How to Design Deadlock-Free Networks Using CSP and Verification Tools – A Tutorial Introduction

J.M.R.Martin
Oxford University Computing Services,
13 Banbury Road,
Oxford 0X2 6NN, UK

S.A.Jassim
Department of Mathematics,
Statistics, and Computer Science,
University of Buckingham,
MK18 1EG, UK

Abstract. The CSP language of C.A.R.Hoare originated as a blackboard mathematical notation for specifying and reasoning about parallel and distributed systems. More recently sophisticated tools have emerged which provide automated verification of CSP-specified systems. This has led to a tightening and standardisation of syntax. This paper outlines the syntax and semantics of CSP as it is now used and then describes how to design CSP networks, which are guaranteed to be free of deadlock, through a succession of increasingly complex worked examples, making use of the verification tool Deadlock Checker.

1 Introduction

CSP, which stands for Communicating Sequential Processes, was introduced by C.A.R.Hoare in 1978[3]. By the time of publication of his textbook in 1985[4] the language had evolved significantly. Since then, however, it has remained virtually stable[1]. CSP provides an excellent means of describing and reasoning about complex communication patterns, for the following reasons.

- It encapsulates the fundamental principles of communication in a simple and elegant manner.
- It is semantically defined in terms of a structured mathematical model which may be used to deduce system properties rigorously.
- It is sufficiently expressive to enable reasoning about the pathological problems of deadlock and livelock.
- The principles of abstraction and refinement are central to the underlying theory.
- Robust software engineering tools exist for formal verification in CSP[1, 5].
- Certain mainstream programming languages, such as `occam`, Ada and some dialects of parallel C are derived directly from the CSP model. CSP-style communications libraries are available for other languages, such as Java.

In the next section we shall describe briefly the syntax and semantics of CSP. This will be followed by a series of worked examples of designing complex systems that tackle the specific problem of guaranteeing freedom from deadlock, a state which is potentially catastrophic in a safety-critical system. We shall make use of design rules developed for this purpose and an automatic CSP verification tool.

[1]The only significant change has been to drop the requirement for every process to have an alphabet of possible events defined. Alphabets are now used instead as an integral part of the parallel composition operator.

2 The CSP Language

Basic Syntax and Informal Description

The basic syntax of CSP is described by the following grammar

$$
\begin{aligned}
Process \;:== \;& STOP \;\Big|\\
& SKIP \;\Big|\\
& event \rightarrow Process \;\Big|\\
& Process \,;\, Process \;\Big|\\
& Process \;[\![\, alph \mid alph \,]\!]\; Process \;\Big|\\
& Process \;|||\; Process \;\Big|\\
& Process \sqcap Process \;\Big|\\
& Process \;\Box\; Process \;\Big|\\
& Process \setminus event \;\Big|\\
& f(Process) \;\Big|\\
& name
\end{aligned}
$$

Here *event* ranges over a universal set of events, Σ, *alph* ranges over subsets of Σ, f ranges over a set of function names, and *name* ranges over a set of process names.

A process describes the behaviour of an object in terms of the events in which it may engage. The simplest process of all is *STOP*. This is the process which represents a deadlocked object. It never engages in any event. Another primitive process is *SKIP* which does nothing but terminate successfully; it only performs the special event $\surd$, which represents successful termination.

An event may be combined with a process using the prefix operator, written $\rightarrow$. The process $bang \rightarrow UNIVERSE$ describes an object which first engages in event *bang* then behaves according to process *UNIVERSE*. If we want to give this new process the name *CREATION* we write this as an equation

$$CREATION = bang \rightarrow UNIVERSE$$

Processes may be defined in terms of themselves using the principle of recursion. Consider a process to describe the ticking of an everlasting clock.

$$CLOCK = tick \rightarrow CLOCK$$

CLOCK is a process which performs event *tick* and then starts again. (This is a somewhat abstract definition. No information is given as to the duration or frequency of ticks. We are simply told that the clock will keep on ticking.)

The recursive notation is commonly extended to a set of simultaneous equations where a number of processes are defined in terms of each other. This is known as mutual recursion, examples of which will be found in later sections.

There are a number of CSP operations which combine two processes to produce a new one. The first of these that we shall consider is sequential composition.

$$UNIVERSE = EXPAND \,;\, CONTRACT$$

is the process which first behaves like *EXPAND*, but when *EXPAND* is ready to terminate it continues by behaving like *CONTRACT*. However it may also be possible that *EXPAND* will never terminate.

It is rather more complicated to compose two processes in parallel than in sequence. It is necessary to specify a set of events for each process, known as its *alphabet*. The process denoted

$$PANTOMIMEHORSE = \quad FRONT \;|[\, \{forward, backward, nod\} \mid \{forward, backward, wag\} \,]|\; BACK$$

represents the parallel composition of two processes: *FRONT* with alphabet $\{forward,\ backward, nod\}$ and *BACK* with alphabet $\{forward, backward, wag\}$[2]. Here each process behaves according to its own definition, but with the constraint that events which are in the alphabet of both *FRONT* and *BACK*, *i.e.* *forward* and *backward*, require their simultaneous participation. However they may progress independently on those events belonging solely to their own alphabet. If a situation were to arise where *FRONT* could only perform event *forward* and *BACK* could only perform event *backward* then deadlock would have occurred.

Parallel composition may be extended to three or more processes; given a sequence of processes $V = \langle P_1, .., P_N \rangle$ with corresponding alphabets $\langle A_1, .. A_N \rangle$ we write their parallel composition as

$$PAR(V) \quad = \quad \|_{i=1}^{n} \ (P_i, A_i)$$

Note that it is implicitly assumed that the termination event $\surd$ requires the joint participation of each process P_i, whether or not it is included in their process alphabets.

An alternative form of parallel composition is *interleaving*, where there is no communication between the component processes. In the parallel combination

$$BRAIN \,|||\, MOUTH$$

the two processes, *BRAIN* and *MOUTH*, progress independently of each other and no cooperation is required on any event, except for $\surd$, the termination event. Any other actions which are possible for both processes will only be performed by one process at a time. Interleaving is a commutative and associative operation and so we may extend the notation to various indexed forms, such as

$$|||_{i=1}^{n} \ P_n, \quad |||_{x:X} \ P_x$$

A useful feature of CSP is the ability to describe *nondeterministic* behaviour, which is where a process may operate in an unpredictable manner. The process

$$BUFFER = TWOPLACE \sqcap THREEPLACE$$

may behave either like process *TWOPLACE* or like process *THREEPLACE*, but there is no way of telling which in advance. The purpose of the $\sqcap$ operator is to specify concurrent systems in an abstract manner. At the design stage, there is no reason to provide any more detail than is necessary and, where possible, implementation decisions should be deferred until later.

This operation is known as *internal choice*. CSP also contains an *external choice* operator $\Box$ which enables the future behaviour of a process to be controlled by other processes running along side it in parallel, which, collectively, we call its *environment*.

The process

$$MW = DEFROST \,\Box\, COOK$$

may behave like *DEFROST* or like *COOK*. Its behaviour may be controlled by its environment provided that this control is exercised on the very first event. If an initial event *button1* is offered by *DEFROST* that is not an initial event of *COOK*, then the environment may coerce *MW* into behaving like *DEFROST*, by performing *button1* as its initial event. If, however, the environment were to offer an initial event that is allowed by both *DEFROST* and *COOK* then the choice between them would be nondeterministic.

Both the choice operators may be extended to indexed forms. We write

$$\Box_{x:A} \ x \rightarrow P_x$$

[2] A *pantomime* is a traditional British theatrical entertainment which often features a "horse" consisting of two actors in a single costume, one of whom plays the front legs and head while the other plays the hind legs and tail.

to represent the behaviour of an object which offers any event of a set A to its environment. Once some initial event x has been performed the future behaviour of the object is described by the process P_x. However, the process

$$\sqcap_{x:A}\ x \rightarrow P_x$$

(where, for technical reasons, A must be finite) offers exactly one event x from A to its environment, the choice being non-deterministic.

Sometimes it is useful to be able to restrict the definition of a process to a subset of relevant events that it performs. This is done using the hiding operator ($\backslash$). The process

$$CREATION \setminus bang$$

behaves like *CREATION*, except that each occurrence of event *bang* is concealed. Note that it is not permitted to hide event $\surd$.

Concealment may introduce nondeterminism into deterministic processes. It may also introduce the phenomenon of *divergence*. This is a drastic situation where a process performs an endless series of hidden actions. Consider, for instance, the process

$$CLOCK \setminus tick$$

which is clearly a divergent process.

It is conventional to extend the notation to $P \setminus A$, where A is a finite set of events.

Finally let us briefly consider process relabelling. Let f be an *alphabet transformation function* $f : \Sigma \rightarrow \Sigma$, which satisfies the property that only finitely many events may be mapped onto a single event. Then the process $f(P)$ can perform the event $f(e)$ whenever P can perform event e. As an example consider a function *new* which maps *tick* to *tock*. Then we have

$$new(CLOCK) = tock \rightarrow new(CLOCK)$$

Denotational Semantics

The meaning of a CSP process is defined in terms of the circumstances under which it might deadlock or diverge. This is the Failures-Divergences model.

A *trace* of a process P is any finite sequence of events that it may initially perform. A *divergence* of a process is a trace after which it might diverge. A *failure* of a process P consists of a pair (s, X) where s is a trace of P and X is a set of events which if offered to P by its environment after it has performed trace s, might be completely refused.

Each CSP process is then uniquely defined by a pair of sets (F, D), corresponding to its *failures* and *divergences*.

The *failures* and *divergences* of the fundamental CSP terms are defined by equations such as

$$\begin{array}{rcl}
divergences(STOP) & = & \{\} \\
failures(STOP) & = & \{\langle\rangle\} \times \mathbf{P}\,\Sigma \\
divergences(x \rightarrow P) & = & \{\langle x\rangle^\frown s \mid s \in divergences(P)\} \\
failures(x \rightarrow P) & = & \{(\langle\rangle, X) \mid X \subseteq \Sigma - \{x\}\} \\
& \cup & \{(\langle x\rangle^\frown s, X) \mid (s, X) \in failures(P)\}
\end{array}$$

(but there is not room for the complete set of equations here). These particular equations define the meaning of *STOP* and the event-prefix operator $\rightarrow$. First we are told that *STOP* does not diverge, but refuses to perform any event whatever set of events is offered to it. Then we are told that the divergent traces of $x \rightarrow P$ are the divergent traces of P with event x prefixed to them, and the failures of $x \rightarrow P$ to be the failures of P with x prefixed to the trace part of each failure, *together with* pairings of the empty trace with all subsets of Σ which exclude x.

This model is also used for formal reasoning about the behaviour of concurrent systems defined by CSP equations. There is a natural *partial ordering* on the set of all processes given by

$$(F_1, D_1) \sqsubseteq (F_2, D_2) \iff F_1 \supseteq F_2 \wedge D_1 \supseteq D_2$$

The interpretation of this is that process P_1 is worse than P_2 if it can deadlock or diverge whenever P_2 can[3]. This partial ordering is very important to the stepwise refinement of concurrent systems. Starting from an abstract non-deterministic definition, details of components may be independently fleshed out whilst preserving important properties of the overall system such as freedom from deadlock and divergence. The FDR tool of Formal Systems Europe[1] can automatically verify this refinement relation in the failures-divergences model.

A process P is deadlock-free if there is no trace after which it might refuse to perform any event, i.e. $\nexists s.\ \ (s, \Sigma) \in \mathit{failures}(P)$. It is divergence-free if it has an empty set of divergences. A particularly important point to stress is that when a network of CSP processes is composed in parallel it becomes a single CSP process. So these definitions apply equally well to parallel networks of processes as they do to single sequential processes, for which deadlock and divergence are not usually a problem.

Algebraic Semantics

From the failures-divergences model can be deduced a complete set of algebraic laws which govern CSP processes, for instance

$$\begin{array}{rcl}
(P\,;Q)\,;R & = & P\,;(Q\,;R) \\
(a \rightarrow P)\,;Q & = & a \rightarrow (P\,;Q) \\
P \ [\![A \mid B]\!]\ Q & = & Q \ [\![B \mid A]\!]\ P \\
P \ [\![A \mid B \cup C]\!]\ (Q \ [\![B \mid C]\!]\ R) & = & (P \ [\![A \mid B]\!]\ Q) \ [\![A \cup B \mid C]\!]\ R \\
P \,\Box\, (Q \sqcap R) & = & (P \,\Box\, Q) \sqcap (P \,\Box\, R) \\
P \sqcap (Q \,\Box\, R) & = & (P \sqcap Q) \,\Box\, (P \sqcap R) \\
P \,\Box\, \mathit{STOP} & = & P
\end{array}$$

There are many more such rules, but there is insufficient room for their inclusion here. The rules are used to derive correctness properties of CSP systems using algebraic manipulation. See [4] for more details.

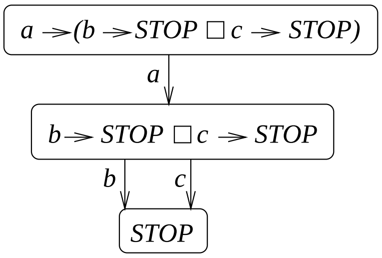
Operational Semantics

So far we have encountered two ways of looking at communicating processes: firstly as algebraic expressions and secondly in terms of abstract mathematical sets based on their observable behaviour. There is no obvious way of seeing from either of these representations how CSP might be realised on a machine. A more concrete approach is given by operational semantics. The operational semantics of CSP is a mapping from CSP expressions to *transition systems*. For example, figure 1 illustrates the transition system for the process $a \rightarrow (b \rightarrow \mathit{STOP} \,\Box\, c \rightarrow \mathit{STOP})$. (There is insufficient room to explain the mapping here but the curious reader should refer to [5] for further explanation.)

The behaviour of a process predicted by its failures and divergences will be the same as that which can be observed of its operational representation. So we may use the operational semantics of CSP in order to prove properties of process behaviour which are phrased in the Failures-Divergences model. This feature turns out to be particularly useful when the operational representation of a process is finite although its failures and divergences are infinite, as is usually the case in practice. Therefore this is the representation of processes which is used inside the various CSP verification programs, such as FDR[1] and Deadlock Checker[7].

[3] This ordering is in fact a *complete partial order*. The bottom, or worst, element $\perp$ represents the process which always diverges.

Figure 1: CSP Transition System



Language Extensions

The core CSP syntax described above is very abstract, and lacks certain useful features found in conventional sequential and parallel programming languages. The extensions outlined below are useful for writing more detailed specifications and may be defined in terms of the core constructors.

Sometimes we define processes with parameters, such as

$$BUFF(in, out) = in \rightarrow out \rightarrow BUFF(in, out)$$

This is a process-schema, rather than an actual process. It defines a CSP process for each combination of parameter values. CSP parameters may be integers, real numbers, events, sets, matrices, *etc.*

A *communication* is a special type of event described by a pair $c.v$, where c is the name of the channel on which the event takes place, and v is the value of the message that is passed.

The set of messages communicable on channel c is defined

$$type(c) = \{v | c.v \in \Sigma\}$$

Input and output are defined as follows. A process which first outputs v on channel c, then behaves like P is defined simply as

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Outputs may involve expressions of parameters such as $P(x) = c!x^2 \rightarrow Q$. The expressions are evaluated according to the appropriate laws.

A process which is initially prepared to input any value x communicable on the channel c, then behave like $P(x)$ is defined.

$$(c?x \rightarrow P(x)) = \Box_{v:type(c)} (c.v \rightarrow P(v))$$

It is usual for a communication channel to be used by at most two processes at any time: one for input and the other for output. This restriction, which is known as *triple-disjointedness*, is not enforced in the modern version of CSP but it applies to all the classes of network which are discussed in this paper.

Another important aspect to real programming languages is the use of conditionals. Let b be a boolean expression (either true or false). Then

$$P \lhd b \rhd Q \qquad (\text{“}P \text{ if } b \text{ else } Q\text{”})$$

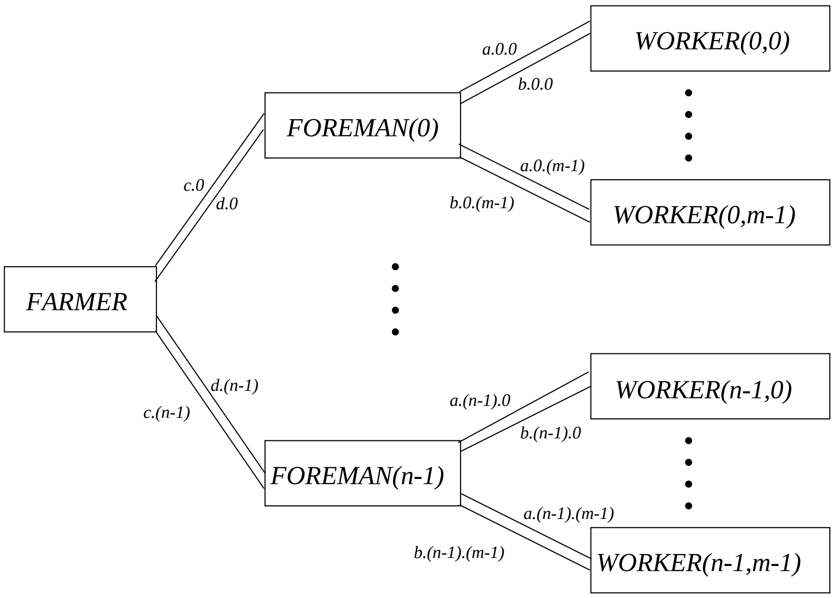
is a process which behaves like P if the value of expression b is true, or like Q otherwise.

These extensions are especially useful for specifying fine detail during the later stages of program refinement. At the design stage we shall tend to stick to more abstract, non-deterministic definitions of processes. The deadlock issue will usually be addressed at this point. In this way we can build robust programs for which deadlock-freedom cannot be compromised by implementation decisions made at a later stage.

3 Example 1 – A Client Server System

We consider an application where computing-intensive tasks are performed in parallel using a standard farm network configuration. A farmer employs n foremen each of whom is responsible for m workers. When a worker process becomes idle it reports the result of any work done to its foreman, using channel $a.i.j$, where j denotes worker and i denotes foreman. The foreman reports this on channel $c.i$ to the farmer who, in turn, replies with a new task using channel $d.i$, The foreman then assigns the new task to the worker with channel $b.i.j$. The process-channel connection diagram for this network is given in figure 2.

Figure 2: Connection Diagram for Process Farm



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

$$\begin{aligned}
FARMER &= \Box_{i=0}^{n-1}\ c.i \rightarrow d.i \rightarrow FARMER \\
FOREMAN(i) &= \Box_{j=0}^{m-1}\ a.i.j \rightarrow c.i \rightarrow d.i \rightarrow b.i.j \rightarrow FOREMAN(i) \\
WORKER(i,j) &= a.i.j \rightarrow b.i.j \rightarrow WORKER(i,j)
\end{aligned}$$

Because we are to compose these processes in parallel we need to declare their alphabets.

$$\begin{aligned}
\alpha FARMER &= \{c.0,..,c.(n-1), d.0,..,d.(n-1)\rangle\} \\
\alpha FOREMAN(i) &= \{a.i.0,..,a.i.(m-1), b.i.0,..,b.i.(m-1), c.i, d.i\} \\
\alpha\, WORKER(i,j) &= \{a.i.j, b.i.j\} \\
FARM &= PAR\left\langle \begin{array}{c} WORKER(0,0),.., WORKER(n-1,m-1), \\ FOREMAN(0),.., FOREMAN(n-1), FARMER \end{array} \right\rangle
\end{aligned}$$

This system is a very simple example of a client-server network. The *client-server* protocol is a widely used design technique for building deadlock-free concurrent systems which is explained formally in [5] and [8]. It may be described in simple terms as follows.

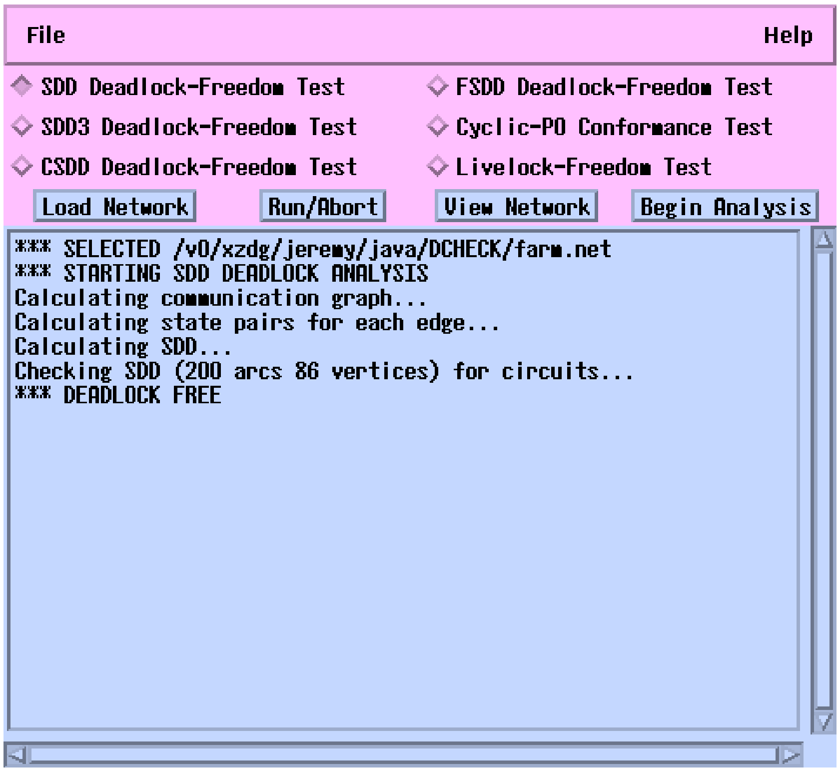
In a client-server network all channels between processes are partitioned into channel bundles. These consist *either* of single channels which are used by clients to send commands to servers, *or* pairs of channels which are used for requests which require answers. Processes may communicate with some neighbours as a client and with others as a server but the following protocol must be obeyed.

- All the component processes should be non-terminating and deadlock-free
- When a server is ready to receive a request or command from a client it must be willing to do so on all such channels.
- No process should ever try to communicate on an answer channel out of sequence, i.e. there must always be a request first.
- When a client sends a request to a server it must guarantee to accept the answer.

If these conditions are satisfied within a network then it is guaranteed deadlock-free as long as there is no cycle of client-server relationships, i.e. a sequence of processes $\langle P_1, ..P_n \rangle$ where each process is a client of its successor and P_n is also a client of P_1.

In the *FARM* network the relationship between worker and foreman and the relationship between foreman and farmer are both client to server so there is no cycle of client-server relationships. We can verify that the farm network is deadlock-free (for particular values of n and m) by running it through the SDD (State Dependence Digraph) test of the CSP verification tool Deadlock Checker[7], as shown in figure 3. We can also use this tool to explore the process transition systems and to draw a pictorial representation of the network process topology, as shown in figure 4.

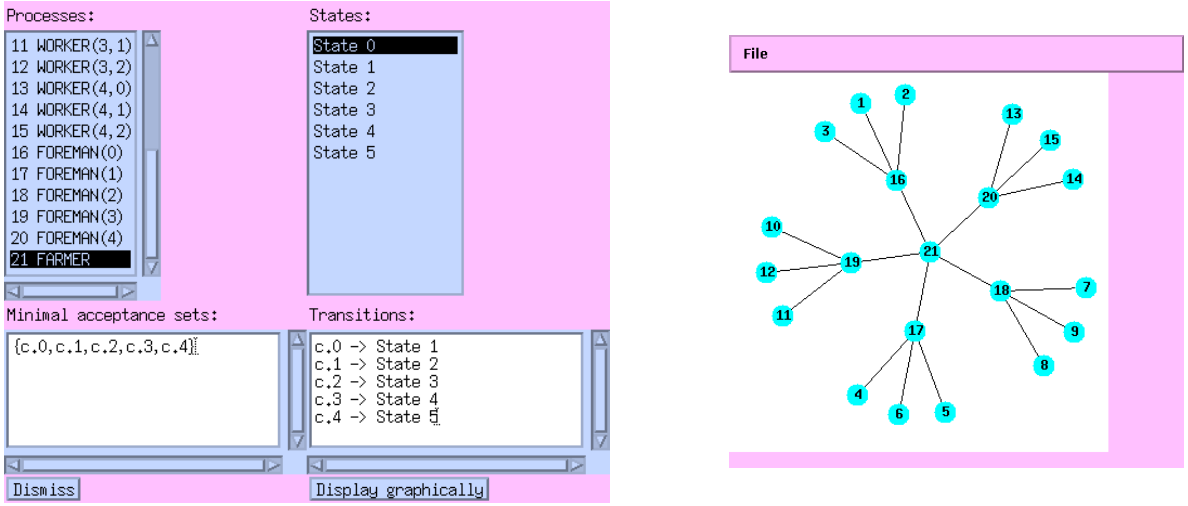
Figure 3: Verifying Deadlock Freedom for *FARM*



4 Example 2 – A User-Resource System

In this example we consider a system based on the client-server protocol which is then extended using the *resource allocation protocol*, thus merging two separate design rules for deadlock-freedom.

Figure 4: Exploring the FARM Process Structure



We consider a college with five philosophers who spend their time sitting around a table, thinking and challenging each other to arm-wrestling bouts. They are linearly ordered in terms of seniority, and observe the rule that a philosopher may only challenge his seniors to a contest. In this way a client-server hierarchy is imposed on the philosophers and deadlock-freedom can be guaranteed.

A flaw in this scenario, however, is that the philosophers may become very hungry after a while. In order to remedy this problem we shall mate the network with its more famous cousin, the Dining Philosophers. Forks are added to the system, one between each pair of philosophers, and an everlasting bowl of spaghetti in the middle of the table. In order to eat some spaghetti a philosopher must pick up both his adjacent forks. The connection diagram for the resulting network is shown in figure 5.

It is well known that the Dining Philosophers network, which is an illustration of a user resource system, is prone to deadlock. The deadlock arises if all the philosophers pick up one fork at the same time. In a user resource network processes are either active user threads or passive resource objects. Each resource waits to be claimed by a user for some purpose and then to be released again. Deadlock is prevented if the *resource allocation protocol* is adhered to. This means that a linear ordering is placed on the resources and each user process may only ever claim resources below those it is already holding. (Full details of the resource allocation protocol are given in [5].) In this case we shall number the forks from 0 to 4 around the table and make sure that each philosopher always picks up his forks in descending order.

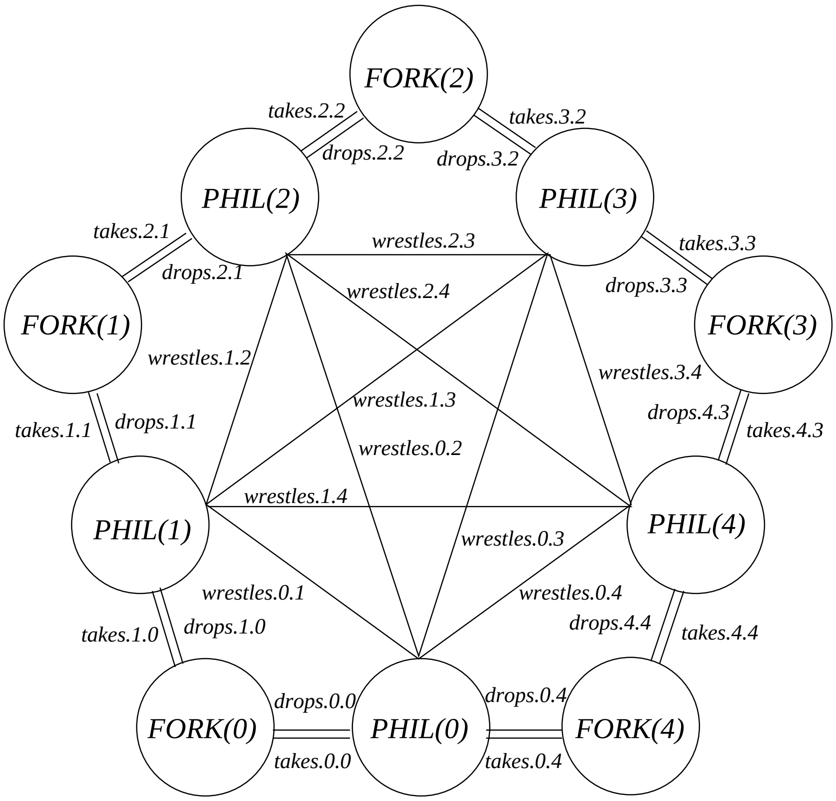
If the user processes within a user resource system can also communicate amongst themselves then a further stipulation is that they are not allowed to do so whilst holding a resource. So we also need to restrict arm-wrestling contests to being held between meals. (The subnetwork of users must, of course, itself be deadlock-free.)

The following CSP code defines the dining, arm-wrestling philosophers network, taking on board these design rules.

Let

$$
\begin{aligned}
PHIL(0) \quad &= \quad \begin{pmatrix} takes.0.4 \to takes.0.0 \to eats.0 \to \\ drops.0.4 \to drops.0.0 \to PHIL(0) \end{pmatrix} \sqcap \\
&\qquad \left(\sqcap_{i=1}^{4}\ wrestles.0.i \to PHIL(0)\right) \\
PHIL(i) \quad &= \quad \begin{pmatrix} \begin{pmatrix} takes.i.i \to takes.i.(i-1) \to eats.i \to \\ drops.i.(i-1) \to drops.i.i \to PHIL(i) \end{pmatrix} \sqcap \\ \left(\sqcap_{k=i+1}^{4}\ wrestles.i.k \to PHIL(i)\right) \end{pmatrix} \Box \\
&\qquad \left(\Box_{k=0}^{i-1}\ wrestles.k.i \to PHIL(i)\right) \qquad\qquad i = 1, 2, 3
\end{aligned}
$$

Figure 5: Connection Diagram for Arm-Wrestling Philosophers



$$
\begin{aligned}
PHIL(4) &= \begin{pmatrix} takes.4.4 \rightarrow takes.4.3 \rightarrow eats.4 \rightarrow \\ drops.4.3 \rightarrow drops.4.4 \rightarrow PHIL(4) \end{pmatrix} \Box \\
& \quad \left(\Box_{k=0}^{3}\ wrestles.k.4 \rightarrow PHIL(4)\right) \\
FORK(i) &= takes.i.i \rightarrow drops.i.i \rightarrow FORK(i) \ \Box \\
& \quad takes.(i+1).i \rightarrow drops.(i+1).i \rightarrow FORK(i) \\
\alpha PHIL(i) &= \{takes.i.i, takes.i.(i-1), eats.i, drops.i.(i-1), drops.i.i\} \\
& \cup \ \{wrestles.i.k | k > i\} \quad \cup \quad \{wrestles.k.i | k < i\} \\
\alpha FORK(i) &= \{takes.i.i, drops.i.i, takes.(i+1).i, drops.(i+1).i\} \\
ARMPHILS &= PAR \left\langle \begin{matrix} PHIL(0), & PHIL(1), & PHIL(2), & PHIL(3), & PHIL(4) \\ FORK(0), & FORK(1), & FORK(2), & FORK(3), & FORK(4) \end{matrix} \right\rangle
\end{aligned}
$$

where integer arithmetic is *modulo* 5. This network may also be verified as deadlock-free using the SDD technique from the Deadlock Checker tool.

The user resource model is very similar to the concept of *monitors* used for parallel programming in java. A realistic analogy to the arm-wrestling philosophers network might be a java distributed system, where the forks represent object monitors and the philosophers represent cooperating threads of computation.

5 Example 3 – Simulating a Physical System

Finally we consider a network which operates an interactive simulation of a bunjee-jump, based on an occam program due to P. H. Welch[8, 9].

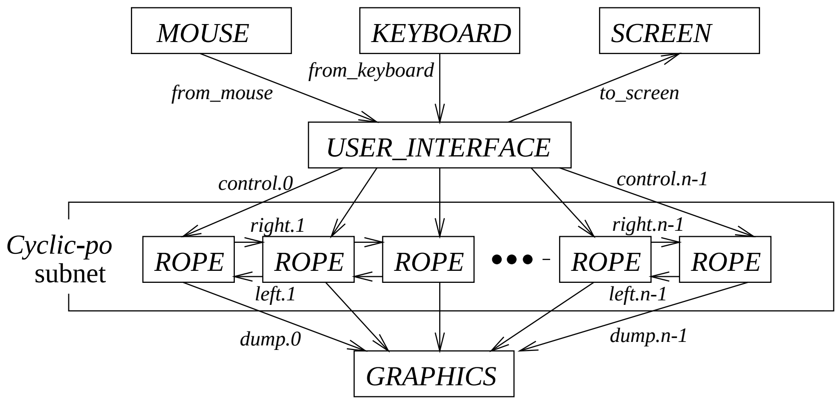
The computationally-intensive core of the system consists of a chain of *ROPE* processes which represent various discrete points along the length of the bunjee cord. These processes communicate with each other according to the *cyclic-po* protocol[6].

This is a design rule which relates to networks where each process runs cyclically, communicating on all its channels once each cycle according to some partial ordering. We define the *channel dependency graph* to be the aggregate of the partial orderings for each process. The network is deadlock-free if, and only if, the channel dependency graph contains no circuit.

In the case of the rope subnetwork, we shall instruct each component process to communicate on its two channels to the left in parallel (unless it is the leftmost component) and then on its two channels to the right in parallel (unless it is the rightmost component). The overall channel dependency relationship that this defines is clearly acyclic.

A client-server network is then superimposed over the cyclic-po core. Each *ROPE* process communicates as a client to a graphics handler *GRAPHICS*. and also as a server of *USER_INTERFACE*. This is a process which alters the control parameters for the simulation when instructed to do so by processes *KEYBOARD* or *MOUSE* and sends messages to a user console process *SCREEN*. The network connection diagram is shown in figure 6.

Figure 6: Connection Diagram for Bunjee-Jump Simulation



This time we shall present the CSP in the machine readable format of B. Scattergood that is understood by verification tools FDR[1] and Deadlock Checker. (See [7] for details.)

```
-- Define the set of indices for ROPE elements
indices = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19}
n = 20

-- Declare the various channels and channel arrays
pragma channel left, right, dump, control:indices
pragma channel from_mouse, from_keyboard, to_screen

-- Each rope process has four distinct phases on each cycle
ROPE(i) = ROPEA(i);ROPEB(i);ROPEC(i);ROPED(i);ROPE(i)
-- First it polls USER_INTERFACE (this construction cannot get blocked)
ROPEA(i) = control.i -> SKIP [] SKIP
-- Then, unless it is the leftmost process, it communicates to the left
ROPEB(i) = if (0 < i) then left.i -> right.i -> SKIP []
           right.i -> left.i -> SKIP
           else SKIP
-- Then, unless it is the rightmost process, it communicates to the right
ROPEC(i) = if (i < n-1)then left.(i+1) -> right.(i+1) -> SKIP []
           right.(i+1) -> left.(i+1) -> SKIP
           else SKIP
```

```
-- Finally it sometimes dumps its state to GRAPHICS
ROPED(i) = dump.i -> SKIP |~| SKIP

-- The USER_INTERFACE process may decide non deterministically
-- to send new parameters to the ROPE processes or to communicate
-- with SCREEN, MOUSE or KEYBOARD
USER_INTERFACE = ((|~| i:indices @ control.i -> USER_INTERFACE) |~|
                  to_screen -> USER_INTERFACE) []
                  from_mouse -> USER_INTERFACE []
                  from_keyboard -> USER_INTERFACE

-- User-controlled mouse process
MOUSE = from_mouse -> MOUSE

-- User's console
SCREEN = to_screen -> SCREEN

-- Keyboard input
KEYBOARD = from_keyboard -> KEYBOARD

-- Graphical display
GRAPHICS = [] i:indices @ dump.i -> GRAPHICS

-- Define the network for Deadlock Checker
-- BUNJEE = PAR <
--+  MOUSE,KEYBOARD,SCREEN,USER_INTERFACE,
--+  ROPE(0), ROPE(1), ROPE(2), ROPE(3), ROPE(4),
--+  ROPE(5), ROPE(6), ROPE(7), ROPE(8), ROPE(9),
--+  ROPE(10),ROPE(11),ROPE(12),ROPE(13),ROPE(14),
--+  ROPE(15),ROPE(16),ROPE(17),ROPE(18),ROPE(19),
--+  GRAPHICS
-- >
```

It is not generally safe to superimpose client-server communications over an existing cyclic-po network. Special precautions need to be taken[8]. If, in this case, we regard the rope subnetwork as being a single process so that only the client-server communications remain visible then we can see that it breaks the protocol in one respect. When it is ready to receive a command as a server it is not necessarily ready to communicate on all such channels, because the individual *ROPE* processes are only loosely synchronised. We circumvent this problem by using *polling* for these communications. The rope subnetwork can never get blocked waiting for a command from *USER_INTERFACE*.

Once again it is no problem for Deadlock Checker to prove this system deadlock-free using the SDD algorithm. (See figure 7.)

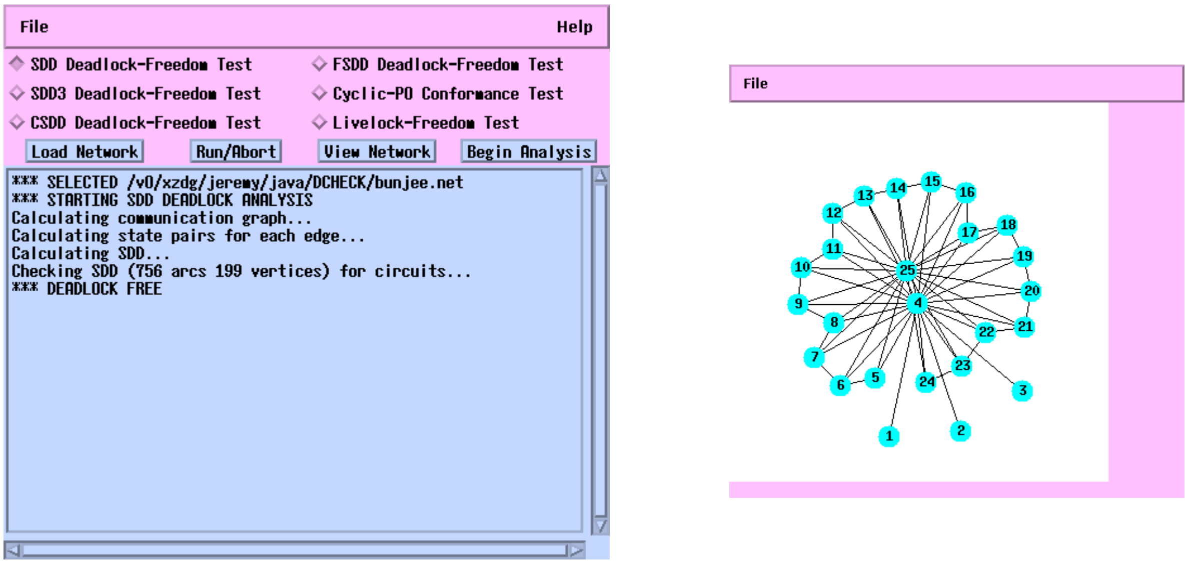
6 Related Issues

We have shown how to build fairly complex CSP deadlock-free networks using design rules[5, 6, 8, 9] and automatic verification with Deadlock Checker[7]. There are, of course, many other useful properties that can be verified using CSP. The FDR tool of Formal Systems Europe is to be recommended as an excellent means of proving general properties of CSP systems[1]. Its sole function is to verify the refinement condition $SPEC \sqsubseteq PROCESS$. Due to the completely partially ordered structure of all CSP processes this is a very general condition.

The advantage of Deadlock Checker over FDR for proving deadlock-freedom is in scalability. FDR uses exhaustive state checking which severely limits the size of networks that it can analyse. Deadlock Checker takes some shortcuts (described in [7]) which enable it to analyse arbitrarily large networks.

Many designers of concurrent systems will wish to build timing constraints into their networks. For this purpose a variant of CSP exists, *timed CSP*[2]. As yet there are no model checkers for timed CSP so all proofs that involve real time have to be done by hand. However it is often possible to approximate real time systems using discrete time, and FDR can then be used to prove properties of their behaviour.

Figure 7: Deadlock Analysis of Bunjee Jump Simulation



References

[1] *FDR User Manual and Tutorial* Formal Systems (Europe) Ltd. 3 Alfred Street, Oxford OX1 4EH. Version 1.4 1994

[2] J. Davies, *Specification and Proof in Real Time CSP*, Cambridge University Press 1993

[3] C. A. R. Hoare *Communicating Sequential Processes*, Communications of the ACM 21(8) 1978.

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

[5] J. M. R. Martin *The Design and Construction of Deadlock-Free Concurrent Systems.* D. Phil. Thesis, University of Buckingham 1996,also available at `http://www.hensa.ac.uk/parallel/theory/formal/csp`

[6] J. M. R. Martin, I. East and S. Jassim *Design Rules for Deadlock-Freedom*, Transputer Communications, September 1994.

[7] J. M. R. Martin and S. A. Jassim *A Tool for Proving Deadlock Freedom* in this volume

[8] J. M. R. Martin and P. H. Welch *A Design Strategy for Deadlock-Free Concurrent Systems* to appear in Transputer Communications

[9] P. H. Welch, G. R. R. Justo, and C. J. Willcock *High-Level Paradigms for Deadlock-Free High-Performance Systems*, Transputer Applications and Systems '93, IOS Press 1993.