Trapping Mutual Exclusion in the Box Calculus

by

Javier Esparza and Glenn Bruns
Trapping Mutual Exclusion in the Box Calculus

Javier Esparza and Glenn Bruns
Laboratory for Foundations of Computer Science
University of Edinburgh

Abstract

The box calculus is a process algebra with a simple Petri net semantics. We show that it allows verification techniques from process algebra and Petri nets to be combined. This is done by proving some properties of mutual exclusion algorithms.

1 Introduction

The box calculus is a process algebra with a simple, compositional Petri net semantics [1,2], that serves as the foundation of a concurrent programming notation [4]. It is mainly inspired by CCS[19], and has been influenced by the work of Boudol and Castellani [7], Degano, De Nicola, and Montanari [11], Olderog [23] and others on net semantics of process algebras. Here we show, through a study of Dekker's mutual exclusion algorithm, that the box calculus provides a framework in which the verification techniques of Petri nets and process algebra can be smoothly integrated. From process algebra we borrow a technique for process abstraction [8,9], allowing a process term to be simplified such that if the simplified term satisfies the property of interest, then so does the term itself. From Petri nets we borrow S-invariants, T-invariants, and traps [25].

The body of the paper begins with a description of Dekker's algorithm in B(PN)^2 (Basic Petri Net Programming Notation) [4], a simple programming language for Petri nets. Then the box calculus is introduced, and the semantics of B(PN)^2 programs is defined as a translation to expressions of the box calculus. The box calculus, in turn, is given a Petri net semantics. To prove that Dekker's algorithm really does provide mutual exclusion, its box expression is simplified through process abstraction techniques, simplified further with algebraic laws for box expressions, and then S-invariants and traps are used. To prove that requests to enter the critical section are granted, T-invariants are used.

Before concluding, we present Dijkstra’s mutual exclusion algorithm and show that mutual exclusion can be proved for it just as for Dekker’s algorithm.

Throughout the paper, we try to be precise without being unnecessarily formal. In particular, most concepts are defined on the fly and, where possible, in plain English.

*This work was supported in part by ESPRIT WG 6067 CALIBAN.
2 A Basic Petri Net Programming Notation

We consider only a fragment of $B(PN)^2$ large enough to conduct our study. The syntax of the fragment is

\[
\begin{align*}
  \text{program} & := \text{decl}; \text{par} \\
  \text{decl} & := \text{var} \ \text{varname} : \text{set init value} \mid \text{decl};\text{decl} \\
  \text{par} & := \text{com} \mid \text{com} \mid \text{com} \\
  \text{com} & := (\text{expr}) \mid \text{com};\text{com} \mid \text{do} \ \text{altset} \ \text{od} \\
  \text{expr} & := \text{varname} := \text{value} \mid \text{varname} = \text{value} \mid \\
  & \quad (\text{expr} \lor \text{expr}) \mid (\text{expr} \land \text{expr}) \\
  \text{altset} & := (\text{expr});\text{com repeat} \mid (\text{expr});\text{com exit} \mid \text{altset} \mid \text{altset}
\end{align*}
\]

We do not describe the syntax of \text{varname}, \text{set} and \text{value}. It is similar to that used in PASCAL. In this paper, the only sets used are \{true, false\} and \{1, 2\}. A program generated by this syntax is \text{well-formed} if it satisfies the following context conditions:

- every variable that appears in \text{par} is declared in \text{decl};
- no variable is declared twice in \text{decl};
- in a expression \text{var} \ \text{varname} : \text{set init value}, \text{value} is an element of \text{set}.

We will give an informal, operational account of the semantics of $B(PN)^2$ commands (a formal semantics is given later in terms of Petri nets). Variable declarations are familiar from traditional programming languages; for instance, the clause `\text{var b : \{true, false\} init true}' declares a boolean variable \text{b}, with `true' as initial value. The command $(\text{expr})$ denotes the atomic execution of \text{expr}. $B(PN)^2$ makes no syntactic distinction between assignments and guards; $(x := 0)$ assigns 0 to \text{x}, while $(x = 0)$ delays execution until some moment in which \text{x} has value 0. Observe that an atomic action can have a complicated structure, such as $(x = 1 \land (y = 0 \lor z = 2))$. The command $\text{com}_1 \mid \text{com}_2$ denotes the parallel, independent execution of $\text{com}_1$ and $\text{com}_2$, while the command $\text{com}_1;\text{com}_2$ denotes the sequential execution of $\text{com}_1$ and $\text{com}_2$.

Finally, the command

\[
\begin{align*}
  & \text{do} \quad p_1 \ \text{repeat} \ [\ldots] \ p_n \ \text{repeat} \\
  & \quad \qquad q_1 \ \text{exit} \ [\ldots] \ q_m \ \text{exit} \\
  & \quad \text{od}
\end{align*}
\]

denotes a combination of looping and non-deterministic choice. The expressions in front of the alternatives (guards) are evaluated based on the state of the environment. An alternative is non-deterministically chosen from those with ready guards, and execution of that alternative begins. If it terminates, and the alternative is of \text{repeat} type, then the program loops back; if the alternative is of \text{exit} type, then the control is transferred to the point after \text{od}.

We can write a version of Dekker's mutual exclusion algorithm [24] for two processes in $B(PN)^2$. We first encode the processes into commands $p_1$ and $p_2$, which use two shared boolean variables, $b_1$, $b_2$, as well as a shared variable $k$ having \{1, 2\} as its set of values. We give the code for $p_1$; the code for $p_2$ is obtained by exchanging 1 and 2 everywhere.
do
  \( \langle b_1 := \text{true} \rangle \);
  do \( \langle b_2 = \text{true} \land k = 2 \rangle ; \langle b_1 := \text{false} \rangle ; \langle k = 1 \rangle ; \langle b_1 := \text{true} \rangle \) repeat
  \( \langle b_2 = \text{false} \rangle \) exit
od;
\( \langle k := 2 \rangle \); critical section; \( \langle b_1 := \text{false} \rangle \)
repeat
od

The composite atomic action \( \langle b_2 = \text{true} \land k = 2 \rangle \) is important for correctness: if it were replaced by \( \langle b_2 = \text{true} \rangle ; \langle k = 2 \rangle \), then the program could deadlock. The box calculus allows these composite atomic actions to be translated smoothly into multisets of names. It is possible to avoid the composite atomic actions in this program at the price of introducing a busy waiting loop (as done in [27]).

The complete program (with a little bit of syntactic sugar in the declarations) is

\[
\begin{align*}
\text{var } b_1, b_2 &: \{\text{true, false}\} \text{ init false}; \\
\text{var } k : \{1, 2\} \text{ init 1; } \\
p_1 &\parallel p_2
\end{align*}
\]

We want the algorithm to have two important properties. First, \( p_1 \) and \( p_2 \) should never both be in their critical sections. Second, if a process requests access to the critical section by setting its local variable to true, then the request should be eventually granted. We will prove these properties using a combination of process algebra and net techniques. But first we look at the semantics of \( \text{B(PN)}^2 \).

3 The small box calculus

The semantics of \( \text{B(PN)}^2 \) is defined in [4] by a mapping from programs to expressions of the box calculus [1]. The box calculus is a powerful process algebra, and for our case study we only need a small fragment of it. We shall only describe this fragment, which we call the small box calculus.

Let \( \text{Nam} \) be a set of names; for our purposes, we can assume \( \text{Nam} \) to be finite. Following CCS, we also assume that there exists a bijection \( ^\land: \text{Nam} \to \text{Nam} \) with \( \hat{a} \neq a \). The name \( \hat{a} \) is called the conjugate of \( a \). The carrier of the algebra is the set \( \text{Act of actions}, \) defined as the set of finite sets over \( \text{Nam} \); if \( a \) and \( b \) are names, then sets like \{\( a, b \)\} and \{\( a, \hat{a} \)\} are actions\(^1\). In particular, the empty set \( \emptyset \) is an action; its role is similar to the role of \( \tau \) in CCS. Throughout the paper we identify the name \( a \) and the action \{\( a \)\}.

The expressions of the small box calculus have the following syntax, where \( \alpha \) ranges over \( \text{Act} \) and \( \mathcal{N} \) ranges over subsets of \( \text{Nam} \):

\[
\begin{align*}
E &::= F \mid [\mathcal{N}; E] \\
F &::= G \mid F \parallel F \\
G &::= \text{stop} \mid \alpha \mid G; G \mid G \cdot G \mid G * G
\end{align*}
\]

Notice that this is a structured grammar. In the sequel, \( G \) (or \( G_i \)) denotes an expression generated by the last line of the syntax, and similarly for \( F \) and \( E \).

\(^1\)In the full box calculus, recursion suggests that multisets of names be taken as actions, instead of sets.
Informally, the expression stop does nothing, while α just executes the action α. The expression \( G_1; G_2 \) represents sequential composition, and \( G_1 \ ⌢ G_2 \) represents choice. The expression \( G_1 * G_2 \) is behaviourally equivalent to the recursively-defined expression:

\[
X = (G_1; X) ⌢ G_2
\]

Accordingly, \( * \) is called the iteration operator. The expression \( F_1 \parallel F_2 \) represents independent parallel composition, with no synchronisation between \( F_1 \) and \( F_2 \). Finally, the expression \([N : E]\), called scoping of \( N \) in \( E \), enforces synchronisation in \( E \) for the set \( N \) of names. More than two parallel components may synchronise: for instance, the three components of the expression

\[
[\{a, b\} : a \parallel \{\tilde{a}, \tilde{b}\} \parallel \tilde{d}]
\]

communicate in a single \( \emptyset \) action.

**Remark 3.1**

The iteration operator is slightly different from the one defined in [1], which has three arguments instead of two. The expression \( G_1 * G_2 * G_3 \) is equivalent to \( G_1; G_2; G_3 \). In other words, \( G_1 \) guards the execution of \( G_2 \) and \( G_3 \). This guard was introduced in [1] to ensure the consistency between the operational and the net semantics. However, when translating B(PN)² programs into box expressions, only expressions of the form \( \emptyset * G_2 * G_3 \) appear; in this context they can be safely replaced by \( G_2 * G_3 \). So the iteration operator with two arguments allows us to get rid of some \( \emptyset \) actions which, though harmless, complicate the exposition a bit. ■ 3.1

The full box calculus has been given a net semantics [1] and an operational semantics [15] that are formally consistent, in the sense that the transition system obtained from the operational semantics and the one obtained from the net semantics are isomorphic. We do not use the operational semantics in this paper; we refer the interested reader to [15]. The net semantics of the small box calculus is given in the next section, but first we describe how to translate Dekker’s algorithm into an expression of the small box calculus.

The B(PN)² program for Dekker’s algorithm translates to the box expression

\[
\text{Dekker} = [N : P_1 \parallel P_2 \parallel B_1 \parallel B_2 \parallel K]
\]

where commands \( p_1 \) and \( p_2 \) translate to \( P_1 \) and \( P_2 \), and the variables \( b_1, b_2, \) and \( k \) translate to \( B_1, B_2, \) and \( K \). The set \( N \) contains the names on which the parallel components synchronise.

To explain how variables and commands are translated to box expressions we define a mapping \( E \) from some B(PN)² code \( p \) to a box expression \( E(p) \). In translating variables, we assume that every expression of the form \( \text{varname} := \text{value} \) or \( \text{varname} = \text{value} \) has been assigned a name. An expression \( v := c \), where \( v \) is a variable name and \( c \) a value, is assigned the name \( \text{vwc} \) (\( \text{w} \) stands for ‘write’); an expression \( v = c \) is assigned the name \( \text{vrc} \) (\( \text{r} \) stands for ‘read’). Then, \( E(\text{var} b_1; \{\text{true, false}\} \text{init false}) \) is the following expression \( B_1 \) (we assume that ‘:’ binds stronger than ‘\( \emptyset \)’, and ‘\( \| \)’ binds stronger than ‘\( * \)’)

\[
B_1 \equiv (b_1rf \ ⌢ b_1wf * b_1wt) ; (b_1rt \ ⌢ b_1wt * b_1wf) * \text{stop}
\]

Similarly, the declarations of \( b_2 \) and \( k \) translate to the expressions \( B_2, K \):

\[
B_2 \equiv (b_2rf \ ⌢ b_2wf * b_2wt) ; (b_2rt \ ⌢ b_2wt + b_2wf) * \text{stop}
\]

\[
K \equiv (kr1 \ ⌢ kw1 * kw2) ; (kr2 \ ⌢ kw2 * kw1) * \text{stop}
\]

Next we consider the translation of commands. The command \( (\text{expr}) \), where expr is the composition of subexpressions by conjunction and disjunction, translates to a choice \( \alpha_1 \ ⌢ \ldots \ ⌢ \alpha_n \) between actions. To compute the actions:
• put \texttt{expr} in disjunctive normal form; let \texttt{expr}_1, \ldots, \texttt{expr}_n be the disjuncts of the normal form;

• for every \(i\), take \(\alpha_i\) as the set of the names given to the conjuncts of \texttt{expr}_i.

For example, we have

\[
\mathcal{E}( ((k = 1 \lor (k = 2 \land b_2 = \text{false})) ) = \text{kr1} \parallel \{\text{kr2}, b_{2\text{rf}}\}
\]

The commands for sequential and parallel composition are easy to translate:

\[
\begin{align*}
\mathcal{E}(\text{com}_1; \text{com}_2) &= \mathcal{E}(\text{com}_1) \cdot \mathcal{E}(\text{com}_2) \\
\mathcal{E}(\text{com}_1 \parallel \text{com}_2) &= \mathcal{E}(\text{com}_1) \parallel \mathcal{E}(\text{com}_2)
\end{align*}
\]

Finally, the command

\[
\text{do } p_1 \text{ repeat } \square \cdots \square p_n \text{ repeat} \\
\square q_1 \text{ exit } \square \cdots \square q_m \text{ exit} \\
\text{od}
\]

is translated to

\[
\mathcal{E}(p_1) \square \cdots \square \mathcal{E}(p_n) \star \mathcal{E}(q_1) \square \cdots \square \mathcal{E}(q_n)
\]

If there are no \texttt{repeat} (\texttt{exit}) clauses, then we write \texttt{stop} in the first (second) argument of the iteration.

The expression \(P_1\) for the command \(p_1\) of Dekker's algorithm is then given by the following abbreviations:

\[
\begin{align*}
P_1 &\equiv \{b_{1\text{wt}}, \text{request}_1\}; P_{11} \star \text{stop} \\
P_{11} &\equiv (P_{12} \star b_{2\text{rf}}); \text{kr2}; \text{enter}_1; \{\text{exit}_1, b_{1\text{wf}}\} \\
P_{12} &\equiv \{b_{2\text{rt}}, b_{2\text{rf}}\}; b_{1\text{wf}}, \text{kr1}; b_{1\text{wt}}
\end{align*}
\]

The names \texttt{request}_1, \texttt{enter}_1, and \texttt{exit}_1 have been inserted to show when the process requests entry, enters, and exits its critical section. The process \(P_1\) requests access to the critical section when it executes \(b_{1\text{wt}}\) in \(P_1\). This can be faithfully modelled using the fact that an action can be a set of names: in \(\{b_{1\text{wt}}, \text{request}_1\}\), both names occur at the same time. If we had used \(b_{1\text{wt}}; \text{request}_1\) instead, the names would occur in sequence.\footnote{A similar construction is possible in MEIJE [6].} This point is further discussed in the conclusions.

4 Net Semantics of the small box calculus

Here we describe the net semantics for the small box calculus. We first need some basic notions about labelled Petri nets and boxes.

Labelled Petri nets, boxes

A labelled net \(N\) is a four-tuple \((S, T, F, l)\), where

• \(S\) and \(T\) are disjoint, finite sets,
F is a relation on S ∪ T such that F ∩ (S × S) = F ∩ (T × T) = ∅, and

l is a mapping T → Act, where Act is a set of actions.

The elements of S and T are called places and transitions, respectively. Places and transitions are generically called nodes. Given a node x of N, \(x^* = \{y \mid (y, x) ∈ F\}\) is the preset of x and \(x^+ = \{y \mid (x, y) ∈ F\}\) is the postset of x. Given a set of nodes X of N, we define \(X^* = \bigcup_{x ∈ X} x^*\)

A marking of N is a mapping \(M : S → N\). A marking \(M\) enables a transition \(t\) if \(M(s) > 1\) for every place \(s ∈ t^*\). If a transition \(t\) is enabled at \(M\), then it can occur, and its occurrence leads to the successor marking \(M'\), written \(M \xrightarrow{t} M'\), which is defined for every place \(s\) by

\[M'(s) = \begin{cases} M(s) & \text{if } s ∉ t^* \text{ and } s ∉ t^+ \\ M(s) - 1 & \text{if } s ∈ t^* \text{ and } s ∉ t^+ \\ M(s) + 1 & \text{if } s ∉ t^* \text{ and } s ∈ t^* \end{cases}\]

A labelled Petri net is a pair \((N, M_0)\) where N is a labelled net and \(M_0\) is a marking of N. A marking \(M\) is reachable from \(M_0\) if \(M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M\) for some sequence of steps \(t_1, \ldots, t_n\). \((N, M_0)\) is said to be 1-safe if for every reachable marking \(M\) and every place \(s\), \(M(s) ≤ 1\).

A box is a labelled net with two sets of distinguished places, the entry and exit places\(^3\). Formally, in a box we add to the labelling map \(l : T → Act\) a partial mapping \(l' : S → \{e, x\}\); the entry and exit places are labelled by \(e\) and \(x\), respectively. The initial marking of a box is the one which puts one token in each entry place and no tokens in the rest. In the sequel, we identify the box and the Petri net composed by the box and its initial marking.

### Box semantics

The box semantics of the small calculus is given by 0 a mapping \(B\) which assigns a box \(B(E)\) to the box expression \(E\). The mapping B is defined inductively on the structure of box expressions: we first define \(B(G)\), then \(B(F)\), and finally \(B(E)\).

We make use of a merging operation. Merging two or more places of a net consists in adding a new place, whose set of input (output) transitions is the union of the sets of input (output) transitions of the old places, and then removing the old places.

The definition of \(B(G)\) is illustrated in Figure 1. The box \(B(\text{stop})\) has just an entry place \(e\) and an exit place \(x\). The box \(B(\alpha)\) has moreover a transition labelled \(\alpha\). The box \(B(G_1; G_2)\) is constructed by merging the exit place of \(B(G_1)\) and the entry place of \(B(G_2)\); the resulting place carries no label. For the box \(B(G_1 □ G_2)\), merge the entry places of \(B(G_1)\) and \(B(G_2)\), labelling the result with \(e\), and then merge the exit places, labelling the result with \(x\). The box \(B(G_1 \star G_2)\) is constructed by merging the entry and exit places of \(B(G_1)\) and the entry place of \(B(G_2)\); the resulting place is labelled by \(e\).

The following proposition follows immediately from the definitions:

**Proposition 4.1**

For every expression \(G\), the box \(B(G)\) has exactly one entry and one exit place. Moreover, every transition of \(B(G)\) has exactly one input and one output place.
Figure 1: Box semantics

The box \( B(G_1 \parallel G_2) \) is constructed by simply putting \( B(G_1) \) and \( B(G_2) \) side by side. Therefore, the box for a expression \( B(F) \) may have more than one entry and more than one exit place, but it is still the case that every transition has one input and one output place. It also follows from this definition that the operator \( \parallel \) is associative and commutative.

Finally, we describe the box \( B([N: E]) \). If \( N = \{a_1, \ldots, a_m\} \), then we define

\[
[A: E_1 \parallel \cdots \parallel E_n] = [a_1: [\cdots [a_m: E_1 \parallel \cdots \parallel E_n] \cdots]
\]

The operator \([a: E]\) has been proved to be associative and commutative in [1], and therefore \([N: E]\) is independent of the order in which the scopings are nested. Now, \( B([a: E]) \) is defined as the result of performing the following operation on \( B(E) \):

1. For every pair of transitions \( t_1, t_2 \) such that some name \( a \) appears in the label of \( t_1 \) and \( \tilde{a} \) appears in the label of \( t_2 \), add a new transition whose set of input (output) places is the union of the sets of input (output) places of the old transitions (but without removing the old transitions); label the new transition with the set of all names present in the labels of \( t_1, t_2 \), except \( a \) and \( \tilde{a} \).

2. Remove from the resulting net all transitions having \( a \) or \( \tilde{a} \) in its label.

Figure 2 shows these steps for \([\{a, b\}: a \parallel \{\tilde{a}, b\} \parallel \tilde{b}\] .

Observe that the box \( B([N: G_1 \parallel \cdots \parallel G_n]) \) has the same places as \( B(G_1 \parallel \cdots \parallel G_n) \).

Figure 3 shows several steps in the construction of the box corresponding to the expression

\[
B_1 \equiv (b_1 \overline{r} f \parallel b_1 w t \ast b_1 w t) \; ; \; (b_1 \overline{r} t \parallel b_1 w t \ast b_1 w f) \ast \text{stop}
\]
A consequence of this definition is that the boxes corresponding to expressions of the small box calculus are 1-safe.

Proposition 4.2

If $E$ is an expression of the small box calculus, then $B(E)$ is a 1-safe net.

Proof: Since $B(G)$ has one entry place, the total number of tokens of the initial marking is 1. Since every transition of $B(G)$ has exactly one input place and one output place, the total number of tokens at any reachable marking is 1 as well. So $B(G)$ is 1-safe.

It follows immediately from the definition of 1-safeness that if the boxes $B(G_1), \ldots, B(G_n)$ are 1-safe, then $B(G_1) \parallel \ldots \parallel B(G_n)$ is 1-safe. So it remains to prove that $B([N: G_1 \parallel \ldots \parallel G_n])$ is 1-safe. For that, we show that every reachable marking of $B([N: G_1 \parallel \ldots \parallel G_n])$ is also a reachable marking of $B(G_1) \parallel \ldots \parallel B(G_n)$. This is proved by the following two observations:

- the new transitions do not add any new reachable markings: the marking reached after the occurrence of a new transition obtained from transitions $t_1, t_2$ can be reached by the sequential occurrence of $t_1, t_2$;
- removing transitions whose label contains some name of $N$ can only decrease the set of reachable markings.

Devillers has proved more general results about 1-safeness in the box calculus in [12].
Figure 3: Box corresponding to a boolean variable
5. Proving Mutual Exclusion

The most important property to show of Dekker’s algorithm is that two processes are never both in their critical section. Before expressing it formally, we recall some definitions about traces.

The expression $M \xrightarrow{\alpha} M'$, where $M$ and $M'$ are markings of a net, denotes that $M$ enables some transition labelled by an action $\alpha$, and that the marking reached by its occurrence is $M'$.  

A trace of a box is a finite sequence $\alpha_1 \alpha_2 \ldots \alpha_n$ of actions such that

$$M_0 \xrightarrow{\alpha_1} M_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} M_n$$

for some markings $M_1, \ldots, M_n$, where $M_0$ is the initial marking of the box. Two box expressions $E_1$ and $E_2$ are trace equivalent if $B(E_1)$ and $B(E_2)$ have the same traces.

A weak trace is obtained from a trace by removing all $\emptyset$ actions. Box expressions $E_1$ and $E_2$ are weak trace equivalent, written $E_1 = E_2$, if $B(E_1)$ and $B(E_2)$ have the same weak traces.

We say that an equivalence $\approx$ is a congruence for the small box calculus if $G_1 \approx G'_1$ implies

- $G_1 \circ G_2 \approx G'_1 \circ G_2$ for $\circ \in \{; \},$ and
- $[N:G_1 || \ldots || G_n] \approx [N':G'_1 || \ldots || G_n].$

Using results of [18,5], it is easy to prove the following result:

Theorem 5.1

Weak trace equivalence is a congruence for the small box calculus.

Proof: The only nontrivial part is to prove that $G_1 = G'_1$ implies

$$[N:G_1 || \ldots || G_n] = [N':G'_1 || \ldots || G_n].$$

It is shown in [5] (Sections 4.2.5 and 4.2.6, see also [18]) that the set of nonsequential processes [3] of the box $B([N:G_1 || \ldots || G_n])$ can be computed from the set of nonsequential processes of the boxes $B(G_1), \ldots, B(G_n)$ (it has to be taken into account that our scoping operator $[a : E]$ is the composition of two operators of [5]: $[a : E] = (E \, s y \, a) \, r s \, a$).

Consider the boxes $B(G_1)$, since it has exactly one entry place, the total number of tokens of its initial marking is 1; since all its transitions have exactly one input and one output place, the total number of tokens at any reachable marking is 1 as well. Therefore, no reachable marking concurrently enables two transitions. In this case, the set of processes of $B(G_1)$ can be retrieved from its set of traces. So, if $G_1$ and $G'_1$ are trace equivalent, then $[N:G_1 || \ldots || G_n]$ and $[N':G'_1 || \ldots || G_n]$ have the same nonsequential processes. In particular, this implies that they are trace equivalent.

This proves that trace equivalence is a congruence. To prove the same result for weak trace equivalence, we observe that, from the definition of the box semantics of the scoping operator, the transitions labelled by $\emptyset$ do not generate any new transition when scoping is applied. Therefore, if $G_1 = G'_1$, then $[N:G_1 || \ldots || G_n]$ and $[N':G'_1 || \ldots || G_n]$ have the same weak pomsets of transitions (pomsets in which the transitions labelled by $\emptyset$ are removed), and are thus weakly trace equivalent.

We can now formally express the property that two processes are never both in their critical section.
In every weak trace of Dekker, between every two different occurrences of an enter name there is an occurrence of an exit name.

We refer to this as the property of mutual exclusion. We will simplify Dekker using a technique [8,9] from process algebra such that, if the simplified expression satisfies mutual exclusion, then so does Dekker. Then we will apply Petri net techniques to the simplified expression.

The idea behind the simplification is that only some of the variables in the algorithm play a role in establishing mutual exclusion. To remove the others, we define an operation that removes all occurrences of a synchronisation name in an expression, thereby weakening the synchronisation between concurrent components.

Definition 5.2

The hiding of a name \( a \) in a box expression \( E \), written \( E \backslash a \), is defined as follows:

\[
\begin{align*}
\text{stop} \backslash a &\equiv \text{stop} \\
\alpha \backslash a &\equiv \alpha - \{a, \overline{a}\} \\
(G_1 ; G_2) \backslash a &\equiv G_1 \backslash a ; G_2 \backslash a \\
(G_1 \parallel G_2) \backslash a &\equiv G_1 \backslash a \parallel G_2 \backslash a \\
(G_1 \ast G_2) \backslash a &\equiv G_1 \backslash a \ast G_2 \backslash a \\
(F_1 || F_2) \backslash a &\equiv F_1 \backslash a || F_2 \backslash a \\
[N : E] \backslash a &\equiv [N : E] \backslash a \\
\end{align*}
\]

When hiding of a name \( a \) is applied to an expression of the form \([N : E]\), where \( a \in N \), it "enlarges" its set of weak traces, as we shall prove in the next theorem. However, this enlargement is only with respect to traces that have no actions containing more than one name. The linearisation of a set \( \Sigma \) of traces is the set of all traces that can be derived from traces in \( \Sigma \) by replacing actions of form \( \{a_1, \ldots, a_n\} \) by the sequence \( \{a_1\} \ldots \{a_n\} \) of actions. Since enter and exit occur only as singly-named actions, the traces of Dekker will satisfy mutual exclusion iff the linearisation of the traces do.

Theorem 5.3

If \( a \in N \), then the set of linearised weak traces of \([N : G_1 \parallel \ldots \parallel G_n]\) is contained in the set of linearised weak traces of \([N : G_1 \parallel \ldots \parallel G_n] \backslash a\).

Proof: We can write \([N : G_1 \parallel \ldots \parallel G_n]\) equivalently as \([a : \{N - \{a\} : G_1 \parallel \ldots \parallel G_n\}]\) since scoping is associative and commutative. Suppose \(B([a : \{N - \{a\} : G_1 \parallel \ldots \parallel G_n\}]\) can perform action \(\alpha\). If the action is not due to synchronisation on \(a\), then \(B([N - \{a\} : G_1 \parallel \ldots \parallel G_n])\) can also perform \(\alpha\) to reach the same marking. Since \(a \not\in \alpha\), so can \(B([N - \{a\} : G_1 \backslash a \parallel \ldots \parallel G_n \backslash a])\) which is the same box as \(B([N : G_1 \parallel \ldots \parallel G_n] \backslash a)\).

On the other hand, if \(\alpha = \beta \cup \gamma\), due to synchronisation of an action \(\{a\} \cup \beta\) of some \(G_i\) and an action \(\{\overline{a}\} \cup \beta\) of some \(G_j\), then \(B([N - \{a\} : G_1 \backslash a \parallel \ldots \parallel G_n \backslash a])\) can perform both \(\gamma\beta\) and \(\beta\gamma\) to reach the same marking. Furthermore, the weak linearised traces of \(\{\gamma \cup \beta\}\) are the same as those of \(\{\gamma \beta, \beta \gamma\}\). Since in either case the same marking is reached, \(B([a : \{N - \{a\} : G_1 \parallel \ldots \parallel G_n\}]\) can continue to match actions of \(B([N : G_1 \parallel \ldots \parallel G_n] \backslash a)\) in this way. 

\[\blacksquare\] 5.3
Our intuition about the algorithm tells us that only variables $b_1$ and $b_2$ are important for establishing mutual exclusion. Therefore, we will hide the names of the variable $k$. Hiding is commutative and associative, so we will allow hiding of sets of names. Furthermore, we will write $b_1$ for the set $\{b_{1rt}, b_{1rf}, b_{1wt}, b_{1wt}\}$, and similarly for $b_2$ and $k$.

Applying hiding, we have

$$\text{Dekker}'k = [N : P'_1 || P'_2 || B_1 || B_2 || N']$$

where $P'_1$ is an abbreviation for $P_1 \setminus k$, and similarly for $P'_2$ and $N'$. Observe that $B_1$ and $B_2$ are not affected by the hiding. Repeated application of the definition of hiding gives:

$$P'_1 \equiv \{b_{1wt}, \text{request}_1\}; P'_{11} * \text{stop}$$
$$P'_{11} \equiv (P'_{12} * b_{2rf}); \emptyset; \text{enter}_1; \{\text{exit}_1, b_{1wf}\}$$
$$P'_{12} \equiv b_{2rt}; b_{1wf}; \emptyset; b_{1wt}$$

and similarly for $P'_2$, while $N'$ is obtained replacing all the names that appear in $N$ by $\emptyset$.

By Theorem 5.3, if $\text{Dekker}'k$ satisfies mutual exclusion, so does $\text{Dekker}$. We can further simplify $\text{Dekker}'k$ by the following algebraic rules for the box calculus, which follow easily from the definitions.

**Proposition 5.4**

The following equations hold:

$$G; \emptyset = G$$
$$\emptyset; G = G$$
$$G \parallel G = G$$
$$G \parallel \text{stop} = G$$
$$\emptyset * G = G$$
$$\text{stop} * G = G$$

Since weak trace equivalence is a congruence, we can substitute an expression $G$ by an equivalent one $G'$ anywhere within $\text{Dekker}'$. By repeatedly simplifying $P'_1$, we get the expression

$$P''_1 \equiv \{b_{1wt}, \text{request}_1\}; (b_{2rt}; b_{1wf}; b_{1wt} * b_{2rf}); \text{enter}_1; \{\text{exit}_1, b_{1wf}\} * \text{stop}$$

The expression for $N'$ simplifies to stop, and then, by the fourth equation, disappears. In the end, we get the following expression $\text{Dekker}''$

$$\text{Dekker}'' = [N : P''_1 || P''_2 || B_1 || B_2]$$

The box $B(P''_1 || P''_2 || B_1 || B_2)$ is shown in Figure 4 (the exit places, which are not connected to any transition, have been omitted). The identities of places and transitions are shown inside the circles and boxes, while their labels are written close to them. The box $B(\text{Dekker}''')$ has the same places, and fewer transitions, but a more complicated interconnection pattern (which makes it difficult to draw).

We now apply net techniques to $\text{Dekker}'''$ using the mapping $B$. In particular, we compute some $S$-invariants and some traps of the box $B(\text{Dekker}'''')$, and prove that $\text{Dekker}'''$ satisfies mutual exclusion. We first recall the basic definitions and notions about $S$-invariants and traps.
Let $N = (S, T, F)$ be a net. A mapping $I: S \rightarrow Q$ is an $S$-invariant if, for every transition $t \in T$

$$\sum_{s \in t^*} I(s) = \sum_{s \in t^*} I(s).$$

Given a marking $M$ of $N$, the product $I \cdot M$ is defined as $\sum_{s \in S} I(s)M(s)$. The fundamental property of an $S$-invariant $I$, which follows easily from its definition and the occurrence rule, is that in a Petri net $(N, M_0)$, if $M$ is a marking reachable from $M_0$ then we have

$$I \cdot M = I \cdot M_0$$

This property can be used to prove that a given marking is not reachable, by finding a suitable $S$-invariant and showing that the equality does not hold. $S$-invariants can be efficiently computed using linear algebraic techniques (see, for instance, [10]).

A trap of a net $N = (S, T, F)$ is a set of places $R \subseteq S$ such that $R^* \subseteq \cdot R$, i.e., every output transition of some place of $R$ is also an input transition of some other place of $R$. A trap $R$ is marked at a marking $M$ if $M(s) > 0$ for some place $s$ of $R$. The fundamental property of a trap $R$ is that if $R$ is marked at a marking $M$ and $M'$ is reachable from $M$, then $R$ is also marked at $M'$; loosely speaking, marked traps remain marked. The reason is that, from the definition of a trap, every transition which removes tokens from some place of $R$ also adds tokens to some place of $R$. 

Figure 4: The box $B(P_1' \parallel P_2' \parallel B_1 \parallel B_2)$
The traps of a net can also be efficiently computed using linear algebraic techniques [16, 14], or resolution algorithms for Horn clauses [20]. Several net tools (e.g., [26]) can compute both S-algorithms and traps.

It is easy to deduce from $B(P_1^1 || P_2^2 || B_1 || B_2)$ some of the S-invariants of $B(Dekker''')$ using the following result:

**Proposition 5.5**

*Every S-invariant of $B(E)$ is also an S-invariant of $B([N: E])$.*

**Proof:** First, observe that $B(E)$ and $B([N: E])$ have the same sets of places. Let $I$ be an S-invariant of $B(E)$. We have to show that for every transition $t$ of $B([N: E])$ we have

$$
\sum_{s \in t^*} I(s) = \sum_{s \in t^*} I(s)
$$

By the definition of scoping, the set of input (output) places of a transition $t$ of $B([N: E])$ is the union of the sets of input (output) places of a certain set $T$ of transitions of the box $B(E)$. We then have

$$
\sum_{s \in t^*} I(s) = \sum_{t' \in T} \sum_{s \in t'} I(s) = \sum_{t' \in T} \sum_{s \in t'} I(s) = \sum_{s \in t^*} I(s)
$$

which proves that $I$ is an S-invariant of $B([N: E])$.$\blacksquare$

These are the S-invariants:

$$
\sum_{i=1}^{6} M(s_i) = 1 \quad \sum_{i=1}^{6} M(r_i) = 1 \tag{1}
$$

$$
M(b_1 f) + M(b_1 t) = 1 \quad M(b_2 f) + M(b_2 t) = 1 \tag{2}
$$

We can also compute the following three traps:

$$
\{s_1, s_4, b_1 t\} \quad \{r_1, r_4, b_2 t\} \tag{3}
$$

$$
\{s_2, s_3, r_2, r_3, b_1 f, b_2 f\} \tag{4}
$$

Since the initial marking of $B(Dekker''')$ puts tokens on the places $s_1$, $r_1$, $b_1 f$ and $b_2 f$, all these traps are marked at the initial marking, and therefore remain marked at any reachable marking.

We prove that *mutual exclusion* holds for Dekker'' by contradiction. Assume *mutual exclusion* does not hold. Then, for some reachable marking $M$ we have $M(s_6) = 1$ and $M(r_6) = 1$ in $B(Dekker''')$. For this marking $M$, we have, by (1):

$$
M(s_2) = M(s_3) = M(s_4) = 0 \quad M(r_2) = M(r_3) = M(r_4) = 0 \tag{5}
$$

Then by (5) and (3)

$$
M(b_1 t) > 0 \quad M(b_2 t) > 0 \tag{6}
$$

which gives, by (2), that

$$
M(b_1 f) = 0 \quad M(b_2 f) = 0 \tag{7}
$$

But then, by (5) and (7) we get

$$
M(s_2) + M(s_3) + M(r_2) + M(r_3) + M(b_1 f) + M(b_2 f) = 0
$$

which contradicts that (4) is an initially marked trap.
6 Proving Liveness

The liveness property we wish to prove of Dekker's algorithm is that if a process requests access to the critical section, then it eventually enters the critical section\(^4\).

In order to formalise this property it is necessary to consider the infinite behaviours of systems. So we extend the notion of occurrence sequence to include also, for a box with initial marking \(M_0\), infinite sequences of the form

\[
M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \cdots
\]

We also extend the notion of trace to include those produced by infinite occurrence sequences.

We can now formalise the liveness property as

For \(i = 1, 2\), if \(\text{request}_i\) occurs at some point in an infinite trace, then \(\text{enter}_i\) occurs at some later point.

We refer to this as the property of \textit{liveness}. Notice that \textit{liveness} considers only infinite traces, and therefore may hold for algorithms that can deadlock before a request is granted. This possibility does not exist for Dekker's algorithm, which can be shown to be deadlock-free using S-invariants\(^5\).

In a trace of Dekker, the possible actions are \(\emptyset\), or one of \(\text{request}_1\), \(\text{entry}_1\), \(\text{exit}_1\) for \(i = 1, 2\). As in CCS, we cannot tell, from the occurrence of a synchronisation action, the name or names on which the synchronisation took place. For example, an \(\emptyset\) action could be the result of the synchronisation of \(a\) and \(\bar{a}\), \(b\) and \(\bar{b}\), or some other possibility. However, for reasoning about the liveness property we would like to use this information.

In the box calculus this problem can be solved without changing the semantics of the scoping operator, by changing the naming convention in the translation of B(PN)\(^2\) to the box calculus. While the translation of variables remains unchanged, we now assign to expressions \(v := c\) and \(v = c\) the sets of names \(\{\text{vwc, vwc}\}\) and \(\{\text{vrc, vrc}\}\). So, for instance, we now have

\[
\mathcal{E}( (\langle k = 1 \lor (k = 2 \land b_2 = \text{false}) \rangle ) = \{\text{kr1, kr1}\} \parallel \{\text{kr2, kr2, b2rf, b2rf}\}
\]

The new names \text{vwc}, \text{vrc} never synchronise, because their conjugates do not appear in the box expression. There are no longer any \(\emptyset\) actions, and each action indicates the synchronising names.

We will try to prove that Dekker satisfies the part of \textit{liveness} corresponding to \(i = 2\); the other part is similar\(^6\). First, we introduce T-invariants, the additional analysis technique we will use. All the results we present are well-known; however, we give short proofs of some of them for completeness.

Let \(N = (S, T, F)\) be a net. A mapping \(J: T \rightarrow Q\) is a \textit{T-invariant} if, for every place \(s \in S\)

\[
\sum_{i \in s} J(t) = \sum_{i \in s} J(t).
\]

A T-invariant \(J\) is \textit{semi-positive} if \(J(t) \geq 0\) for every transition \(t\), and \(J(t) \neq 0\) for some transition \(t\). The \textit{support} of a semi-positive T-invariant \(J\) is the set of transitions \(t\) such that \(J(t) > 0\).

The property of T-invariants we are interested in is the following:

\(^4\)Liveness' refers here to the usual division of properties into safety and liveness properties, and not to the notion of liveness in Petri nets (absence of global or partial deadlocks).

\(^5\)The proof is not included for conciseness; it uses the same techniques as the proofs of \textit{mutual exclusion} and \textit{liveness}.

\(^6\)Notice that Dekker is not completely symmetrical, because the variable \(k\) is initially set to 1.
Proposition 6.1

Let \((N, M_0)\) be a 1-safe Petri net, and let \(C = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots\) be an infinite occurrence sequence. The set of transitions of \(N\) that appear infinitely often in the sequence is the support of a semi-positive \(T\)-invariant.

Proof: Let \(C' = M_{k-1} \xrightarrow{t_k} M_{k+1} \xrightarrow{t_{k+1}} \ldots\) be a suffix of \(C\) such that every transition of \(N\) appears either never or infinitely often in \(C'\). Since \((N, M_0)\) is 1-safe, it has finitely many reachable markings. So there exists a marking \(M\) such that \(M = M_{k+j}\) for infinitely many values of \(j\).

Choose two of those values, \(j_1, j_2\), such that for every transition \(t\) that appears in \(C'\), \(t = t_k\) for some \(j_1 \leq k \leq j_2\). Define the mapping \(J: T \to \mathbb{Q}\) in the following way (\(T\) is the set of transitions of \(N\)):

\[
J(t) = \text{number of times that } t \text{ appears between } M_{j_1} \text{ and } M_{j_2}
\]

By the occurrence rule, we have for every place \(s\):

\[
M_{j_2}(s) = M_{j_1}(s) + \sum_{t \in s} J(t) - \sum_{t \not\in s} J(t)
\]

Since \(M_{j_1}(s) = M(s) = M_{j_2}(s)\), \(J\) is a \(T\)-invariant of \(N\), and its support is the set of transitions that appear infinitely often in \(C\). \(\blacksquare\) 6.1

A minimal \(T\)-invariant is a semi-positive \(T\)-invariant whose support is minimal with respect to set inclusion. We have the following property:

Proposition 6.2

Every semi-positive \(T\)-invariant is the sum (defined componentwise) of minimal \(T\)-invariants.

Proof: It follows immediately from the definition that semi-positive \(T\)-invariants are closed under sum (defined componentwise) and multiplication by a positive scalar. Let \(J\) be a semi-positive invariant. If \(J\) is minimal, we are done. If \(J\) is not minimal, then there exists a minimal \(T\)-invariant \(J'\) with smaller support. We can find a constant \(k\) such that \(J'' = J - k \cdot J'\) is a semi-positive \(T\)-invariant, whose support is smaller than that of \(J\). So \(J = k \cdot J' + J''\). If \(J''\) is minimal, we are done. Otherwise, we iterate the procedure. \(\blacksquare\) 6.2

Minimal \(T\)-invariants can be calculated using techniques of linear algebra and Linear Programming (see [10]).

This finishes our introduction to \(T\)-invariants.

To prove liveness, we assume that some infinite occurrence sequence \(C = M_0 \xrightarrow{t_1} M_1 \rightarrow M_2 \xrightarrow{t_2} \ldots\) of \(B(Dekker)\) contains a transition labelled by \(\text{request}_2\), but no later transition labelled by \(\text{enter}_2\) action. Let \(A(C)\) be the set of actions which label the transitions occurring infinitely often in \(C\). Note that \(\text{enter}_2\) does not belong to \(A(C)\).

Now we will attempt to show that this assumption leads to a contradiction. We compute the set of minimal \(T\)-invariants of \(B(Dekker)\) whose support does not contain the transition labelled \(\text{enter}_2\). The sets of labels of their supports are:

\[
A_1 = \{ \text{request}_1, B_1\overline{WT} \}, B_2 RF, KW_2, \text{enter}_1, \{ \text{exit}_1, B_1\overline{WF} \}
\]

\[
A_2 = \{ B_2 RT, KR_2 \}, B_1\overline{WF}, KR_1, B_1\overline{WT} \}
\]

\[
A_3 = \{ B_1 RT, KR_1, B_2\overline{WF}, KR_2, B_2\overline{WT} \}
\]

16
To help visualise these minimal T-invariants, the box $B(P_1 \parallel P_2 \parallel K)$ is shown in Figure 5 (to obtain $B(P_1 \parallel P_2 \parallel B_1 \parallel B_2 \parallel K)$, add the boxes for $B_1$ and $B_2$ shown in Figure 4). The additional uppercase names introduced to make $\emptyset$ actions visible are not shown for clarity. For instance, a transition labelled by $b_2rf$ in the figure is in fact labelled by $\{b_2rf, b_2RF\}$. The sets $A_1$ and $A_2$ correspond to the big and small circuits, respectively, of $B(P_1)$, while $A_3$ corresponds to the small circuit of $B(P_2)$.

By Proposition 6.1, the set of transitions of $B(\text{Dekker})$ that occur in $C$ is the support of a semi-positive T-invariant, and therefore the union of the supports of some minimal T-invariants. So $A(C)$ is the union of one or more of the sets $A_1$, $A_2$ and $A_3$.

We immediately observe that $A_2$ cannot be included in $A(C)$, because every computation which executes transitions labelled by $\text{KR2}$ and $\text{KR1}$ infinitely often must also execute some
transition labelled by $\bar{KW}_1$ infinitely often, and $\bar{KW}_1$ is not present in any of $A_1$, $A_2$, $A_3$. We can exclude $A_2$ for the same reason. The only remaining possibility is $A(C) = A_1$, and we cannot find any argument to exclude it.

So, in the end, we are not able to reach a contradiction. This makes us suspect that, after all, Dekker does not satisfy liveness. We try to find an infinite trace of Dekker containing request$_2$ but not enter$_2$, and we succeed. It is the following:

$\{B_1WT, request_1\} \{B_2WT, request_2\} B_1RT KR_1 B_2WF$  
$\left( B_2RF \bar{KW}_2 \text{ enter}_1 \{\text{exit}_1, B_1WF\} \{B_1WT, request_1\}\right)^\omega$

(this trace is easy to find once we know that $A(C)$ must necessarily be $A_1$). Roughly speaking, what happens in the trace is that $P_2$ refuses to perform $KR_2$, although it can do it at any time after the occurrence of $B_2WF$. This trace suggests that Dekker may satisfy liveness if we restrict the formulation of the property to the infinite traces that satisfy an adequate fairness condition. The one we choose is justice, introduced in [17], but relativised with respect to a set of actions.

Let $A$ be a set of actions. An infinite occurrence sequence $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ is $A$-just if every transition which is enabled almost everywhere (that is, enabled at all but finitely many of the markings $M_0, M_1, \ldots$) and carries a label from $A$ occurs infinitely often in the sequence. A trace is $A$-just if some $A$-just occurrence sequence generates it.

The relativisation is introduced to take care of the fact that a process may never wish to access the critical section. Traces in which transitions labelled with $\{B_1WT, request_1\}$ are enabled almost everywhere but never occur should not be excluded. So we define $A$ as the set of all actions that occur in Dekker except the actions $\{B_1WT, request_1\}$ and $\{B_2WT, request_2\}$, and redefine liveness as:

For $i = 1, 2$, if request$_i$ occurs at some point in an $A$-just infinite trace, then enter$_i$ occurs at some later point.

Assume now that there exists an $A$-just infinite trace $\sigma$ of Dekker which contains request$_2$ but not enter$_2$. We proceed as before by observing that neither $A_2$ nor $A_3$ can be included in the set $A(C)$, but now we also prove that $A(C)$ cannot be the set $A_1$. This produces the desired contradiction.

To prove that $A(C)$ cannot be $A_1$, we first look at the box $B(P_1 \parallel P_2 \parallel B_1 \parallel B_2 \parallel K)$. Using Proposition 5.5, it is easy to deduce from this box some of the S-invariants of $B(Dekker)$.

\[
\begin{align*}
\sum_{i=1}^{8} M(s_i) &= 1 \\
\sum_{i=1}^{8} M(r_i) &= 1 \quad (1) \\
M(b_1f) + M(b_1t) &= 1 \\
M(b_2f) + M(b_2t) &= 1 \quad (2) \\
M(kf) + M(k\ell) &= 1 \quad (3)
\end{align*}
\]

We can also compute the following initially marked trap:

$\{r_1, r_6, r_8, b_2t\}$  

(4)

Now, let $C = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ and assume $A(C) = A_1$. Then there exists an index $j$ such that for every $l \geq j$

(a) $M_l(k\ell) = 1$ (because $KW_2$ occurs infinitely often in $C$, but $KW_1$ does not),

(b) $M_l(b_2f) = 1$ (because $B_2RF$ occurs infinitely often in $C$, but $B_2WT$ does not), and
(c) $M_i(r_m) = 1$ for some place $r_m$ of $B(P_2)$ (because the transitions labelled by the names of $P_2$ occur only finitely often in $C$; so $j$ can be taken greater than the index of the last occurrence of any of these transitions).

Making use of (c), we explore the possibilities for $r_m$: If $r_m$ is one of \{r_2, r_3, r_4, r_5, r_7\} then, by (1), (2), and (4) for all $l \geq j$ we have $M_i(b_{2f}) = 0$. This contradicts (b). If $r_m$ is one of \{r_1, r_8\} then its output transition in $B(\text{Dekker})$ is labelled with a 'write' action, which is enabled at every $M_i$, $l \geq j$. This contradicts the justice of $C$. If $r_m = r_6$, then its output transition carries the label $\text{KR2}$ and so $\text{KR2}$ is enabled at every $M_i$, $l \geq j$ by (a). This contradicts the justice of $C$.

Since there are no other possibilities for $r_m$, we reach a contradiction. So the new formulation of liveness holds for $i = 2$.

7 Dijkstra's algorithm

The techniques we have used for Dekker's algorithm can also be applied to other algorithms. As an example, we prove in this section that Dijkstra's algorithm [13] for two processes satisfies mutual exclusion. We follow the same steps of Section 5.

We encode the two processes into programs $p_1$, $p_2$. They use four shared boolean variables, $b_1$, $b_2$, $c_1$ and $c_2$, as well as a shared variable $k$ having \{1, 2\} as its set of values. We give the code for $p_1$; the code for $p_2$ is obtained by exchanging 1 and 2 everywhere.

\[
\langle b_1 := \text{false} \rangle;
\]
\[
\text{do}
\]
\[
\langle k = 2 \rangle; \langle c_1 := \text{true} \rangle;
\]
\[
\text{do}
\]
\[
\langle k = 1 \text{ } \lor \text{ } (k = 2 \text{ } \land \text{ } b_2 = \text{false}) \rangle \text{ exit}
\]
\[
\langle k = 2 \text{ } \land \text{ } b_2 = \text{true} \rangle; \langle k := 1 \rangle \text{ exit}
\]
\[
\text{od repeat}
\]
\[
\langle k = 1 \rangle; \langle c_1 := \text{false} \rangle;
\]
\[
\text{do} \langle c_2 = \text{false} \rangle \text{ exit}
\]
\[
\langle c_2 = \text{true} \rangle;
\]
\[
\text{critical section;}
\]
\[
\langle c_1 := \text{true} \rangle; \langle b_1 := \text{true} \rangle; \langle b_1 := \text{false} \rangle \text{ exit}
\]
\[
\text{od repeat}
\]

The complete program is

\[
\text{var } b_1, b_2, c_1, c_2 : \{\text{true, false}\} \text{ init false;}
\]
\[
\text{var } k : \{1, 2\} \text{ init 1}
\]
\[
p_1 \parallel p_2
\]

Once again, if \( k = 2 \land b_2 = \text{true} \) is replaced by \( (k = 2); (b_2 = \text{true}) \), then the program may deadlock. If composite atomic actions are not available, then the atomicity of these two reads has to be enforced by introducing extra semaphores in the process expression. This is the solution adopted in [27].

The expression $P_1$ for the subprogram $p_1$ of Dijkstra's algorithm is given by the following abbreviations

\[
P_1 \equiv \overline{b_1}\overline{b_2}; P_{11}
\]
Figure 6: The box $B(P'_1 \parallel P'_2)$

\[ P_{11} \equiv kr2; c_{1wt}; P_{12} \sqcup kr1; c_{1wf}; P_{13} \ast stop \]
\[ P_{12} \equiv stop \ast kr1 \sqcup \{ kr2, b_{2rf} \} \sqcup \{ kr2, b_{2rt}; c_{1wt} \} \ast \bar{wt} \]
\[ P_{13} \equiv stop \ast c_{2rf} \sqcup c_{2rt}; enter1; exit1; c_{1wt}; b_{1wt}; b_{1wf} \]

We define

\[ \text{Dijkstra} = [N : P_1 \parallel P_2 \parallel B_1 \parallel B_2 \parallel C_1 \parallel C_2 \parallel K] \]

where $N$ are the names of the five variables.

In this case, we restrict the actions of the variables $b_1$, $b_2$, and $k$ of $B(\text{Dijkstra})$. After simplifying by the equations of Proposition 5.4, we obtain the following box expression

\[ \text{Dijkstra}' = [N : P'_1 \parallel P'_2 \parallel C_1 \parallel C_2] \]

where

\[ P'_1 = c_{1wt} \sqcup c_{1wf}; (c_{2rf} \sqcup c_{2rt}; enter1; exit1; c_{1wt}) \ast stop \]

In this case, the simplification is more significant than the one we achieved for Dekker's algorithm. The box $B(P'_1 \parallel P'_2)$ is shown in Figure 6.

The proof of mutual exclusion is very similar to that of Dekker's algorithm. We have the following invariants of $B(\text{Dijkstra}')$:

\[ \sum_{i=1}^{5} M(s_i) = 1 \quad \sum_{i=1}^{5} M(r_i) = 1 \]

(1)
\[ M(b_1 f) + M(b_1 t) = 1 \quad M(b_2 f) + M(b_2 t) = 1 \] (2)

and the following three traps:
\[ \{s_1, c_1 f\} \quad \{s_2, c_2 f\} \] (3)
\[ \{s_1, s_2, r_1, r_2, c_1 t, c_2 t\} \] (4)

Assume mutual exclusion does not hold. Then, for some reachable marking \( M \) we have \( M(s_4) = 1 \) and \( M(r_4) = 1 \) in \( B(\text{Dijkstra}') \).

For this marking \( M \), we have, by (1):
\[ M(s_1) = M(s_2) = 0 \quad M(r_1) = M(r_2) = 0 \] (5)

Then by (5) and (3)
\[ M(c_1 f) > 0 \quad M(c_2 f) > 0 \] (6)

which gives, by (2), that
\[ M(c_1 t) = 0 \quad M(c_2 t) = 0 \] (7)

But then, by (5) and (7) we get
\[ M(s_1) + M(s_2) + M(r_1) + M(r_2) + M(c_1 t) + M(c_2 t) = 0 \]
which contradicts that (4) is an initially marked trap.

8 Conclusions

We have shown how properties of Dekker’s and Dijkstra’s mutual exclusion algorithms can be proved in the box calculus using a combination of techniques from Petri nets and process algebra. Our work was inspired by that of Walker [27], who formalised these and other mutual exclusion algorithms in CCS and automatically verified properties of them with the Concurrency Workbench. This kind of verification is also possible in the box calculus (a transition system can be constructed using the operational semantics of [15]), but our goal in this paper was different; we wanted to verify properties of the algorithms with manual or computer-assisted means.

We borrowed from process algebra the notion of hiding and algebraic simplification of process terms; from Petri nets we borrowed S- and T-invariants and traps. These techniques complemented each other well in the analysis of mutual exclusion: hiding reduced Dekker’s net from 45 to 34 nodes, and Dijkstra’s from 65 to 32; this simplified the analysis with S-invariants and traps, and allowed us to focus on the relevant parts of the algorithm. Hiding could not be used in proving liveness of Dekker’s algorithm, since liveness (more precisely, its just version) holds because of a delicate interplay of all the variables. In this case, the box calculus permitted reasoning about a complicated net without ever having to draw it.

The definition of actions as sets of names provides a clean solution to the problem of selectively making hidden actions visible: new names are added to the actions we are interested in, which do not synchronise with any others. In this way, the \( b\_\text{wait} \) action of Dekker’s algorithm which models the request of access to the critical section is transformed into the action \( \{b\_\text{wait}, \text{request}_1\} \). Observe that this action models the simultaneous occurrence of the names \( b\_\text{wait} \) and \( \text{request}_1 \). If \( b\_\text{wait}; \text{request}_1 \) were used instead, as Walker does in [27], then we would have to take care (especially when using a model checker) that the results of the verification are
meaningful. For instance, in order to prove that a certain action $\alpha$ cannot occur after the access is requested, it may not be sufficient to prove that $\alpha$ cannot occur after the action request$_1$. The addition of new names actions was also applied to making all $\emptyset$ actions visible in the proof of liveness, without changing the semantics of the scoping operator.

We have used only a subset of $B(PN)^2$ and a subset of the box calculus in this paper. It was mostly done to simplify the exposition. For instance, there are no fundamental problems in allowing nested parallel blocks in $B(PN)^2$ (see [4]). The use of trace semantics was however suggested by our restrictions. In a more general setting, step equivalence or partial order equivalences will probably enjoy better properties.

Acknowledgements

We learned of the difficulties of proving liveness properties in Dekker's algorithm at a tutorial given by Mogens Nielsen at the 13th Petri Net Conference [22]. We thank Eike Best and Richard Hopkins for helpful discussions on $B(PN)^2$. Very special thanks to Rolf Walter, who spotted a mistake in a former version of the paper.

References


