LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Automated Analysis of Mutual Exclusion Algorithms using CCS

by

D. J. Walker

LFCS Report Series

ECS-LFCS-89-91
(also published as CSR-308-89)

August 1989

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Automated Analysis of Mutual Exclusion Algorithms using CCS

Preprint of a paper to be published in Formal Aspects of Computing

D. J. Walker
Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, U.K.

Abstract

A number of mutual exclusion algorithms are studied by representing them as agents in the Calculus of Communicating Systems and using an automated tool embodying some of the theory of the Calculus to analyse the representations. It is determined whether or not each of the algorithms preserves mutual exclusion and is live.

Keywords: CCS, Mutual Exclusion Algorithms, Concurrency Workbench

1 Introduction

During the last decade a number of algebraic calculi of concurrency and communication have been introduced and studied. Each of these consists essentially of a language for describing concurrent and communicating systems, a means of assigning meaning to the terms of the language, and an array of techniques for analysing systems described in the language. The literature devoted to the investigation of these calculi contains many small illustrative examples. However in relatively little published work are any of the calculi used to analyse somewhat more substantial systems. One notable exception is Milner's monograph [Mil80] in which the Calculus of Communicating Systems (CCS) was introduced. This work and its extension and elaboration [Mil89] contain a wide range of examples. The most ambitious of these require such detailed calculation that they lie at the limit of reliable pencil-and-paper application of the Calculus.

In order to investigate further the applicability of calculi such as CCS by tackling case studies more substantial than those considered to date, it is thus essential that suitable automated tools be available to aid in the analyses. The

`Concurrency Workbench` [CPS89a], [CPS89b] currently under development in a joint SERC-funded project between the Laboratory for Foundations of Computer Science in the University of Edinburgh and the Department of Computer Science in the University of Sussex, is such a tool. In addition to facilitating some applications of the Calculus, the Concurrency Workbench has been used successfully in teaching, and it is likely that it will prove useful in further theoretical work. The purpose of the present paper is to report the outcome of an experiment in which the Concurrency Workbench was used to investigate the behaviour of a number of concurrent algorithms.

The difficulty in understanding and reasoning reliably about all but the simplest concurrent algorithms has long been recognized. In this paper we study a number of mutual exclusion algorithms. These algorithms have been considered by many authors and a variety of techniques have been used to investigate their properties (see e.g. [Dij65, Kn66, Lam86]). Thus there is a ready source of material against which to compare the method and results of the present paper. This method may be briefly summarized as follows:

1. The algorithms were represented as CCS agents.

2. Questions of interest concerning the algorithms were formulated precisely within the framework of the Calculus.

3. The Concurrency Workbench was used to determine automatically the answers to the questions.

For each of the algorithms two questions were considered:

1. Does the algorithm preserve mutual exclusion?

2. Is the algorithm live (in the sense that if a process requests execution of its critical section, then it will eventually execute its critical section)?

CCS is by no means a purely algebraic calculus. For the semantics of the language is given in terms of labelled transition systems and such systems are natural models for modal and temporal logics. One point of this paper is to illustrate the interplay between algebraic and logical techniques.

Many temporal properties of systems have been classified as either *safety properties* or *liveness properties*. Preservation of mutual exclusion and liveness in the sense above are examples characteristic of these classes respectively. While it is generally acknowledged that the Calculus is suitable for the study of safety properties of systems, it is sometimes asserted that within CCS no adequate treatment of liveness properties is possible. We give here some evidence to indicate that this assertion is unwarranted. For further discussion of this point see [St89a], [SW89b].

Some familiarity with CCS would be useful for reading this paper, though the Preliminaries below contains a brief introduction to the Calculus, together with some material on modal and temporal logics and a short description of the Concurrency Workbench. The following section begins with an illustration of the method used to represent the algorithms as CCS agents, followed by a detailed discussion of the formulation within the Calculus of the questions of interest. It contains also representations of mutual exclusion algorithms due to Dekker, Dijkstra, Hyman, Knuth, Peterson and Lamport. The final section contains a summary of the results of the analyses, some discussion of the methods of the paper, and a brief summary of some related work.

2 Preliminaries

This section contains a very brief discussion of CCS, some material on modal and temporal logics, and a brief description of the Concurrency Workbench.

2.1 The Calculus of Communicating Systems

Originally introduced in [Mil80], Milner's Calculus of Communicating Systems was the first of the algebraic calculi of concurrency and communication. In this paper we use only the *pure* Calculus without explicit data communication. The terms of its language (*agents*) are constructed by means of a number of operators and a recursive definition mechanism. Agents are given a structured operational semantics as a labelled transition system, with labels drawn from a set *Act* of *actions*. The action set is of the form $\mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$ where the sets $\mathcal{A}$ and $\overline{\mathcal{A}}$ are in a one-one correspondence via $\bar{\cdot}$. τ is a distinguished *silent action*; all other actions are *visible*. Among the operators are *action prefixing*, denoted $a.$ ($a \in Act$), *summation* $+$, *composition* $|$ and *restriction* $\backslash L$ ($L \subseteq Act - \{\tau\}$). The rules for these operators include

- $a.P \xrightarrow{a} P$
- if $P \xrightarrow{a} P'$ then $P + Q \xrightarrow{a} P'$
- if $P \xrightarrow{a} P'$ then $P|Q \xrightarrow{a} P'|Q$
- if $P \xrightarrow{a} P'$ and $Q \xrightarrow{\overline{a}} Q'$ then $P|Q \xrightarrow{\tau} P'|Q'$
- if $P \xrightarrow{a} P'$ and $a, \overline{a} \notin L$ then $P\backslash L \xrightarrow{a} P'\backslash L$

In fact there is a second transition system associated with CCS. We shall call the basic system above the $\longrightarrow$ *transition system*. If P and Q are agents then for $a \in A$, the interpretation of $P \xrightarrow{a} Q$ is that P may evolve into Q by performing the action a which may be observed by the environment, whereas $P \xrightarrow{\tau} Q$ means

that P may evolve autonomously into Q. The second, derived system, which we call the $\Longrightarrow$ *transition system*, has instead the family $\{\stackrel{a}{\Longrightarrow} \mid a \in A \cup \{\varepsilon\}\}$ where $\stackrel{\varepsilon}{\Longrightarrow}$ is the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$, so that $P \stackrel{\varepsilon}{\Longrightarrow} Q$ if P may evolve into Q by a sequence of (possibly zero) autonomous steps, and for $a \in A$, $\stackrel{a}{\Longrightarrow} = \stackrel{\varepsilon}{\Longrightarrow}\stackrel{a}{\longrightarrow}\stackrel{\varepsilon}{\Longrightarrow}$, so that $P \stackrel{a}{\Longrightarrow} Q$ if P may evolve to Q by performing a preceded and succeeded by any number of silent actions.

Several notions of equivalence and approximation between agents may be defined based on action capabilities, each providing some abstract notion of *process* and a rich set of algebraic laws for manipulating agents. In this paper we refer only to one of these: *observation equivalence*. For the definition of this we refer to the standard reference for the Calculus: [Mil89]. An alternative introduction is [Wa87].

2.2 Modal and Temporal Logics

Modal and temporal logics have been found to be appropriate for describing properties of concurrent and communicating systems. They have a vast literature. Survey papers containing extensive bibliographies are [ES89, MP89, St89b], while [St89a] discusses temporal logics for CCS. Here we describe the logics used in this paper.

Hennessy-Milner logic is a propositional modal logic with relativized modal operators. Its formulae are given by

$$A ::= \mathtt{true} \mid \neg A \mid A \wedge A \mid [a]A$$

where a ranges over the set of actions labelling the transition relations of the models of the language. Such a model is of the form $M = (S, \{R_a \mid a \in Act\})$ where S is a set of states (agents) and each R_a is a binary transition relation on states. In this paper all the models we consider are finite subsystems of the CCS $\longrightarrow$ and $\Longrightarrow$ transition systems, so from now on we assume that S is finite. The set $\|A\|$ of states *satisfying* the formula A is defined as follows:

$$\begin{aligned} \|\mathtt{true}\| &= S \\ \|\neg A\| &= S - \|A\| \\ \|A \wedge B\| &= \|A\| \cap \|B\| \\ \|[a]A\| &= \{s \in S \mid \forall s'.\ \text{if } sR_as' \text{ then } s' \in \|A\|\} \end{aligned}$$

We write $s \models A$ if $s \in \| A \|$. We have the derived operators: $\mathtt{false} \equiv \neg\mathtt{true}$, $A \vee B \equiv \neg(\neg A \wedge \neg B)$, and $\langle a\rangle A \equiv \neg[a]\neg A$. We shall use two further derived operators: for $K \subseteq Act$, $[K]A \equiv \bigwedge_{a\in K}[a]A$ and $\langle K\rangle A \equiv \bigvee_{a\in K}\langle a\rangle A$.

Hennessy-Milner logic characterizes the notion of bisimulation equivalence on transition systems. However each formula describes only a finite part of the behaviour of agents, so it is useful to add recursively defined formulae. This leads

to the *modal mu-calculus* [Pr81, Ko83, SW89a] whose formulae are obtained by adding to the description above the clauses

$$A ::= \ldots \mid Z \mid \nu Z.A$$

where Z ranges over a family of propositional variables, and where in $\nu Z.A$ every occurrence of Z in A lies within the scope of an even number of negations[1]. This language is interpreted over models of the form $(S, \{R_a \mid a \in Act\}, V)$ where V is a *valuation* mapping variables to sets of states. The additional semantic clauses are

$$\begin{aligned} \|Z\|_V &= V(Z) \\ \|\nu Z.A\|_V &= \bigcup\{S' \subseteq S \mid S' \subseteq \|A\|_{V[S'/Z]}\} \end{aligned}$$

where $V[S'/Z]$ is the valuation V' which agrees with V except that $V'(Z) = S'$. By the syntactic restriction each $\|A\|_V$ is monotonic in V and thus $\|\nu Z.A\|_V$ is the maximal fixed point of the the function $\lambda S'.\ \|A\|_{V[S'/Z]}$.

Such maximal fixed point formulae express invariance properties. For example

$$s \models \nu Z.(\langle b\rangle \mathtt{true} \wedge [a]Z)$$

expresses that at every point on every path from s consisting solely of a actions, a b action is possible. The dual of the maximal fixed point operator ν is the minimal fixed point operator μ: $\mu Z.A$ abbreviates $\neg\nu Z.\neg A[Z := \neg Z]$ where $[Z := \neg Z]$ denotes syntactic substitution. Minimal fixed point formulae express eventuality properties. For example

$$s \models \mu Z.(\langle b\rangle \mathtt{true} \vee \langle a\rangle Z)$$

expresses that there is some point along some path from s consisting only of a actions at which a b action is possible. The modal mu-calculus is a branching time temporal logic with considerable expressive power. In this paper we consider only simple formulae with at most two fixed point operators. For some discussion of how to formulate properties in a restricted portion of the modal mu-calculus see [Lar89].

2.3 The Concurrency Workbench

The Concurrency Workbench [CPS89a, CPS89b] is a prototype automated system designed to provide machine assistance in carrying out analyses of systems described in CCS. It has been used to analyse communications protocols [Par87]. A user of the Workbench builds an environment of system descriptions of interest at any one time, either interactively or by reading from a file, and then requests that certain analyses be performed on them. Some sixty commands are currently implemented. As a sample we mention that there are commands for:

[1] In general atomic propositions other than `true` would be included. The purpose of the syntactic restriction is to permit a uniform interpretation of formulae.

- Exploring the behaviour of an agent.
- Constructing the state space of an agent.
- Checking for possible deadlocks in an agent.
- Determining whether or not two agents are related by bisimulation and testing equivalences and preorders.
- Minimizing an agent w.r.t. observation-equivalence.
- Determining whether or not an agent satisfies a closed formula of the modal mu-calculus.
- Supporting semi-automatic synthesis of agents solving certain equations.

The Concurrency Workbench is being developed in a joint SERC-funded project between the Laboratory for Foundations of Computer Science in the University of Edinburgh and the Department of Computer Science in the University of Sussex. It is written in Standard ML. New analytical capabilities and a graphical interface are currently being added.

The Workbench is intended to be useful both in understanding and in reasoning about systems. A person grappling with a system may ask certain questions of the Workbench, which without machine assistance would require long, tedious and error-prone calculations for their solution. The answers supplied will enhance understanding, perhaps by suggesting new questions. The usefulness of a particular answer will of course depend on how informative it is to the person receiving it. At present when asked to evaluate an assertion such as $P \models A$ the answer returned by the Workbench is brief: `true` or `false`. The extraction and presentation in a helpful format of information useful in guiding a user towards an understanding of a particular answer will no doubt be of importance in future generations of Workbench-like tools.

3 The algorithms and their representations as CCS agents

In chapter 9 of [Mil80] (and chapter 8 of [Mil89]), Milner describes a means of representing as CCS agents programs expressed in a concurrent imperative programming language. The phrase-by-phrase translation may be viewed as defining the constructs of the programming language as derived combinators of the Calculus. Since the Calculus has a rigorously defined meaning, the translation gives rise to the possibility of formulating precisely within the framework of the Calculus questions concerning the language and programs expressed in it, and of employing the analytical techniques of the Calculus to answer questions so formulated.

As stated in the Introduction, in this paper we are concerned with a family of mutual exclusion algorithms and with the following questions about each of them:

1. Does the algorithm preserve mutual exclusion?

2. Is the algorithm live?

In this section we illustrate the translation using an algorithm due to Dekker and then consider how to formulate the questions of interest. Finally the translations of the other algorithms are given.

It is somewhat tedious to apply the full translation of [Mil80] by hand (although it would be straightforward to mechanize the process). For this reason a slightly *ad hoc* representation of each of the algorithms is used. Each of the *ad hoc* representations is, however, very similar to the corresponding agent obtained from the full translation, and indeed the representations are believed to be observation equivalent in each case.

3.1 Dekker's algorithm

We consider first an algorithm due to Dekker (the formulation here is taken from [PS85]). There are two processes P_1 and P_2, two boolean-valued variables b_1 and b_2 whose initial values are `false`, and a variable k which may take the values 1 and 2 and whose initial value is arbitrary. The i^{th} process ($i = 1, 2$) may be described as follows (where j is the index of the other process):

```
while true do
begin
        ⟨ noncritical section ⟩ ;
        b_i := true ;
        while b_j do
                if k = j then begin
                                        b_i := false ;
                                        while k = j do skip ;
                                        b_i := true
                              end ;
        ⟨ critical section ⟩ ;
        k := j ;
        b_i := false
end ;
```

Note that P_i indicates its wish to execute its critical section by setting b_i to `true`; note also the rôle of the variable k.

In translating into CCS each program variable is represented as a family of CCS agents. A variable v of some type D is represented as a family $\{Vd \mid d \in D\}$

of agents with *access sorts* $\{rd \mid d \in D\} \cup \{\overline{wd} \mid d \in D\}$ ('r' for 'read', 'w' for 'write') defined as follows:

$$Vd \stackrel{\text{def}}{=} \overline{rd}.\, Vd + \Sigma_{e \in D} we.\, Ve \qquad (d \in D)$$

Thus if the current value of v is d, that value may be read by any process which may communicate with Vd by performing the action rd, while a value e may be written to v by any process which may perform the action $\overline{we}$. Thus reading from and writing to a variable are modelled as atomic actions, reflecting a common assumption in reasoning about mutual exclusion algorithms, and one adopted throughout the present paper.

The access sort of each variable is disjoint from that of all other variables[2]. Thus rather than simply 'r' and 'w' we use names unique to each individual variable. For example, the variables b_1, b_2 and k are represented as follows:

$$\begin{aligned}
\mathtt{B_1f} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}.\mathtt{B_1f} + \mathtt{b_1wf}.\mathtt{B_1f} + \mathtt{b_1wt}.\mathtt{B_1t} \\
\mathtt{B_1t} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rt}}.\mathtt{B_1t} + \mathtt{b_1wt}.\mathtt{B_1t} + \mathtt{b_1wf}.\mathtt{B_1f} \\
\\
\mathtt{B_2f} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2rf}}.\mathtt{B_2f} + \mathtt{b_2wf}.\mathtt{B_2f} + \mathtt{b_2wt}.\mathtt{B_2t} \\
\mathtt{B_2t} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2rt}}.\mathtt{B_2t} + \mathtt{b_2wt}.\mathtt{B_2t} + \mathtt{b_2wf}.\mathtt{B_2f} \\
\\
\mathtt{K1} &\stackrel{\text{def}}{=} \overline{\mathtt{kr1}}.\mathtt{K1} + \mathtt{kw1}.\mathtt{K1} + \mathtt{kw2}.\mathtt{K2} \\
\mathtt{K2} &\stackrel{\text{def}}{=} \overline{\mathtt{kr2}}.\mathtt{K2} + \mathtt{kw2}.\mathtt{K2} + \mathtt{kw1}.\mathtt{K1}
\end{aligned}$$

The processes P_1 and P_2 are represented by the following agents (also called) $\mathtt{P_1}$ and $\mathtt{P_2}$:

$$\begin{aligned}
\mathtt{P_1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\mathtt{P_{11}} \\
\mathtt{P_{11}} &\stackrel{\text{def}}{=} \mathtt{b_2rf}.\mathtt{P_{14}} + \mathtt{b_2rt}.\mathtt{P_{12}} \\
\mathtt{P_{12}} &\stackrel{\text{def}}{=} \mathtt{kr1}.\mathtt{P_{11}} + \mathtt{kr2}.\overline{\mathtt{b_1wf}}.\mathtt{P_{13}} \\
\mathtt{P_{13}} &\stackrel{\text{def}}{=} \mathtt{kr1}.\overline{\mathtt{b_1wt}}.\mathtt{P_{11}} + \mathtt{kr2}.\tau.\mathtt{P_{13}} \\
\mathtt{P_{14}} &\stackrel{\text{def}}{=} \mathtt{enter_1}.\mathtt{exit_1}.\overline{\mathtt{kw2}}.\overline{\mathtt{b_1wf}}.\mathtt{P_1} \\
\\
\mathtt{P_2} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}.\mathtt{P_{21}} \\
\mathtt{P_{21}} &\stackrel{\text{def}}{=} \mathtt{b_1rf}.\mathtt{P_{24}} + \mathtt{b_1rt}.\mathtt{P_{22}} \\
\mathtt{P_{22}} &\stackrel{\text{def}}{=} \mathtt{kr2}.\mathtt{P_{21}} + \mathtt{kr1}.\overline{\mathtt{b_2wf}}.\mathtt{P_{23}} \\
\mathtt{P_{23}} &\stackrel{\text{def}}{=} \mathtt{kr2}.\overline{\mathtt{b_2wt}}.\mathtt{P_{21}} + \mathtt{kr1}.\tau.\mathtt{P_{23}} \\
\mathtt{P_{24}} &\stackrel{\text{def}}{=} \mathtt{enter_2}.\mathtt{exit_2}.\overline{\mathtt{kw1}}.\overline{\mathtt{b_2wf}}.\mathtt{P_2}
\end{aligned}$$

[2]In this paper we need not consider the scoping of variables. However as described in [Mil80] (and [Mil89]), the restriction operator of CCS provides an elegant means of handling the scoping of declarations.

Note that the critical section of process P_i is modelled in agent $\mathtt{P_i}$ by a pair of actions, $\mathtt{enter_i}$ and $\mathtt{exit_i}$, representing, respectively, entry to and exit from the critical section. Thus at any point during a computation, P_i will be in its critical section if and only if $\mathtt{P_i}$ may perform the action $\mathtt{exit_i}$. Note also that the noncritical sections of the processes are not directly represented. In particuluar, we do not consider the possiblity that process P_i may terminate in its noncritical section. To model this possiblity we could modify the definition of agent $\mathtt{P_i}$ to:

$$\mathtt{P_i} \stackrel{\text{def}}{=} \tau.\,\mathtt{b_1wt}.\,\mathtt{P_{i1}} + \tau.\,\mathbf{0}$$

Doing this would necessitate some alteration to the treatment of the question of liveness below, though the analysis of mutual exclusion would remain unchanged.

Finally the algorithm is represented by the agent `Dekker` defined by:

$$\mathtt{Dekker} \stackrel{\text{def}}{=} (\mathtt{P_1} \mid \mathtt{P_2} \mid \mathtt{B_1f} \mid \mathtt{B_2f} \mid \mathtt{K1})\backslash \mathtt{L}$$

where $\mathtt{L}$ is the union of the access sort of the variables, and we assume that the initial value of k is 1. The sort of `Dekker` is $\mathtt{K} = \{\mathtt{enter_1}, \mathtt{exit_1}, \mathtt{enter_2}, \mathtt{exit_2}\}$.

We now consider how to express the questions concerning preservation of mutual exclusion and liveness.

3.1.1 Mutual exclusion

To say that the algorithm preserves mutual exclusion means that at no point during any computation may both processes be in their critical sections. As we have seen above, process P_i being in its critical section is modelled as the capability of agent $\mathtt{P_i}$ to perform the action $\mathtt{exit_i}$. Thus to establish that the algorithm preserves mutual exclusion it suffices to show that the agent `Dekker` cannot evolve into a state in which both $\mathtt{exit_1}$ and $\mathtt{exit_2}$ may be performed. We can express this property as an infinite conjunction of formulae of Hennessy-Milner logic as follows:

$$\bigwedge_{\mathtt{s}\in\mathtt{K}^*} [\mathtt{s}]\neg(\langle \mathtt{exit_1}\rangle \mathtt{true} \wedge \langle \mathtt{exit_2}\rangle \mathtt{true})$$

where $\mathtt{s}$ ranges over $\mathtt{K}^*$, the set of finite sequences of actions from $\mathtt{K}$. We can give a fixed point rendition of the property as the modal mu-calculus formula

$$\mathtt{PME} \;\equiv\; \nu \mathtt{Z}.\,(\neg(\langle \mathtt{exit_1}\rangle \mathtt{true} \wedge \langle \mathtt{exit_2}\rangle \mathtt{true}) \wedge [\mathtt{K}]\mathtt{Z})$$

where $[\mathtt{K}]$ is the derived operator defined in Section 2. Thus we say that Dekker's algorithm *preserves mutual exclusion* if

$$\mathtt{Dekker} \models \mathtt{PME}$$

Here we are working in the $\Longrightarrow$ transition system. To work in the $\longrightarrow$ transition system we would replace $[\mathtt{K}]$ by $[\mathtt{K} \cup \{\tau\}]$ in the above formula. In this case it does

not matter in which transition system we work: we find the same answer in both cases.

In addressing the question of mutual exclusion it is convenient to work in the $\Longrightarrow$ transition system. For let `DekkerMin` be such that `DekkerMin` $\approx$ `Dekker` and `DekkerMin` has the minimal number of states among all such agents. Then `Dekker` $\models$ `PME` if and only if `DekkerMin` $\models$ `PME`. This is a consequence of the following fact.

Fact If $P \approx Q$ then for all formulae A of the modal mu-calculus, $P \models A$ if and only if $Q \models A$[3].

The Concurrency Workbench was used both to compute `DekkerMin` and to determine whether or not it satisfies `PME`. Thus the amount of computation required to carry out the model checking was reduced at the expense of the extra computation incurred by the minimization. In treating larger examples this saving would assume greater importance. Even in the present example where the agent `Dekker` has only a few hundred states significant savings were observed.

The above illustrates the method used to study the question of mutual exclusion for each of the six algorithms considered in this paper. In each case the representing agent has sort $\mathtt{K} = \{\mathtt{enter}_1, \mathtt{exit}_1, \mathtt{enter}_2, \mathtt{exit}_2\}$. The results of the analyses are collected in Section 4.

This completes the discussion of the formulation of the question of mutual exclusion. We now consider the more difficult issue of liveness.

3.1.2 Liveness

It is possible to give a number of similar but distinct interpretations of the term *live* as it might be applied to Dekker's algorithm. In some of the papers cited below in which mutual exclusion algorithms are studied, the interpretations of the term are remarkably vague. Moreover, assumptions made in reasoning about the algorithms are often not made explicit. Since we have a rigorous semantics for the programming language, both the interpretation and the assumptions may be formulated precisely. To fix ideas let us say that, roughly speaking, the algorithm is live if whenever at some point in a computation a process P_i requests execution of its critical section, then at some later point P_i will enter its critical section.

The first difficulty we encounter in trying to express within the framework of CCS that the algorithm is live arises from the way in which a request by a process to execute its critical section is represented in the agent `Dekker`. For process P_i performs such a request by setting b_i to `true`, and this action is represented simply as one of many τ-actions: the CCS composition operator leaves no trace of the origin of a particular τ-action. It should be noted that other composition

[3]The converse also holds, under the assumption of image-finiteness of the transition system.

operators have been studied in which the actions arising from synchronizations are distinct from one another. Examples include || of ACP [Ba86] and || of CSP [Ho85]. The approach we adopt here makes use of *probes*. We have already seen an example of their use: the actions $\mathtt{enter_i}$ and $\mathtt{exit_i}$ in the definition of the agent $\mathtt{P_i}$ are probes. By inserting these actions at appropriate points we were able to model in terms of action capabilities what might otherwise be represented as properties of states.

In SCCS [Mil83] it is assumed that actions are drawn from a commutative monoid and the (synchronous) composition operator $\times$ is defined by the rule

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{a \cdot b} P' \times Q'}$$

One interesting possibility would be to modify the definition of the CCS composition operator by allowing actions to be tagged by probes which alone remain after synchronization. The modified rule might be

$$\frac{P \xrightarrow{a\langle p \rangle} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{p} P'|Q'}$$

where $a\langle p \rangle$ represents action a tagged with probe p. Then, e.g., a request by P_i might be modelled by the action $\overline{\mathtt{b_iwt}}\langle \mathtt{req_i} \rangle$. Such a request would then be visible as the performance of the action $\mathtt{req_i}$ by `Dekker`.

The Concurrency Workbench currently supports only the operators of CCS so we must take a different approach here. Recall that $\mathtt{P_i}$ is defined by

$$\mathtt{P_i} \stackrel{\text{def}}{=} \overline{\mathtt{b_iwt}}.\mathtt{P_{i1}}$$

We modify this to

$$\mathtt{P_i} \stackrel{\text{def}}{=} \overline{\mathtt{b_iwt}}.\mathtt{req_i}.\mathtt{P_{i1}}$$

The probes $\mathtt{req_i}$ are not restricted in `Dekker`. Thus in any state of `Dekker` in which an action $\mathtt{req_i}$ is possible, process P_i has 'just' made a request to execute its critical section.

In this way we overcome the first difficulty in expressing liveness, namely that of modelling when a request is made. Now we turn to the delicate question of how to express that such a request is *eventually* met.

We are concerned with temporal properties of states of labelled transition systems. We note that in interpreting temporal logics it is customary to assume that there is at least one transition emanating from each state, if necessary by grafting loops labelled by imaginary labels onto offending states. The $\Longrightarrow$ transition system is trivially a *total* system in this sense, since for any agent Q, $Q \stackrel{\varepsilon}{\Longrightarrow} Q$. The $\longrightarrow$ transition system is not in general total, though by examining the agents

$\mathtt{P_i}$ it may be checked that the subsystems we consider are all total (this may be checked using the Workbench).

A more refined interpretation of what might be meant by liveness of the algorithm is the following.

> Liveness, Interpretation 1 The algorithm is *live* if whenever at some point in a computation a process P_i requests execution of its critical section, then in any continuation of the computation from that point in which between them the processes execute an infinite number of critial sections, P_i performs its critical section at least once.

In terms of the CCS representation of the algorithm this means that there is no infinite path π through the `Dekker` transition system containing an infinite number of visible actions from `K` which is of the form $\pi = s \cdot \langle \mathtt{req_i} \rangle \cdot \pi'$ where s is finite and $\mathtt{enter_i}$ does not occur in π'. We shall consider an alternative interpretation below but first we examine Interpretation 1 in detail.

We can express the liveness property by interpreting the following modal mu-calculus formula over the $\Longrightarrow$ transition system:

$$\mathtt{Live} \equiv \mathtt{Live_1} \wedge \mathtt{Live_2}$$

where

$$\mathtt{Live_i} \equiv \nu \mathtt{Z}.\,([\mathtt{req_i}]\mu \mathtt{Y}.\,(\langle \mathtt{exit_i} \rangle \mathtt{true} \vee [\mathtt{K}]\mathtt{Y}) \wedge [\mathtt{K}]\mathtt{Z})$$

To see this consider the `Dekker` $\Longrightarrow$ transition system and note that for any `Q`

$$\mu \mathtt{Y}.\,(\mathtt{Q} \vee [\mathtt{K}]\mathtt{Y})$$

expresses that along every path containing infinitely many visible actions `Q` holds eventually, while

$$\nu \mathtt{Z}.\,(\mathtt{Q} \wedge [\mathtt{K}]\mathtt{Z})$$

expresses that along every such path `Q` holds invariantly.

Again we economize on computational effort by minimizing `Dekker` w.r.t $\approx$ and then determining whether or not the minimized agent satisfies `Live`. The results obtained by following this procedure for each of the six algorithms are collected in Section 4. Two of the algorithms, Knuth's and Peterson's, are live under this interpretation.

An alternative to Interpretation 1, involving a different treatment of τ-loops in the transition systems, is possible. The significance of such loops is clearly seen by replacing [K] by $[\mathtt{K} \cup \{\tau\}]$ in the definition of $\mathtt{Live_i}$, and interpreting the modified formula `Live` on the $\longrightarrow$ transition systems: we find using the Workbench that each of the six agents representing the algorithms does *not* satisfy it.

By examining the agents $\mathtt{P_i}$ it is apparent that *some* of the τ-loops in `Dekker` arise from infinite sequences of reading and writing of variables by a single process. Indeed this phenomenon is common to all six algorithms. Note that the τ-loops

are not somehow *introduced* by the translation into CCS. Rather their presence reflects the faithfulness of the translation.

Consider now the following alternative to Interpretation 1:

> Liveness, Interpretation 2 The algorithm is *live* if whenever at any point in any computation a process P_i requests execution of its critical section, then in any continuation of the computation from that point there is a point at which P_i enters its critical section.

From the above discussion we see that, given the semantics of the language, in order to establish that any of the algorithms is live under Interpretation 2 we must make some additional assumption. This is an important point which, as mentioned earlier, is often not handled with the precision necessary to make clear exactly what is established by certain arguments.

One characteristic of the trivial τ-loops arising from repeated reading and writing of variables by one of the processes is that the other process is excluded from an infinite computation of the system. It is natural to ask whether it is only the existence of such *unfair* computations which prevents any of the algorithms from being live under Interpretation 2. In other words, if we consider as *admissible* only those paths through each of the transition systems representing computations to which each process contributes infinitely often, an assumption which can be viewed as refining the semantics of the language, we may ask which of the algorithms is live under Interpretation 2.

The assumption that each process contributes infinitely often to each infinite computation is just one of the many *fairness assumptions* which have been studied (see e.g. [Fr86]). Reasoning under such assumptions exposes remarkable subtleties in some of the algorithms. τ-loops in the *extended transition systems*, the transition systems augmented with some means of delineating the admissible paths through them, can in some cases arise from a delicate pattern of mutual blocking by the two processes, resulting in *starvation* of them both. Moreover it is possible for an algorithm to be free from such starvation without being live under Interpretation 2 (see Knuth's discussion of Dijkstra's algorithm [Kn66]).

An alternative to the above idea of interpreting formulae over extended transition systems incorporating fairness assumptions is to interpret formulae incorporating fairness assumptions over transition systems. Such formulae would be of the form

$$\mathtt{F} \Rightarrow \mathtt{P}$$

where F expresses a fairness assumption and P a property of interest. Below we consider this in more detail in the case of linear time logics, where a formula is satisfied by a state if it is satisfied by each path from the state (for a discussion of branching time see [EL87].)

At the time of writing no linear time logic model checker is incorporated in the Workbench so no automated analysis has been possible. However we close this

discussion by highlighting another apparent difficulty and indicating how it may be overcome.

We saw above that some modification to the representations of the algorithms was required in order to make visible a request by a process to execute its critical section. We face a similar problem in expressing that a process contributes infinitely often to a computation. We may overcome this difficulty as we did the earlier one: by using probes. Perhaps the most attractive solution is to use the modified composition operator described above and to tag each action of $\mathtt{P_i}$ representing a read of write to a variable with a unique probe[4]. We could then express that $\mathtt{P_i}$ contributes infinitely often to a computation by saying that each of its probes is infinitely often *not* a possible action of `Dekker`. For $\mathtt{P_i}$ does *not* contribute infinitely often to a computation if and only if one of its actions is always possible from some point in that computation.

We may then express the liveness of the algorithm under the fairness assumption by letting $\mathtt{K_i}$ be the set of probes of $\mathtt{P_i}$ and defining

$$\mathtt{FairLive} \equiv \mathtt{FairLive_1} \wedge \mathtt{FairLive_2}$$

where

$$\mathtt{FairLive_i} \equiv \bigwedge_{\mathtt{a} \in \mathtt{K_i}} \mathtt{GF[a]false} \Rightarrow \mathtt{G(\langle req_i \rangle true} \Rightarrow \mathtt{F\langle exit_i \rangle true)}$$

where `G` and `F` are the linear time *always* and *eventually* operators.

It is anticipated that a model checking algorithm for a linear time mu-calculus with relativized *next* operators (a logic subsuming many common linear time logics and in which the above formula may be rendered) will soon be incorporated into the Workbench (see [SW89b]).

This completes the discussion of the formulation of the questions of interest. We now describe the other five algorithms and their representations.

3.2 Dijkstra's algorithm

In Dijkstra's algorithm [Dij65] it is assumed that there are $n \geq 2$ processes, global variables

$$\begin{aligned} b, c &: \textbf{array } [1..n] \textbf{ of boolean} \\ k &: \text{integer} \end{aligned}$$

and that initially all components of b and of c have value `true` and the value of k is one of $1, 2, \ldots, n$. The i^{th} process ($i = 1, 2, \ldots, n$) may be represented as follows:

[4]By a more careful analysis of the agents we could reduce the number of probes required.

```
var j : integer ;
while true do
begin
        ⟨ noncritical section ⟩ ;
        b[i] := false ;
        L_i : if k ≠ i then begin
                                    c[i] := true ;
                                    if b[k] then k := i ;
                                    goto L_i
                              end ;
              else begin
                         c[i] := false ;
                         for j := 1 to n do
                             if (j ≠ i and ¬(c[j])) then goto L_i
                   end ;
        ⟨ critical section ⟩ ;
        c[i] := true ;
        b[i] := true
end ;
```

Note that process P_i indicates its wish to execute its critical section by setting $b[i]$ to `false`. In light of later developments this algorithm seems remarkably complicated. It was, however, the first published solution to a mutual exclusion problem.

We give here the CCS representation in the case $n = 2$. In order to model faithfully the assumption that any read of or write to a program variable is an atomic action, we must use a more complicated representation of the variable k than in the previous cases. The reason for this is that we must ensure that if in the course of executing the statement

$$\textbf{if } b[k] \textbf{ then } k := i$$

the i^{th} process has read the variable k but not yet read the variable $b[k]$, then no other process may access (and in particular change the value of) the variable k. The following representation, suggested by Robin Milner, ensures this:

$$\begin{aligned}
\mathtt{K1} &\stackrel{\text{def}}{=} \overline{\mathtt{kr1}}.\mathtt{K1} + \mathtt{kw1}.\mathtt{K1} + \mathtt{kw2}.\mathtt{K2} \\
&\quad + \mathtt{get}.(\overline{\mathtt{k_1r1}}.\mathtt{put}.\mathtt{K1} + \overline{\mathtt{k_2r1}}.\mathtt{put}.\mathtt{K1}) \\
\mathtt{K2} &\stackrel{\text{def}}{=} \overline{\mathtt{kr2}}.\mathtt{K2} + \mathtt{kw2}.\mathtt{K2} + \mathtt{kw1}.\mathtt{K1} \\
&\quad + \mathtt{get}.(\overline{\mathtt{k_1r2}}.\mathtt{put}.\mathtt{K2} + \overline{\mathtt{k_2r2}}.\mathtt{put}.\mathtt{K2})
\end{aligned}$$

The variables b_1 and b_2 are represented as in `Dekker` and c_1 and c_2 are represented similarly as agents $\mathtt{C_1f}$ and $\mathtt{C_1t}$, and $\mathtt{C_2f}$ and $\mathtt{C_2t}$. The algorithm is then represented by the agent `Dijkstra` defined by

$$\mathtt{Dijkstra} \stackrel{\text{def}}{=} (\mathtt{P_1}|\mathtt{P_2}|\mathtt{B_1t}|\mathtt{B_2t}|\mathtt{C_1t}|\mathtt{C_2t}|\mathtt{K1})\backslash\mathtt{L}$$

where L is the union of the sorts of the variables and

$$
\begin{aligned}
\mathtt{P}_1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wf}}.\,\mathtt{req}_1.\,\mathtt{P}_{11}\\
\mathtt{P}_{11} &\stackrel{\text{def}}{=} \mathtt{kr1}.\,\mathtt{P}_{13} + \mathtt{kr2}.\,\overline{\mathtt{c_1wt}}.\,\mathtt{P}_{12}\\
\mathtt{P}_{12} &\stackrel{\text{def}}{=} \overline{\mathtt{get}}.\,(\mathtt{k_1r1}.\,(\mathtt{b1rt}.\,\overline{\mathtt{put}}.\,\overline{\mathtt{kw1}}.\,\mathtt{P}_{11} + \mathtt{b_1rf}.\,\overline{\mathtt{put}}.\,\mathtt{P}_{11})\\
&\qquad + \mathtt{k_1r2}.\,(\mathtt{b_2rt}.\,\overline{\mathtt{put}}.\,\overline{\mathtt{kw1}}.\,\mathtt{P}_{11} + \mathtt{b_2rf}.\,\overline{\mathtt{put}}.\,\mathtt{P}_{11}))\\
\mathtt{P}_{13} &\stackrel{\text{def}}{=} \overline{\mathtt{c_1wf}}.\,(\mathtt{c_2rf}.\,\mathtt{P}_{11} + \mathtt{c_2rt}.\,\mathtt{P}_{14})\\
\mathtt{P}_{14} &\stackrel{\text{def}}{=} \mathtt{enter}_1.\,\mathtt{exit}_1.\,\overline{\mathtt{c_1wt}}.\,\overline{\mathtt{b_1wt}}.\,\mathtt{P}_1
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{P}_2 &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wf}}.\,\mathtt{req}_2.\,\mathtt{P}_{21}\\
\mathtt{P}_{21} &\stackrel{\text{def}}{=} \mathtt{kr2}.\,\mathtt{P}_{23} + \mathtt{kr1}.\,\overline{\mathtt{c_2wt}}.\,\mathtt{P}_{22}\\
\mathtt{P}_{22} &\stackrel{\text{def}}{=} \overline{\mathtt{get}}.\,(\mathtt{k_2r2}.\,(\mathtt{b2rt}.\,\overline{\mathtt{put}}.\,\overline{\mathtt{kw2}}.\,\mathtt{P}_{21} + \mathtt{b_2rf}.\,\overline{\mathtt{put}}.\,\mathtt{P}_{21})\\
&\qquad + \mathtt{k_2r1}.\,(\mathtt{b_1rt}.\,\overline{\mathtt{put}}.\,\overline{\mathtt{kw2}}.\,\mathtt{P}_{21} + \mathtt{b_1rf}.\,\overline{\mathtt{put}}.\,\mathtt{P}_{21}))\\
\mathtt{P}_{23} &\stackrel{\text{def}}{=} \overline{\mathtt{c_2wf}}.\,(\mathtt{c_1rf}.\,\mathtt{P}_{21} + \mathtt{c_1rt}.\,\mathtt{P}_{24})\\
\mathtt{P}_{24} &\stackrel{\text{def}}{=} \mathtt{enter}_2.\,\mathtt{exit}_2.\,\overline{\mathtt{c_2wt}}.\,\overline{\mathtt{b_2wt}}.\,\mathtt{P}_2
\end{aligned}
$$

3.3 Hyman's algorithm

Hyman published [Hy66] a 'simplification' of Dijkstra's algorithm in the case of two processes. The following formulation is taken (with some change of notation) from [PS85]. There are two processes P_1 and P_2, two boolean-valued variables b_1 and b_2 with initial value `false`, and a variable k which may assume the values 1 and 2 and whose initial value is arbitrary. The i^{th} process ($i = 1, 2$) may be described as follows (where j is the index of the other process):

while true do
begin
⟨ noncritical section ⟩ ;
b_i := **true** ;
while $k \neq i$ **do begin**
while b_j **do skip** ;
$k := i$
end ;
⟨ critical section ⟩ ;
b_i := **false**
end ;

As the reader can perhaps see, Hyman's simplification is not entirely satisfactory. The CCS representation is the agent `Hyman` defined as follows:

$$\mathtt{Hyman} \stackrel{\text{def}}{=} (\mathtt{P_1}|\mathtt{P_2}|\mathtt{B_1f}|\mathtt{B_2f}|\mathtt{K1})\backslash\mathtt{L}$$

where $\mathtt{B_1f}$, $\mathtt{B_2f}$ and $\mathtt{K1}$ are as in the definition of Dekker, L is the union of the access sorts of the variables, and

$$\begin{aligned}
\mathtt{P_1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.\,\mathtt{req_1}.\,\mathtt{P_{11}} \\
\mathtt{P_{11}} &\stackrel{\text{def}}{=} \mathtt{kr1}.\,\mathtt{P_{13}} + \mathtt{kr2}.\,\mathtt{P_{12}} \\
\mathtt{P_{12}} &\stackrel{\text{def}}{=} \mathtt{b_2rf}.\,\overline{\mathtt{kw1}}.\,\mathtt{P_{13}} + \mathtt{b_2rt}.\,\tau.\,\mathtt{P_{12}} \\
\mathtt{P_{13}} &\stackrel{\text{def}}{=} \mathtt{enter_1}.\,\mathtt{exit_1}.\,\overline{\mathtt{b_1wf}}.\,\mathtt{P_1}
\end{aligned}$$

$$\begin{aligned}
\mathtt{P_2} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}.\,\mathtt{req_2}.\,\mathtt{P_{21}} \\
\mathtt{P_{21}} &\stackrel{\text{def}}{=} \mathtt{kr2}.\,\mathtt{P_{23}} + \mathtt{kr1}.\,\mathtt{P_{22}} \\
\mathtt{P_{22}} &\stackrel{\text{def}}{=} \mathtt{b_1rf}.\,\overline{\mathtt{kw2}}.\,\mathtt{P_{23}} + \mathtt{b_1rt}.\,\tau.\,\mathtt{P_{22}} \\
\mathtt{P_{23}} &\stackrel{\text{def}}{=} \mathtt{enter_2}.\,\mathtt{exit_2}.\,\overline{\mathtt{b_2wf}}.\,\mathtt{P_2}
\end{aligned}$$

3.4 Knuth's algorithm

Following the publication of Dijkstra's algorithm [Dij65] and of Hyman's letter [Hy66], Knuth [Kn66] drew attention to some of the shortcomings of Hyman's simplification and to the fact that although Dijkstra's algorithm is free from starvation it is not live, and suggested the following alternative.

We consider only the case when there are 2 processes. There are then 2 variables c_1, c_2 which may take the values $0, 1, 2$ and whose initial values are 0, and a variable k which may take the values $1, 2$ and whose initial value is arbitrary. The i^{th} process ($i = 1, 2$) may be described as follows (where j is the index of the other process):

while true do
begin
⟨ noncritical section ⟩ ;
L_0: $c_i := 1$;
L_1: **if** $k = i$ **then goto** L_2 ;
if $c_j \neq 0$ **then goto** L_1 ;
L_2: $c_i := 2$;
if $c_j = 2$ **then goto** L_0 ;
$k := i$;
⟨ critical section ⟩ ;
$k := j$;
$c_i := 0$;
end ;

This algorithm is remarkably intricate (see [Kn66] for an interesting discussion). The CCS representation, assuming the initial value of k to be 1, is the agent Knuth defined as follows:

$$\mathtt{Knuth} \stackrel{\text{def}}{=} (\mathtt{P_1} \mid \mathtt{P_2} \mid \mathtt{K1} \mid \mathtt{C_10} \mid \mathtt{C_20})\backslash\mathtt{L}$$

where L is the union of the sorts of the variables and

$$
\begin{aligned}
\mathtt{C_1 0} &\stackrel{\text{def}}{=} \mathtt{c_1 w0.C_1 0} + \mathtt{c_1 w1.C_1 1} + \mathtt{c_1 w2.C_1 2} + \overline{\mathtt{c_1 r0}}\mathtt{.C_1 0}\\
\mathtt{C_1 1} &\stackrel{\text{def}}{=} \mathtt{c_1 w0.C_1 0} + \mathtt{c_1 w1.C_1 1} + \mathtt{c_1 w2.C_1 2} + \overline{\mathtt{c_1 r1}}\mathtt{.C_1 1}\\
\mathtt{C_1 2} &\stackrel{\text{def}}{=} \mathtt{c_1 w0.C_1 0} + \mathtt{c_1 w1.C_1 1} + \mathtt{c_1 w2.C_1 2} + \overline{\mathtt{c_1 r2}}\mathtt{.C_1 2}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{C_2 0} &\stackrel{\text{def}}{=} \mathtt{c_2 w0.C_2 0} + \mathtt{c_2 w1.C_2 1} + \mathtt{c_2 w2.C_2 2} + \overline{\mathtt{c_2 r0}}\mathtt{.C_2 0}\\
\mathtt{C_2 1} &\stackrel{\text{def}}{=} \mathtt{c_2 w0.C_2 0} + \mathtt{c_2 w1.C_2 1} + \mathtt{c_2 w2.C_2 2} + \overline{\mathtt{c_2 r1}}\mathtt{.C_2 1}\\
\mathtt{C_2 2} &\stackrel{\text{def}}{=} \mathtt{c_2 w0.C_2 0} + \mathtt{c_2 w1.C_2 1} + \mathtt{c_2 w2.C_2 2} + \overline{\mathtt{c_2 r2}}\mathtt{.C_2 2}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{P_1} &\stackrel{\text{def}}{=} \overline{\mathtt{c_1 w1}}\mathtt{.req_1.P_{11}}\\
\mathtt{P_{11}} &\stackrel{\text{def}}{=} \mathtt{kr1.P_{13}} + \mathtt{kr2.P_{12}}\\
\mathtt{P_{12}} &\stackrel{\text{def}}{=} \mathtt{c_2 r0.P_{13}} + \mathtt{c_2 r1.P_{11}} + \mathtt{c_2 r2.P_{11}}\\
\mathtt{P_{13}} &\stackrel{\text{def}}{=} \overline{\mathtt{c_1 w2}}\mathtt{.P_{14}}\\
\mathtt{P_{14}} &\stackrel{\text{def}}{=} \mathtt{c_2 r0.P_{15}} + \mathtt{c_2 r1.P_{15}} + \mathtt{c_2 r2.P_{16}}\\
\mathtt{P_{15}} &\stackrel{\text{def}}{=} \overline{\mathtt{kw1}}\mathtt{.enter_1.exit_1.}\overline{\mathtt{kw2}}\mathtt{.}\overline{\mathtt{c_1 w0}}\mathtt{.P_1}\\
\mathtt{P_{16}} &\stackrel{\text{def}}{=} \overline{\mathtt{c_1 w1}}\mathtt{.P_{11}}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{P_2} &\stackrel{\text{def}}{=} \overline{\mathtt{c_2 w1}}\mathtt{.req_2.P_{21}}\\
\mathtt{P_{21}} &\stackrel{\text{def}}{=} \mathtt{kr2.P_{23}} + \mathtt{kr1.P_{22}}\\
\mathtt{P_{22}} &\stackrel{\text{def}}{=} \mathtt{c_1 r0.P_{23}} + \mathtt{c_1 r1.P_{21}} + \mathtt{c_1 r2.P_{21}}\\
\mathtt{P_{23}} &\stackrel{\text{def}}{=} \overline{\mathtt{c_2 w2}}\mathtt{.P_{24}}\\
\mathtt{P_{24}} &\stackrel{\text{def}}{=} \mathtt{c_1 r0.P_{25}} + \mathtt{c_1 r1.P_{25}} + \mathtt{c_1 r2.P_{26}}\\
\mathtt{P_{25}} &\stackrel{\text{def}}{=} \overline{\mathtt{kw2}}\mathtt{.enter_2.exit_2.}\overline{\mathtt{kw1}}\mathtt{.}\overline{\mathtt{c_2 w0}}\mathtt{.P_2}\\
\mathtt{P_{26}} &\stackrel{\text{def}}{=} \overline{\mathtt{c_2 w1}}\mathtt{.P_{21}}
\end{aligned}
$$

Note in the above the presence of the agents $\mathtt{P_{i6}}$ and the way in which the statement **goto** L_0 is represented. The reason for this choice is that only the first $\overline{\mathtt{c_i w1}}$ action (setting c_i to 1) is considered as signifying the initiation of an attempt by process i to execute its critical section.

3.5 Peterson's algorithm

In Peterson's algorithm [PS85] there are two processes P_1 and P_2, two boolean-valued variables b_1 and b_2 each having initial value `false`, and one variable k which may take the values 1 and 2 and whose initial value is arbitrary. The i^{th} process ($i = 1, 2$) may be described as follows (where j is the index of the other process):

while true do
begin
⟨ noncritical section ⟩ ;
b_i := **true** ;
$k := j$;
while (b_j **and** $k = j$) **do skip** ;
⟨ critical section ⟩ ;
b_i := **false**
end;

Peterson's algorithm is an attractively simple refinement of ideas from some of the earlier algorithms. The CCS representation of this algorithm (assuming the initial value of k to be 1) is the agent `Peterson` below:

$$\texttt{Peterson} \stackrel{\text{def}}{=} (\texttt{P}_1|\texttt{P}_2|\texttt{B}_1\texttt{f}|\texttt{B}_2\texttt{f}|\texttt{K1})\backslash\texttt{L}$$

where where $\texttt{B}_1\texttt{f}$, $\texttt{B}_2\texttt{f}$ and `K1` are as in the definition of `Dekker`, `L` is the union of the access sorts of the variables, and

$$\begin{aligned}
\texttt{P}_1 &\stackrel{\text{def}}{=} \overline{\texttt{b}_1\texttt{wt}}.\texttt{req}_1.\overline{\texttt{kw2}}.\texttt{P}_{11} \\
\texttt{P}_{11} &\stackrel{\text{def}}{=} \texttt{b}_2\texttt{rf}.\texttt{P}_{12} + \texttt{b}_2\texttt{rt}.(\texttt{kr2}.\tau,.\texttt{P}_{11} + \texttt{kr1}.\texttt{P}_{12}) \\
\texttt{P}_{12} &\stackrel{\text{def}}{=} \texttt{enter}_1.\texttt{exit}_1.\overline{\texttt{b}_1\texttt{wf}}.\texttt{P}_1
\end{aligned}$$

$$\begin{aligned}
\texttt{P}_2 &\stackrel{\text{def}}{=} \overline{\texttt{b}_2\texttt{wt}}.\texttt{req}_2.\overline{\texttt{kw1}}.\texttt{P}_{21} \\
\texttt{P}_{21} &\stackrel{\text{def}}{=} \texttt{b}_1\texttt{rf}.\texttt{P}_{22} + \texttt{b}_1\texttt{rt}.(\texttt{kr1}.\tau.\texttt{P}_{21} + \texttt{kr2}.\texttt{P}_{22}) \\
\texttt{P}_{22} &\stackrel{\text{def}}{=} \texttt{enter}_2.\texttt{exit}_2.\overline{\texttt{b}_2\texttt{wf}}.\texttt{P}_2
\end{aligned}$$

3.6 Lamport's algorithm

In Lamport's 'one-bit algorithm' [Lam86] it is assumed that there are $n \geq 2$ processes and n boolean-valued variables each with initial value `false`. The i^{th} process ($i = 1, 2, \ldots, n$) may be described as follows:

```
var j : integer ;
while true do
begin
        ⟨ noncritical section ⟩ ;
        L_i : b_i := true ;
        for j := 1 to i − 1 do
            if b_j then begin
                                b_i := false ;
                                while b_j do skip ;
                                goto L_i
                        end ;
        for j := i + 1 to n do
            while b_j do skip ;
        ⟨ critical section ⟩ ;
        b_i := false
end ;
```

Notice that this algorithm is not symmetric. The CCS representation in the case $n = 2$ is the agent Lamport defined as follows:

$$\texttt{Lamport} \stackrel{\text{def}}{=} (\texttt{P}_1|\texttt{P}_2|\texttt{B}_1\texttt{f}|\texttt{B}_2\texttt{f})\backslash\texttt{L}$$

where $\texttt{B}_1\texttt{f}$ and $\texttt{B}_2\texttt{f}$ are as in the definition of Dekker, L is the union of the sorts of the variables, and

$$\begin{aligned}
\texttt{P}_1 &\stackrel{\text{def}}{=} \overline{\texttt{b}_1\texttt{wt}}.\texttt{req}_1.\texttt{P}_{11}\\
\texttt{P}_{11} &\stackrel{\text{def}}{=} \texttt{b}_2\texttt{rf}.\texttt{P}_{12} + \texttt{b}_2\texttt{rt}.\tau.\texttt{P}_{11}\\
\texttt{P}_{12} &\stackrel{\text{def}}{=} \texttt{enter}_1.\texttt{exit}_1.\overline{\texttt{b}_1\texttt{wf}}.\texttt{P}_1
\end{aligned}$$

$$\begin{aligned}
\texttt{P}_2 &\stackrel{\text{def}}{=} \overline{\texttt{b}_2\texttt{wt}}.\texttt{req}_2.\texttt{P}_{21}\\
\texttt{P}_{21} &\stackrel{\text{def}}{=} \texttt{b}_1\texttt{rf}.\texttt{P}_{23} + \texttt{b}_1\texttt{rt}.\overline{\texttt{b}_2\texttt{wf}}.\texttt{P}_{22}\\
\texttt{P}_{22} &\stackrel{\text{def}}{=} \texttt{b}_1\texttt{rf}.\overline{\texttt{b}_2\texttt{wt}}.\texttt{P}_{21} + \texttt{b}_1\texttt{rt}.\tau.\texttt{P}_{23}\\
\texttt{P}_{23} &\stackrel{\text{def}}{=} \texttt{enter}_2.\texttt{exit}_2.\overline{\texttt{b}_2\texttt{wf}}.\texttt{P}_2
\end{aligned}$$

4 Experimental results, discussion and some related work

The table below summarizes the results obtained by following for each of the six algorithms the procedure described in detail for Dekker's algorithm in Section 3.

	Preserves Mutual Exclusion	Is Live (Interpretation 1)
Dekker	√	×
Dijkstra	√	×
Hyman	×	×
Knuth	√	√
Peterson	√	√
Lamport	√	×

Under Interpretation 1, Lamport's algorithm is live for process P_1 but not for process P_2, in the sense that

$$\texttt{Lamport} \models \texttt{Live}_1 \wedge \neg\texttt{Live}_2$$

As mentioned in Section 3, at the time of writing it has not been possible to determine automatically which of the algorithms is live under Interpretation 2 because the Concurrency Workbench presently supports model checking only for the modal mu-calculus.

The fact that each of the algorithms can be faithfully modelled as a finite-state agent was important as currently only such agents may be analysed using the Workbench. However within CCS it is quite possible to describe and reason about systems which are not finite-state. Much work remains to be done in developing implementable techniques for reasoning about wider classes of systems (see e.g. [BBK87], [JP89]).

The mutual exclusion algorithms were selected for study partly because although they exhibit fairly intricate and subtle behaviour, their representations are nonetheless small enough to be amenable to analysis through fairly unsophisticated application of an existing tool. (The largest state space is that of the agent `Dijkstra` with only 479 states.) The usefulness of calculi such as CCS, supported by suitable automated tools, in aiding reasoning about abstractions of much larger systems deserves further study.

Automated tools embodying calculi of concurrency have been developed at INRIA (Sophia-Antipolis) by a group led by Gérard Berry. These tools, such as AUTO and ECRINS, have been used to analyse a number of systems. Further related work has been carried out by Kim Larsen in Aalborg. The existing links between these researchers and the developers of the Concurrency Workbench will be enhanced through the ERPSIT Basic Research Action CONCUR scheduled to begin in autumn 1989.

The earliest references to temporal logic model checking of finite-state systems are [CE81] and [QS81]. Edmund Clarke, in particular, has been a strong advocate of this approach (see e.g. [CG87]). A large number of systems devoted to model checking of finite-state systems now exist (see e.g. [Gren]), and many have been used successfully to analyse hardware and communications protocols.

Acknowledgments

I am indebted to Joachim Parrow, Rance Cleaveland and Bernhard Steffen, the authors of the Concurrency Workbench, and to Jo Blishen who ported the Workbench to the New Jersey ML compiler. I would like also to thank Robin Milner, Kevin Mitchell and Colin Stirling for discussions about this work, and the referees for their helpful criticisms. Part of this research was carried out while the author was supported by a grant from the Venture Research Unit of BP.

References

[Ba86] Baeten, J., *Procesalgebra*, Kluwer programmatuurkunde (1986).
[BBK87] Baeten, J., Bergstra, J., and Klop, J.W., *Decidability of bisimulation equivalence for processes generating context-free languages*, in Springer LNCS 299 (1987).
[CE8i] Clarke, E., and Emerson, E.A., *Design and synthesis of synchronization skeletons using branching time temporal logic*, in Springer LNCS 131 (1981).
[CG87] Clarke, E., and Grümberg, O., *Research on automatic verification of finite-state concurrent systems*, Ann. Rev. Comput. Sci. 2 (1987).
[CPS88] Cleaveland, R., Parrow, J., and Steffen, B., *The Concurrency Workbench: Operating Instructions*, University of Edinburgh report (1988).
[CPS89a] Cleaveland, R., Parrow, J., and Steffen, B., *The Concurrency Workbench*, to appear in [Gren] (1989).
[CPS89a] Cleaveland, R., Parrow, J., and Steffen, B., *A Semantics based Verification Tool for Finite State Systems*, to appear in Proc. Ninth Int. Symp. on Protocol Specification, Testing and Verification, North Holland (1989).
[Dij65] Dijkstra, E.W., *Solution of a Problem in Concurrent Programming Control*, Comm. A.C.M. 8/9 (1965).
[EL86] Emerson, E.A., and Lei, C-L., *Efficient Model Checking in Fragments of the Propositional Mu-Calculus*, in Proc. 2nd Annual Symposium on Logic in Computer Science (1986).
[EL87] Emerson, E.A., and Lei, C-L., *Modalities for model checking: branching time strikes back*, Sci. Comput. Prog. 6 (1987).
[ES89] Emerson, E.A., and Srinivasan, J., *Braching time temporal logic*, in Springer LNCS 354 (1989).
[Fr86] Francez, N., Fairness, Springer (1986).
[Gren] Proceedings of Workshop on Automatic Verification Methods for Finite State Systems, Grenoble 1989, to appear in Springer LNCS series.
[Ho85] Hoare, C.A.R., Communicating Sequential Processes, Prentice-Hall (1985).
[Hy66] Hyman, H., *Comments on a Problem in Concurrent Programming Control*, Comm. A.C.M. 9/1 (1966).
[JP89] Jonsson, B., and Parrow, J., *Deciding Bisimulation Equivalences for a Class*

of Non-Finite-State Programs, in Springer LNCS 349 (1989).
[Kn66] Knuth, D.E., *Additional Comments on a Problem in Concurrent Programming Control*, Comm. A.C.M. 9/5 (1966).
[Ko83] Kozen, D., *Results on the Propositional μ-Calculus*, Theoretical Computer Science 27 (1983).
[Lam86] Lamport, L., *The Mutual Exclusion Problem Part II — Statement and Solutions*, J.A.C.M. 33/2 (1986).
[Lar89] Larsen, K., *Proof Systems for Hennessy-Milner logic with recursion*, to appear in Information and Computation (1989).
[MP89] Manna, Z., and Pnueli A., *The anchored version of the temporal framework*, in Springer LNCS 354 (1989).
[Mil80] Milner, R., A Calculus of Communicating Systems, Springer-Verlag (1980); available as University of Edinburgh report ECS-LFCS-87-7.
[Mil83] Milner, R., *Calculi for Synchrony and Asynchrony*, Theoretical Computer Science 25 (1983).
[Mil89] Milner, R., Communication and Concurrency, Prentice-Hall (1989).
[Par87] Parrow, J., *Verifying a CSMA/CD-Protocol with CCS*, University of Edinburgh report ECS-LFCS-87-18 (1987).
[PS85] Peterson, J.L. and Silberschatz, A., Operating System Concepts, 2nd ed., Addison Wesley (1985).
[Pr81] Pratt, V., *A decidable mu-calculus*, in Proc. 22nd ACM FOCS (1981).
[QS81] Queille J., and Sifakis, J., *Specification and Verification of Concurrent Systems in CESAR*, in Springer LNCS 137 (1981).
[St87] Stirling, C., *Modal Logics for Communicating Systems*, Theoretical Computer Science 49 (1987).
[St89a] Stirling, C., *Temporal Logics for CCS*, in Springer LNCS 354 (1989).
[St89b] Stirling, C., *Modal and Temporal Logics*, chapter to appear in Handbook of Logic in Computer Science, OUP (1989).
[SW89a] Stirling, C., and Walker, D., *Local Model Checking in the Modal Mu-Calculus*, in Springer LNCS 351 (1989).
[SW89b] Stirling, C., and Walker, D., *CCS, Liveness, and local model checking in the linear time mu-calculus*, to appear in [Gren] (1989).
[Wa87] Walker, D., *Introduction to a Calculus of Communicating Systems*, University of Edinburgh report, ECS-LFCS-87-22 (1987).