGS-Thomson Microelectronics. *occam2.1 Reference Manual*, 1995. Also available from <URL:http://www.hensa.ac.uk/parallel/occam/documents/>.

[2] J. Martin, I. East, and S. Jassim. Design rules for deadlock freedom. *Transputer Communications*, 2(3):121–133, September 1994. ISSN 1070-454X.

[3] P.H. Welch, G.R.R. Justo, and C. Willcock. High-level paradigms for deadlock-free high-performance systems. In Grebe et al., editors, *Transputer Applications and Systems '93*, pages 981–1004, Amsterdam, 1993. IOS Press. ISBN 90-5199-140-1.

[4] A.W. Roscoe and N. Dathi. The pursuit of deadlock freedom. Technical Report Technical Monograph PRG-57, Oxford University Computing Laboratory, 1986.

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

[6] N. Dathi. *Deadlock and Deadlock-Freedom*. D.Phil thesis, Oxford University Computing Laboratory, 1991.

[7] P. Brinch Hansen. *Operating System Principles*. Prentice Hall, 1973.

[8] B. Scattergood and K. Seidel. Translating occam2 into csp. In *Transputer Applications and Systems '94*, pages 416–430, Amsterdam, September 1994. IOS Press. ISBN 90-5199-177-0.

[9] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5), May 1987.

[10] M.Debbage, M.B.Hill, and D.A.Nicole. Global communications on locally connected message-passing parallel computers. *Concurrency: Practice and Experience*, 5(6):491–509, Sept. 1993.

[11] G. Jones. Carefully scheduled selection with ALT. *OUG Newsletter*, 10:17–23, 1989.

[12] J.M.R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. D.Phil thesis, University of Buckingham, 1996. Also available from <URL:http://www.hensa.ac.uk/parallel/ theory/formal/csp/>.

[13] Formal Systems (Europe) Ltd. FDR user manual and tutorial. Technical Report Version 1.4, 3 Alfred Street, Oxford OX1 4EH., 1994. (Email: `enquiries@fsel.com`).

[14] A.W. Roscoe. *Model Checking* CSP, *A Classical Mind*. Prentice Hall, 1994.

[15] D.J. Beckett and P.H. Welch. A strict occam design tool. In *Proceedings of UK Parallel '96*, pages 53–69, London, July 1996. BCS PPSIG, Springer-Verlag. ISBN 3-540-76068-7.