Tactics for State Space Reduction on the Concurrency Workbench

Matthew J. Morley

February 1990*

Abstract

The Concurrency Workbench is a software tool catering for the automatic analysis of finite state processes expressed in Robin Milner's Calculus of Communicating Systems (CCS). This system is highly flexible in that analyses of concurrent systems can be carried out using a variety of different process semantics. However, the Workbench is limited in its application to realistic problems by spatial constraints. These are imposed by the fact that the analyses carried out deal with a transition graph representation of processes which, for finite state systems, can be exponential in the number of agents in the CCS expression. To address this problem we apply some heuristic transformations in order to arrive at a semantically equivalent expression that leads to a smaller transition graph. This article discusses the techniques employed, the new features available in the Workbench, and suggests how they might be applied in reducing larger problems prior to analysis.

1 Introduction

The Concurrency Workbench [4, 7, 8] is an automated tool for the design and analysis of finite state systems. Its development has been motivated largely by the increased sophistication of computer systems, both in hardware and software terms, and the subsequent need to derive correctness results about their behaviour and integrity. A *specification* of a system is submitted to the Workbench for analysis using several different verification techniques such as model checking, or equivalence checking for instance, according to a variety of process semantics. The current implementation allows specifications to be written in Milner's Calculus of Communicating Systems [2] which has a firm mathematical footing and whose algebraic properties we will use in later sections.

The notion of equivalence between processes expressed in an algebra like CCS depends upon the semantics adopted. The Workbench is a versatile tool and in fact embraces many of the semantic accounts associated with CCS. Following Milner [6] we adopt observational congruence (which we write as $=$) as our semantics, and our aim is to preserve this notion of equality throughout.

*LFCS report series: ECS-LFCS-90-109

1.1 Motivating Agent Transformations

The Concurrency Workbench has a three tier structure. Input and output of agents and logical expressions occurs at the *interface* level. Here, processes are represented as CCS equations and the Workbench binds agent expressions to variables in a global agent environment. It is at this level that the heuristics we develop are to be applied.

At the *semantics* level, processes are represented by transition graphs which the system builds from information supplied at the interface level. Recall that a transition graph is a set of nodes representing states (or agents) and arcs that are labelled to indicate the transition from one state to the next. It is here, in the process of building such graphs, that the Workbench becomes limited by spatial constraints since the physical size of the graph depends upon the state space of the system in consideration. It has been found that for a wide range of problems the state space is prohibitively large for direct mechanical analysis. Finally, the *analysis* level provides the three main algorithms for the verification methods adopted by the Workbench: equivalence checking, preorder checking and model checking. See [4] for full operational details.

We are now able to offer some *agent transformations* that preserve equality up to observational congruence where we have concentrated this initial effort on the standard concurrent form (scf), viz:

$$(\mathsf{A1} \,|\, \mathsf{A2} \,|\, \ldots \,|\, \mathsf{An}) \backslash \mathsf{L}.$$

In order to be precise we declare that $(\mathsf{A1} \,|\, \mathsf{A2} \,|\, \ldots \,|\, \mathsf{An})$ is in scf. Then $\mathsf{P} \backslash \mathsf{L}$ and $\mathsf{P}[\mathsf{f}]$ are in scf as long as P is. For our purposes we also declare that $\alpha.\mathsf{P}$ is in scf if P is. A *process* is defined to be an agent that contains no free variables. In general our transformations will only work for processes although free variables will be tolerated in most cases.

1.2 Laws for Observational Congruence

In the rest of this section we overview the structure of this report and indicate which of the laws for observational congruence we will use in later sections.

Many of these transformations implicitly rely on the associativity and abrorption properties of the summation operator:

1	$\mathsf{P} + \mathsf{Q} = \mathsf{Q} + \mathsf{P}$	2	$\mathsf{P} + (\mathsf{Q} + \mathsf{R}) = (\mathsf{P} + \mathsf{Q}) + \mathsf{R}$
3	$\mathsf{P} + \mathsf{P} = \mathsf{P}$	4	$\mathsf{P} + \mathbf{0} = \mathsf{P}$

Of these it is the last two that are most important in this work since we can use them to simplify expressions and complex agents generated by the transformation procedures. We also rely in this on the laws for composition:

5	$\mathsf{P} \,|\, \mathsf{Q} = \mathsf{Q} \,|\, \mathsf{P}$	6	$\mathsf{P} \,|\, (\mathsf{Q} \,|\, \mathsf{R}) = (\mathsf{P} \,|\, \mathsf{Q}) \,|\, \mathsf{R}$
7	$\mathsf{P} \,|\, \mathbf{0} = \mathsf{P}$		

The τ-laws for observational congruence are of vital importance in CCS— they reflect the fact that only initial tau actions in an agent are of observational significance. In Section 2 we discuss the removal of such τ actions (as well as some methods for dealing with initial τ's) by implementing two of these laws, and one useful derived result:

8 $\mathsf{a}.\tau.\mathsf{P} = \mathsf{a}.\mathsf{P}$

9 $\mathsf{P} + \tau.\mathsf{P} = \tau.\mathsf{P}$

10 $\mathsf{P} + \tau.(\mathsf{P} + \mathsf{Q}) = \tau.(\mathsf{P} + \mathsf{Q})$

We also consider the removal of τ-loops from agents in scf, by implementing the recursion axiom:

11 if $\mathsf{P} \stackrel{def}{=} \tau.\mathsf{P} + \mathsf{Q}$ then $\mathsf{P} = \tau.\mathsf{Q}$.

In Section 3 we make use of the Recursion Law (see [2] for a statement of this) and the Expansion Theorem:

$$\begin{array}{rcl} (\mathsf{P}_1[\mathsf{f}_1] \,|\, \ldots \,|\, \mathsf{P}_n[\mathsf{f}_n]) \backslash \mathsf{L} & = & \sum \{\mathsf{f}_i(\mathsf{a}).(\mathsf{P}_1[\mathsf{f}_1] \,|\, \ldots \,|\, \mathsf{P}_i'[\mathsf{f}_i] \,|\, \ldots \,|\, \mathsf{P}_n[\mathsf{f}_n]) \backslash \mathsf{L}\} \\ & & \;|\, 1 \leq i \leq n,\ \mathsf{P}_i \xrightarrow{\mathsf{a}} \mathsf{P}_i',\ \mathsf{f}_i(\mathsf{a}) \notin \mathsf{L} \cup \overline{\mathsf{L}} \\ & + & \sum \{\tau.(\mathsf{P}_1[\mathsf{f}_1] \,|\, \ldots \,|\, \mathsf{P}_i'[\mathsf{f}_i] \,|\, \ldots \,|\, \mathsf{P}_j'[\mathsf{f}_j] \,|\, \ldots \,|\, \mathsf{P}_n[\mathsf{f}_n]) \backslash \mathsf{L}\} \\ & & \;|\, 1 \leq i < j \leq n, \text{ for } \mathsf{l},\mathsf{k}\ \mathsf{P}_i \xrightarrow{\mathsf{l}} \mathsf{P}_i',\ \mathsf{P}_j \xrightarrow{\mathsf{k}} \mathsf{P}_j',\ \overline{\mathsf{f}_i(\mathsf{l})} = \mathsf{f}_j(\mathsf{k}) \end{array}$$

Since our transformations deal mainly with agents that are in scf we use these laws to 'simplify' such agent expressions. In particular we wish to remove communication bottlenecks between pairs of agents— we call this *local expansion.* This implicitly relys on the laws for the restriction operator, in particular:

12 $(\mathsf{P} \,|\, \mathsf{Q}) \backslash \mathsf{L} = (\mathsf{P} \backslash \mathsf{L}) \,|\, (\mathsf{Q} \backslash \mathsf{L})$, provided $\mathcal{L}(\mathsf{P}) \cap \overline{\mathcal{L}(\mathsf{Q})} \cap (\mathsf{L} \cup \overline{\mathsf{L}}) = \emptyset$.

In Section 4 we look at the deadlock issue and consider how we can hide certain actions that an agent might perform. This hiding is achieved by replacing a given action by τ and, although this does not necessarily preserve the congruence, we can use this device in order to show for example, that an agent is free from deadlock.

2 Removing Silent Actions

2.1 Removing The Obvious

The immediate benefit that we derive from an application of the τ-laws is that we can identify syntactically identical states in the transition graph of a process. These we can collapse into a single state, as is illustrated by the following system:

$$\mathsf{A} \stackrel{def}{=} \mathsf{a}.\tau.\mathsf{A} \quad \mathsf{B} \stackrel{def}{=} \tau.\mathsf{B} + \mathsf{b}.\mathsf{C} + \overline{\mathsf{c}}.\mathbf{0} \quad \mathsf{C} \stackrel{def}{=} \overline{\mathsf{a}}.\mathsf{B} \quad \mathsf{S} \stackrel{def}{=} (\mathsf{A} \,|\, \mathsf{B}) \backslash \{\mathsf{a}\}$$

Since $\mathsf{a}.\tau.\mathsf{A} = \mathsf{a}.\mathsf{A}$ by the first τ-law we can transform $\mathsf{a}.\tau.\mathsf{A} \rightsquigarrow \mathsf{a}.\mathsf{A}$. (Here $\rightsquigarrow$ is used to indicate a transformation of one agent into another, otherwise we will use the notation of [2] throughout.) The diagram in Figure 1 demonstrates the effect of this procedure, which we call *tau filtration*, on the state space of S.

Dialogue: Example 1

Suppose then that we had in our environment the agents A, B, C and S as above. The dialogue with the Workbench that filters the unnecessary τ actions would be as follows:

```
!Command >>> tau filter
!Agent?   : A

Tau Filter  : A
Where
   A = a.A
```

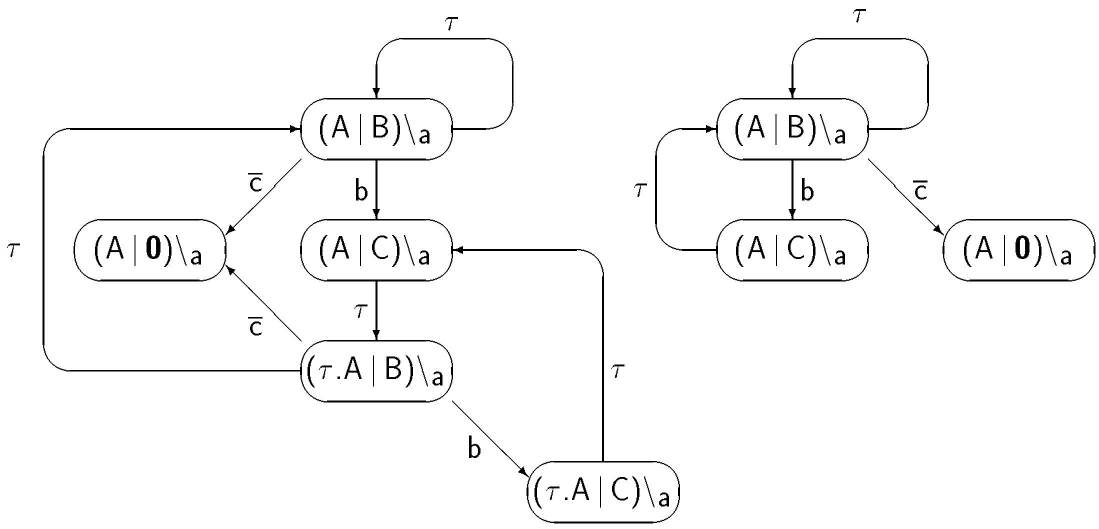

Figure 1: Removing unnecessary τ actions

The system then asks whether this new binding should be retained— in which case the old binding will be overwritten. Note that the same result would have been obtained by running the tau filter on the agent S instead of A.

Now consider the cyclic agent $\mathsf{P} = \tau.\mathsf{a}.\mathsf{P}$, which performs a sequence of a actions interspersed with τ's. Although the agent performs a τ action initially, we can see, again through the first law, that all the other τ's are redundant. We can filter simple cyclic agents such as this in an obvious way:

$$\mathsf{P} = \tau.\mathsf{P'} \quad \textit{where} \quad \mathsf{P'} = \mathsf{a}.\mathsf{P'}.$$

We are then left with a τ prefixed agent that gives rise to a source of a actions. Such an agent is called an α-cycler. This procedure need not be limited to this specific case of course, since any τ prefixed (recursive) agent can be treated in much the same way. The general idea, given $\mathsf{P} = \tau.\mathsf{Q}$, is to re-write P as $\mathsf{P} = \tau.\mathsf{P'}$ where $\mathsf{P'} = \mathsf{Q'}$. Q' is the same as Q except that P' replaces all guarded occurrences of P.

2.2 Removing Tau-Loops

If we can remove finite τ-sequences from CCS expressions then can we not also remove τ-loops? To answer this question consider the agent

$$(\mathsf{a}.\mathsf{P} \,|\, \mathsf{T}) \backslash_{\mathsf{a}} \quad \text{where} \quad \mathsf{T} = \tau.\mathsf{T}.$$

This agent is not deadlocked since it is observationally congruent to $\tau.\mathbf{0} \neq \mathbf{0}$. Yet if the τ-loop were removed we would immediately reveal the deadlock since $(\mathsf{a}.\mathsf{P}) \backslash_{\mathsf{a}} \sim \mathbf{0}$. As $\mathsf{T} = \tau.\mathbf{0}$ we can remove the τ-loop, replacing T by $\tau.\mathbf{0}$ to preserve the congruence. We will defer until Section 4 further discussion of the deadlock issue since it plays no further part here.

Realistically we are unlikely to meet such an explicit τ-loop as that above. A more fruitful result can be obtained by considering the recursion axiom:

$$\mathsf{P} \stackrel{def}{=} \tau.\mathsf{P} + \mathsf{E} \quad \rightsquigarrow \quad \mathsf{P} \stackrel{def}{=} \tau.\mathsf{E}.$$

This result uses the idea that all of the τ's in the loop can be absorbed into a single prefix— the doubtful reader is encouraged to consult [3] for a full discussion. However, we can use the result to remove the unwanted τ-loop whenever an agent such as P occurs in the context of a parallel composition. This may be of particular interest where:

$$\mathsf{P} = \tau.\mathsf{P} + \mathsf{a}.\mathsf{P} \quad \rightsquigarrow \quad \mathsf{P} = \tau.\mathsf{a}.\mathsf{P},$$

so we again reduce to a τ prefixed α-cycler which we will deal with in due course.

The τ-loop can crop up in other guises of course, one being:

$$(\mathsf{Q}\,|\,\mathsf{R}\,|\ldots|\,\mathsf{a}.\mathsf{Co}\,|\,\overline{\mathsf{a}}.\mathsf{Pr})\setminus\{\mathsf{a},\ldots\} \qquad \text{where } \mathsf{Co} = \mathsf{a}.\mathsf{Co} \text{ and } \mathsf{Pr} = \overline{\mathsf{a}}.\mathsf{Pr}.$$

This situation is not as unlikely as it may seem but, where a restricted agent contains such α-cyclers, a more interesting transformation is possible. The discussion on the removal of such α-cyclers from a parallel composition is also deferred until Section 4.2.

Dialogue: Example 1

Continuing with the example started earlier we now go on to remove the tau loop that occurs in B. Unlike the tau filter which can be applied to any agent, this function will only recognise agents that are in standard concurrent form:

!Command >>>remove tau loop
!Agent? : S

Tau Loop : $(\mathsf{A}\,|\,\mathsf{B})\setminus\mathsf{a}$
Where
$\mathsf{B} = \tau.\mathsf{B}'$
$\mathsf{B}' = (\mathsf{b}.\mathsf{C} + \overline{\mathsf{c}}.\mathsf{nil})$

The system again asks whether these bindings should be retained, but in this case a new binding identifier must be given for the composite agent. Note here that the original bindings for A, C and S remain unchanged but that the tau filter has separated the new agent into two.

2.3 Lifting up Initial Taus

Consider the agents $\mathsf{F} \stackrel{def}{=} (\mathsf{P}\,|\ldots|\,\mathsf{Z})\setminus\mathsf{L}$ in scf, and $\mathsf{P} \stackrel{def}{=} \tau.\mathsf{a}.\mathsf{P}$. It is clear that since P evolves silently into $\mathsf{a}.\mathsf{P}$, so F may evolve silently into the agent

$$\mathsf{F}' = (\mathsf{a}.\mathsf{P}\,|\ldots|\,\mathsf{Z})\setminus\mathsf{L}.$$

We can see that F and $\tau.\mathsf{F}'$ are equivalent up to $=$ since, if $\mathsf{F}\xrightarrow{\alpha}\mathsf{F}''$ then $\tau.\mathsf{F}'\xRightarrow{\alpha}\mathsf{F}''$. Trivially, if $\tau.\mathsf{F}'\xrightarrow{\tau}\mathsf{F}'$ then $\mathsf{F}\xrightarrow{\tau}\mathsf{F}'$. So, as long as we retain the initial tau to prefix the composition we can remove, or in this sense *lift up* the initial τ of P to the higher level:

$$(\tau.\mathsf{a}.\mathsf{P}\,|\ldots|\,\mathsf{Z})\setminus\mathsf{L} \quad \rightsquigarrow \quad \tau.(\mathsf{a}.\mathsf{P}\,|\ldots|\,\mathsf{Z})\setminus\mathsf{L}.$$

(For the moment we do not consider possible synchronisations between the parallel components of F which may also result in the silent evolution of the composite agent.) We can repeat this process to lift all such 'initial' τ's in the composite agent to the same level where they can be absorbed into a single τ prefix. We retain the τ prefix in order to preserve observational congruence.

Dialogue: Example 1

Suppose then that having removed the τ-loop as in the previous dialogue we had bound the resulting composition to the identifier New. We can now lift the initial τ as follows:

!Command >>>tau lift
!Agent? : New

Tau Lift : $\tau.(\mathsf{A} \,|\, \mathsf{B'})\backslash_{\mathsf{a}}$
Where
$\mathsf{B'} = \mathsf{b.C} + \bar{\mathsf{c}}.\mathbf{0}$
$\mathsf{C} = \bar{\mathsf{a}}.\mathsf{B'}$
...

Once more the tau filter has removed any unnecessary τ's. If this agent is to be retained then a new binding identifier must be given.

3 Local Expansion

3.1 Removing Taus caused by Initial Communications

We now extend the idea of the previous section to the case where a pair of agents in the composition can only evolve through a communication on some restricted action, so giving rise to a τ transition in the composite. Suppose that we have $\mathsf{P} \stackrel{def}{=} \mathsf{a.x.P}$ and $\mathsf{Q} \stackrel{def}{=} \mathsf{b.\bar{x}.Q}$, then our interest here would be in an agent of the form:

$$\mathsf{S} \quad = \quad (\mathsf{P} \,|\, \bar{\mathsf{a}}.\mathsf{Q} \,|\, \mathsf{R})\backslash\{\mathsf{a},\mathsf{b},\ldots\},$$

where a and $\bar{\mathsf{a}}$ are *not* in the sort of R. This restriction on a is necessary since otherwise P or $\bar{\mathsf{a}}.\mathsf{Q}$ might evolve autonomously by synchronising with some external agent. The condition that neither a nor $\bar{\mathsf{a}}$ is in the sort of R is necessary since we require that there be a unique way in which the composition may evolve through a actions, (Otherwise we would have to consider all possible interleavings of the agents involving a.)

In these circumstances we can move the restriction on a inwards so that it binds tightly on P and $\bar{\mathsf{a}}.\mathsf{Q}$, viz:

$$\mathsf{S} \quad = \quad ((\mathsf{P} \,|\, \bar{\mathsf{a}}.\mathsf{Q})\backslash\{\mathsf{a}\} \,|\, \mathsf{R})\backslash\{\mathsf{b},\ldots\}.$$

An agent that is restricted in this manner contains a *bottleneck*, in this case on the action a. Until the synchronisation has taken place no other agent can communicate with P or Q. Bottlenecks in communicating systems are common fare, and this kind of situation can often arise in communication protocols. In such protocols it is common to find a single hidden action shared between a pair of processes (modelling a private communication, perhaps). We discuss the removal of bottlenecks in the next section.

Now the agent S above can perform a τ action:

$$\mathsf{S} \ \xrightarrow{\tau}\ (\mathsf{x.P} \,|\, \mathsf{Q} \,|\, \mathsf{R}) \backslash \{\mathsf{a,b,\ldots}\}$$

where this is one possible transition for S. Recalling our earlier discussion concerning initial τ actions in composite agents, we can see that

$$\mathsf{S} \ = \ \tau.(\mathsf{x.P} \,|\, \mathsf{Q} \,|\, \mathsf{R}) \backslash \{\mathsf{a,b,\ldots}\}.$$

Our conclusion then is that as long as we retain the initial τ prefix we can substitute for S the expression $\tau.(\mathsf{x.P} \,|\, \mathsf{Q} \,|\, \mathsf{R}) \backslash \{\mathsf{a,b,\ldots}\}$. We can look for other initial communication events like this and absorb all of the silent transitions into the single τ prefix.

3.2 Removing Communication Bottlenecks

The above process is essentially a one-step *local* expansion of the agent $(\mathsf{P} \,|\, \overline{\mathsf{a}}.\mathsf{Q}) \backslash_{\mathsf{a}}$. We now introduce a more powerful technique for dealing with bottleneck agents through a couple of examples. A more realistic example is examined in Section 5.

Suppose that P and Q are as before, but that $\mathsf{S} \stackrel{def}{=} (\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}}$. Two applications of the Expansion Theorem (see Section 1.2) to the agent S yields:

$$\begin{aligned} \mathsf{S} \ &= \ \mathsf{a}.(\mathsf{x.P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}} + \mathsf{b}.(\mathsf{P} \,|\, \overline{\mathsf{x}}.\mathsf{Q}) \backslash_{\mathsf{x}} \\ &= \ \mathsf{a.b}.(\mathsf{x.P} \,|\, \overline{\mathsf{x}}.\mathsf{Q}) \backslash_{\mathsf{x}} + \mathsf{b.a}.(\mathsf{x.P} \,|\, \overline{\mathsf{x}}.\mathsf{Q}) \backslash_{\mathsf{x}}, \end{aligned}$$

and both branches in the expansion characterise a bottleneck on x. One further application of the Expansion Theorem (and removing the unnecessary τ's) gives:

$$\mathsf{S} \ = \ (\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}} \ = \ \mathsf{a.b}.(\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}} + \mathsf{b.a}.(\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}}.$$

We see that $(\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}}$ is a solution to the recursive equation $\mathsf{X} = \mathsf{a.b.X} + \mathsf{b.a.X}$ where X is guarded and sequential in the right hand expression. By the Recursion Law we have that $\mathsf{X} = (\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}} = \mathsf{S}$, where $\mathsf{X} \stackrel{def}{=} \mathsf{a.b.X} + \mathsf{b.a.X}$. The reader might like to consider what would happen if instead we had $\mathsf{S} \stackrel{def}{=} (\mathsf{x.P} \,|\, \mathsf{Q}) \backslash_{\mathsf{x}}$ or $\mathsf{S} \stackrel{def}{=} (\mathsf{x.P} \,|\, \overline{\mathsf{x}}.\mathsf{Q}) \backslash_{\mathsf{x}}$.

In this example, what has happened is that both branches of the derivation have returned to the initial state so we can write down a *single* equation that describes the behaviour of the pair. Now if P and Q were components of a larger system:

$$(\mathsf{P} \,|\, \mathsf{Q} \,|\, \ldots \,|\, \mathsf{Z}) \backslash \{\mathsf{x},\ldots\},$$

then, if we could establish that the communication through x were private to P and Q we could replace the component $\mathsf{P} \,|\, \mathsf{Q}$ by the single recursive agent X. (We might as well also remove x from the restriction since it is no longer needed.)

This simple example illustrates the general principle underlying the local expansion procedure. As long as we require that the derivation along each branch of the expansion returns to the initial state then we will be able to find a single equation to express the behaviour of the expanded component. We now demonstrate that this requirement is a little too restrictive, and in fact turns out to be unnecessary.

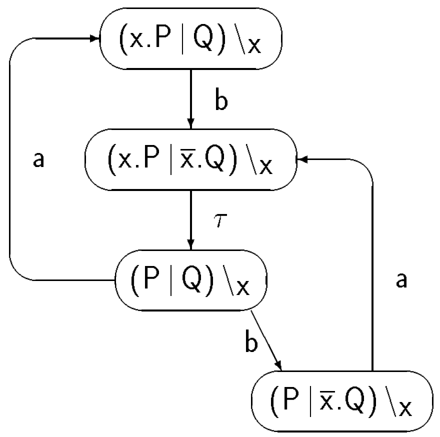

Figure 2: The digram represents the sequence of derivation steps 1 to 6

3.3 Generalising Pairwise Local Expansion

Suppose that P and Q are defined as above, but that $\mathsf{S} \stackrel{def}{=} (\mathsf{x.P} \,|\, \mathsf{Q})\backslash_{\mathsf{x}}$. We can apply the Expansion Theorem to yield the sequence of equations below:

$$
\begin{array}{rcll}
\mathsf{S} & = & (\mathsf{x.P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}} & \ldots(1) \\
 & = & \mathsf{b}.\tau.(\mathsf{x.P}\,|\,\overline{\mathsf{x}}.\mathsf{Q})\backslash_{\mathsf{x}} & \ldots(2) \\
 & = & \mathsf{b}.\tau.(\mathsf{P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}} & \ldots(3) \\
 & = & \mathsf{b}.\tau.(\mathsf{a}.(\mathsf{x.P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}} + \mathsf{b}.(\mathsf{P}\,|\,\overline{\mathsf{x}}.\mathsf{Q})\backslash_{\mathsf{x}}) & \ldots(4) \\
 & = & \mathsf{b}.\tau.(\mathsf{a}.(\mathsf{x.P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}} + \mathsf{b.a}.(\mathsf{x.P}\,|\,\overline{\mathsf{x}}.\mathsf{Q})\backslash_{\mathsf{x}}) & \ldots(5) \\
 & = & \mathsf{b}.\tau.(\mathsf{a}.(\mathsf{x.P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}} + \mathsf{b.a}.\tau.(\mathsf{P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}}) & \ldots(6)
\end{array}
$$

From this we can see that in step (4) the first branch has returned to the initial state whereas the second has not. However, at the next step in the expansion the second branch returns to a state that has already appeared in the derivation at step (2). In terms of transitions, this is clearly illustrated in the diagram in Figure 2. Now if we compare (2) and (5) with the mutually recursive equations

$$
\begin{array}{l}
\mathsf{X} = \mathsf{b.Y} \\
\mathsf{Y} = \tau.(\mathsf{a.X} + \mathsf{b.a.Y})
\end{array}
$$

we see that $(\mathsf{x.P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}}$ and $(\mathsf{x.P}\,|\,\overline{\mathsf{x}}.\mathsf{Q})\backslash_{\mathsf{x}}$ are solutions for X and Y respectively. Appealing to the Recursion Law once more will give us that $\mathsf{X} = (\mathsf{x.P}\,|\,\mathsf{Q})\backslash_{\mathsf{x}} = \mathsf{S}$ where $\mathsf{X} \stackrel{def}{=} \mathsf{b.Y}$ and $\mathsf{Y} \stackrel{def}{=} \tau.(\mathsf{a.X} + \mathsf{b.a.Y})$. As a slight improvement we can see that the initial τ in the definition of Y is unnecessary since Y is guarded in both X and Y (compare (3) and (6) with equations $\mathsf{X} = \mathsf{b.Y}$ and $\mathsf{Y} = \mathsf{a.X} + \mathsf{b.a.Y}$).

For any finite state system we can find a set of equations that describe its behaviour simply by writing down one equation for each state. However, this would give rise to a profusion of mutually recursive equations so we attempt, as above, to find a ‘minimal’ set of equations. This process can therefore be used to remove a pair of agents from a composition and replace them with a single recursive agent that is = to the pair.

The temptation that we should avoid is that of generalising this procedure to an arbitrary set of agents in a composition. This would seem to be a move away from the idea that we wish to make a sensible reduction in the state space of some large system. In generalising this local expansion further at least two problems would be encountered:

- We may in fact preserve states in the derivation that are unreachable in the wider context since not all global information is available at the local level[1]— for instance, if the pair of agents contains bottlenecks on two different actions.

- There is always a possibility of trying to expand a subcomponent that itself has a very large number of states— one would first need to apply some agent transformations to this subcomponent to limit its spatial complexity.

It is difficult to trap either of these possibilities in a simple implementation of the local expansion procedure, although a complete solution to the problem of state space reduction would obviously address both of these points. Since these transformations are designed to be 'user driven' the user should beware of being too ambitious!

4 Proving Deadlock Freedom

Deadlock is first of the many novel bugs that a person new to parallel processing will encounter. It is also the most persistent source of trouble in communicating systems. In CCS, deadlock can be associated with $\mathbf{0}$, the process that can do nothing, but as a 'bug' it is characterised by the agent $(\mathsf{a.P})\backslash_{\mathsf{a}} \sim \mathbf{0}$.

In the context of proving that an agent is free from deadlock we can relax the constraint that we must preserve $=$ at all costs— we need not even preserve deadlock freedom. Naturally, in order to use the above device to expose deadlocks the Workbench must calculate the state space to locate agents such as $(\mathsf{a.P})\backslash_{\mathsf{a}}$. Therein is the problem: in order to prove deadlock freedom the entire state space must be calculated. We show how to reduce the state space of a process while maintaining its potential for deadlock— but we may not, in general, preserve the equality relation.

4.1 Hiding Visible Actions

The first observation is that deadlock is associated with a restriction, so *visible* actions play no part in the issue. It is possible to hide a visible action α in an expression E by rewriting E with τ in place of α, although subtleties are introduced if E contains relabelled agents. We can then remove many these τ's through the τ-laws. To illustrate this idea we present a larger example that occupies much of the rest of this report.

[1] In this first implementation we have concentrated on binding a pair of agents as tight as possible to a restriction on a single action; in principle we could generalise this to any number of restricted actions.

Dialogue: Example 2

Consider an environment where the following agents are defined:

$$\begin{aligned}
&\mathsf{Buff} \stackrel{def}{=} \mathsf{in}.\overline{\mathsf{out}}.\mathsf{Buff} + \overline{\mathsf{empty}}.\mathsf{Buff}\\
&\mathsf{PF} \stackrel{def}{=} \mathsf{Buff}\,[\mathsf{ppf/in},\overline{\mathsf{pff}}/\overline{\mathsf{out}},\overline{\mathsf{nopf}}/\overline{\mathsf{empty}}\\
&\mathsf{FO} \stackrel{def}{=} \mathsf{pff}.\overline{\mathsf{fileout}}.\mathsf{FO}\\
&\mathsf{SysB} \stackrel{def}{=} (\mathsf{UO} \,|\, \mathsf{UI} \,|\, \mathsf{UP} \,|\, \mathsf{PO} \,|\, \mathsf{PI} \,|\, \mathsf{PF} \,|\, \mathsf{FP} \,|\, \mathsf{FO} \,|\, \mathsf{FI})\backslash \mathsf{L}
\end{aligned}$$

where $\mathsf{L} = \{\mathsf{pu,uup,upp,noup,ppf,pff,nopf,ffp,fpp,nopf}\}$. The rest of the agents in this composition will be defined later since our only concern, for the moment, is that pff is known only to PF and FO. We hide the action $\overline{\mathsf{fileout}}$ as follows:

!Command >>> hide action
!Action? : $\overline{\mathsf{fileout}}$
!Agent? : SysB

Hiding : $\mathsf{SysB}_1 = (\ldots \,|\, \mathsf{FO}_2 \,|\, \ldots)\backslash \mathsf{L}$
Where
$\mathsf{FO}_2 = \mathsf{pff}.\mathsf{FO}_2$
...

Notice that the τ that would be introduced (in the agent FO_2) has been filtered. The function responds with a complete list of the agents in the definition of SysB_1. It is worth pointing out also, that the bindings for the agents SysB and FO will not have been removed from the agent environment.

This simple procedure will always preserve deadlock freedom, but it will also preserve the *potential* for deadlock. Some or all of the visible actions in an agent can be hidden in this manner as long as our only interest is in demonstrating deadlock freedom. On the other hand, if we were interested in locating deadlocks then it is visible sequences that are of interest. We can still use this hiding device to help locate deadlocks but some of the visible actions need to be retained.

4.2 Alpha-cyclers

As a model for some computational process, or a physical device, the α-cycler that we can define in CCS is somewhat idealised since it admits no delay. Nevertheless, the idea of a constant 'producer' or 'consumer' is frequent enough in communication protocols that it warrants some attention. Consider then, the agents:

$$(\mathsf{a.Co} \,|\, \overline{\mathsf{a}}.\mathsf{Pr} \,|\, \mathsf{a.X})\backslash\{\mathsf{a},\ldots\} \quad \textit{where} \quad \mathsf{Co} \stackrel{def}{=} \mathsf{a.Co} \quad \textit{and} \quad \mathsf{Pr} \stackrel{def}{=} \overline{\mathsf{a}}.\mathsf{Pr},$$

and where we are not concerned with the details of the agent X. An α-cycler, as a component of a parallel system, is always ready to produce or consume an α action. Now, let us arrange the above expression so that we have one α-cycler in parallel with another agent:

$$(\overline{\mathsf{a}}.\mathsf{Pr} \,|\, \mathsf{R})\backslash\{\mathsf{a},\ldots\} \quad \textit{where} \quad \mathsf{R} = (\mathsf{Co} \,|\, \mathsf{a.X}).$$

Now, for *any* agent R in parallel with the cycler Pr, if $\mathsf{a} \in \mathcal{L}(\mathsf{R})$ then whenever R wishes to make an a action the cycler will be ready with the complementary action and so the communication can always take place. Thus the system will *always* be able to proceed through a τ transition whenever R may perform action a. We can therefore remove the $\overline{\mathsf{a}}$-cycler from the composition altogether as long as we make the τ transition explicit by hiding the action a in R.

$$(\overline{\mathsf{a}}.\mathsf{Pr} \,|\, \mathsf{R}) \backslash \{\mathsf{a}, \ldots\} \;\rightsquigarrow\; (\mathsf{R}[\mathsf{a}{:=}\tau]) \backslash \{\mathsf{a}, \ldots\}$$

It must be remembered that a and $\overline{\mathsf{a}}$ are both restricted in the composition, and since $\overline{\mathsf{a}}$ may be in the sort of R, the restriction on a is still required. This is very important, for if we consider instead:

$$(\mathsf{a}.\mathsf{Co} \,|\, (\mathsf{Pr} \,|\, \mathsf{a}.\mathsf{X})) \backslash \{\mathsf{a}, \ldots\} \;\rightsquigarrow\; ((\mathsf{Pr} \,|\, \mathsf{a}.\mathsf{X})[\overline{\mathsf{a}}{:=}\tau]) \backslash \{\mathsf{a}, \ldots\}$$

we can see that deadlock freedom is not preserved.

Since the restriction on a must be retained, and as this may give rise to some new deadlock it is clear that this process does not *necessarily* preserve deadlock freedom. However, if the transformed agent is found not to deadlock then deadlock freedom is proven in the unreduced agent. We can now derive a new 'law' concerning these α-cyclers:

$$\boxed{\text{if } \mathsf{Q} \overset{def}{=} \mathsf{a}.\mathsf{Q} \text{ and } \mathsf{a} \text{ is not in the sort of } \mathsf{P} \text{ then } (\mathsf{P} \,|\, \mathsf{Q}) \backslash_{\mathsf{a}} = \; \mathsf{P}\,[\overline{\mathsf{a}} := \tau]}$$

Note that the implementation of this procedure is sensitive enough to detect circumstances when this hiding *may* introduce a deadlock.

Dialogue: Example 2

In hiding the action $\overline{\mathsf{fileout}}$ above we reduced one of the parallel components of the system to a pff-cycler. We can now remove this agent from the composition altogether:

!Command >>> remove alpha cycler
!Action? : pff
!Agent? : SysB_1

Cycler : $(\mathsf{UO} \,|\, \mathsf{UI} \,|\, \mathsf{UP} \,|\, \mathsf{PO} \,|\, \mathsf{PI} \,|\, \mathsf{PF}_4 \,|\, \mathsf{FP} \,|\, \mathsf{FI}) \backslash \mathsf{L}$
Where
$\mathsf{PF}_4 = \mathsf{Buff}_5\,[\mathsf{ppf/in}, \overline{\mathsf{nopf}}/\overline{\mathsf{empty}}]$
$\mathsf{Buff}_5 = \mathsf{in}.\mathsf{Buff}_5 + \overline{\mathsf{empty}}.\mathsf{Buff}_5$
...

There are several points to make here. Firstly, in the definition of Buff_5, the τ introduced in hiding pff has been filtered. Secondly, the action pff has been removed from the relabeling function in the definition of PF_4. Thirdly, the cyclic agent has gone from the composition and, finally, the function will ask for an identifier to rebind the composition in the agent environment.

5 A Larger Example

The VAX mail system in the Computer Science department at Edinburgh University had a bug— it was found to deadlock. In order to locate the source of the trouble an abstraction of the system was constructed in CCS and submitted to the Workbench for analysis. Once the deadlocks were found the problem was then to come up with an alternative design and prove, using the Workbench, that it was free from deadlock. The alternative design is the agent SysB that we saw in the previous section.

The easiest way to go about proving deadlock freedom is to use the 'find deadlock' command of the Workbench. In this case this entailed constructing the entire state space of SysB— a task beyond the spatial limitations of the Sun3 workstation running the Workbench. Here, two techniques for reducing the problem to a level at which direct analysis is possible are outlined.

It is worth noting that the model checking facility of the Workbench can be employed to prove that SysB satisfies the formula:

$$\nu\, Z.(<.> \texttt{true} \wedge [\,.\,]Z)$$

This expresses deadlock freedom: it is always possible to perform some action and whatever action is performed the formula is satisfied.

5.1 Hiding Actions

We will now continue with the example started in Section 4.1. Some of the other components of the system are as follows:

$$\mathsf{FI} \stackrel{def}{=} \mathsf{filein}.\overline{\mathsf{ffp}}.\mathsf{FI} \qquad \mathsf{UO} \stackrel{def}{=} \mathsf{mailin}.\overline{\mathsf{uup}}.\mathsf{UO}$$

$$\mathsf{FP} \stackrel{def}{=} \mathsf{Buff}\,[\mathsf{ffp/in},\overline{\mathsf{fpp}}/\overline{\mathsf{out}},\overline{\mathsf{nofp}}/\overline{\mathsf{empty}}] \quad \mathsf{UP} \stackrel{def}{=} \mathsf{Buff}\,[\mathsf{uup/in},\overline{\mathsf{upp}}/\overline{\mathsf{out}},\overline{\mathsf{noup}}/\overline{\mathsf{empty}}]$$

For the moment we will continue to hide visible actions, namely filein and mailin.

!Command >>> hide action
!Action? : filein
!Agent? : SysB_3

Hiding : $\mathsf{SysB}_6 = (\ldots \mid \mathsf{FI}_7 \mid \ldots)\backslash \mathsf{L}$
Where
$\mathsf{FI}_7 = \tau.\overline{\mathsf{ffp}}.\mathsf{FI}_7$
...
!Command >>> tau lift
!Agent? : SysB_6

Tau Lift : $\tau.(\ldots \mid \mathsf{FI}_{7.1} \mid \ldots)\backslash \mathsf{L}$
Where
$\mathsf{FI}_{7.2} = \tau.\mathsf{FI}_{7.1}$
$\mathsf{FI}_{7.1} = \overline{\mathsf{ffp}}.\mathsf{FI}_{7.1}$
...

Although it is not really needed here, the tau-lift function returns a binding for $\mathsf{FI}_{7.2}$. For this command a new binding identifier needs to be given in order to rebind the parallel composition. Suppose that this is bound to New:

!Command >>> remove alpha cycler
!Action?   : $\overline{\mathsf{ffp}}$
!Agent?   : New

Cycler   : $\tau.(\mathsf{UO} \mid \mathsf{UI} \mid \mathsf{UP} \mid \mathsf{PO} \mid \mathsf{PI} \mid \mathsf{PF}_4 \mid \mathsf{FP}_8) \backslash \mathsf{L}$
Where
  $\mathsf{FP}_8 = \mathsf{Buff}_9\,[\overline{\mathsf{fpp}}/\overline{\mathsf{out}}, \overline{\mathsf{nofp}}/\overline{\mathsf{empty}}]$
  $\mathsf{Buff}_9 = \tau.\overline{\mathsf{out}}.\mathsf{Buff}_9 + \overline{\mathsf{empty}}.\mathsf{Buff}_9$
  ...

And so on. We can now go on to remove the agent UO in exactly the same way so that the reduced agent is a composition of six agents instead of nine. For the sake of completeness we record that:

$$\begin{array}{ll} \mathsf{UI} \stackrel{def}{=} \mathsf{pu}.\mathsf{UI1} & \mathsf{UI1} \stackrel{def}{=} \overline{\mathsf{mailout}}.\mathsf{UI} + \overline{\mathsf{uup}}.\mathsf{UI} \\ \mathsf{PI} \stackrel{def}{=} \mathsf{fpp}.\mathsf{PI1} & \mathsf{PI1} \stackrel{def}{=} \overline{\mathsf{pu}}.\mathsf{PI} + \overline{\mathsf{ppf}}.\mathsf{PI} \\ \mathsf{PO} \stackrel{def}{=} \mathsf{upp}.\overline{\mathsf{ppf}}.\mathsf{PO}. & \end{array}$$

Now if we can prove that the final reduced agent (New) is free from deadlock then it follows from our earlier discussion that the agent SysB is also deadlock free. The reader can verify that this is in fact the case by following the example on the Workbench.

5.2 Local Expansion

We can arrive at the same conclusion without hiding, and therefore preserving $=$, through judicious use of local expansion. We note that there are several pairs of agents that can be tightly bound on a single action, including:

$$(\mathsf{UP} \mid \mathsf{PO}) \backslash_{\mathsf{upp}} \text{ and } (\mathsf{UI} \mid \mathsf{PI}) \backslash_{\mathsf{pu}} \text{ and } (\mathsf{PF} \mid \mathsf{FO}) \backslash_{\mathsf{ffp}}.$$

A function has been implemented in the Workbench that can return this information about a given system. We now apply the local expansion procedure to these pairs in turn:

!Command >>> bottle necks
!Action?   : upp
!Agent?   : SysB

Bottleneck : $(\mathsf{LL}_0 \mid \mathsf{UO} \mid \mathsf{UI} \mid \mathsf{UP} \mid \mathsf{PI} \mid \mathsf{PF} \mid \mathsf{FP} \mid \mathsf{FI}) \backslash \mathsf{L}$
Where
  $\mathsf{LL}_0 = \mathsf{uup}.\mathsf{LL}_1 + \overline{\mathsf{noup}}.\mathsf{LL}_0$
  $\mathsf{LL}_1 = \mathsf{uup}.\overline{\mathsf{ppf}}.\mathsf{LL}_1 + \overline{\mathsf{noup}}.\mathsf{LL}_1 + \overline{\mathsf{ppf}}.\mathsf{LL}_0$

!Enter binding identifier: New
!Command >>> bottle necks
!Action?   : pu
!Agent?   : New

Bottleneck : $(\mathsf{LL}_0 \mid \mathsf{LL}_2 \mid \mathsf{UO} \mid \mathsf{UP} \mid \mathsf{PF} \mid \mathsf{FP} \mid \mathsf{FI}) \backslash \mathsf{L}$

Where

$\mathsf{LL}_2 = \mathsf{fpp}.\mathsf{LL}_3$

$\mathsf{LL}_3 = \mathsf{t}.\mathsf{LL}_4 + \overline{\mathsf{ppf}}.\mathsf{LL}_2$

$\mathsf{LL}_4 = \overline{\mathsf{mailout}}.\mathsf{LL}_2 + \overline{\mathsf{uup}}.\mathsf{LL}_2 + \mathsf{fpp}.(\overline{\mathsf{mailout}}.\mathsf{LL}_3 + \overline{\mathsf{uup}}.\mathsf{LL}_3 + \overline{\mathsf{ppf}}.\mathsf{LL}_4)$

!Enter binding identifier: New

...

Note that the bindings LL_0, etc., given in the above dialogue may not be exactly as returned by the Workbench. The reader can again verify that the reduced system obtained in this way is free from deadlock and, since New = SysB, other analyses of this system can also be carried out.

6 Conclusion

In this work we have not sought a complete solution to the problem of state space reduction since that is a task that would require a much greater effort than the time scale of a short MSc. project could allow. Instead we have implemented a small number of agent transformations that will take a closed CCS expression to an equivalent expression, up to observational congruence, that leads to a transition system having a smaller state space. We have also provided functions that are of specific use in proving deadlock freedom but which may not be 'safe' otherwise. Although these individual transformations are simple enough in themselves, their composition can yield some quite powerful results. Experience will indicate which are the most common or useful combinations so that refinements can be made to further automate this reduction procedure.

The important conclusion to draw from the discussion of deadlock in Section 4 is that although in some cases deadlock freedom is not preserved by our transformations, if the transformed agent can be shown to be deadlock free then the same result is true of the unreduced system. Conversely however, the existence of a deadlock in the reduced agent does not imply that such a state is reachable in the unreduced agent. The implementation can identify most situations where this is important.

We note that the results of the local expansion can immediately be minimised in a standard way— using the existing minimisation function. This efficiently minimises the state space of the expanded subsystem and so enhances the usefulness of this procedure.

Our experience then is that the small number of transformations that we have implemented are enough to open up the Workbench to automatic analysis of some interesting realistic problems that have hitherto proved too large for the machine to analyse directly. Suggestion for future additions would of course be welcomed.

Acknowledgement

I must express my gratitude to my supervisor, Bernhard Steffen, for his continuous support and his occasional stern criticism. I should also like to thank George Cleland and David Walker for providing me with some inspiration and for proof reading these notes. I should also thank Jo Blishen for helping with some difficulties with the ML.

References

[1] Robert Harper: *Introduction to Standard ML*, LFCS Report Series ECS-LFCS-86-14, Computer Science Department, Edinburgh University, November 1986.

[2] David Walker: *An Introduction to the Calculus of Communicating Systems*, LFCS Report Series ECS-LFCS-87-22, June 1988.

[3] Robin Milner: *A Complete Axiomatisation for Obsrevational Congruence of Finite State Behaviours*, Information and Computation, Vol 81, May 1989;
(also as LFCS Report Series ECS-LFCS-86-8, August 1986.)

[4] Rance Cleaveland, Joachim Parrow, Bernhard Steffen: *The Concurrency Workbench: A Semantics-Based Verification tool for Finite-State Systems*, LFCS Report Series ECS-LFCS-89-83, June 1989.

[5] Rance Cleaveland, Joachim Parrow, Bernhard Steffen: *The Concurrency Workbench: Operating Instructions*, Technical Memorandum, LFCS-TN-10, September 1988.

[6] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[7] Rance Cleaveland, Joachim Parrow, Bernhard Steffen: *A Semantics-Based Verification tool for Finite-State Systems*, Protocol Specification, Testing and Verification, IX. Elsevier Science Publications B. V. (North Holland) 1990, pp 287–302.

[8] Rance Cleaveland, Joachim Parrow, Bernhard Steffen: *The Concurrency Workbench*, Automatic Verification Methods for Finite-State Systems. Springer Verlag LNCS, Vol 407, 1990, pp 24–37.