Petri Net Based Deadlock Analysis of Process Algebra Programs

P. Rondogiannis and M.H.M. Cheng
Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, BC
Canada V8W 3P6
{prondo,mcheng}@csr.uvic.ca

January 25, 2001

Abstract

Recent research has been conducted on representing Process Algebra programs by safe Petri nets. We suggest that such a representation offers direct benefits: one can use methods that have been developed in the Petri net theory domain, to reason about Process Algebra programs. We propose for a subset of a specific Process Algebra (Milner's Calculus of Communicating Systems or CCS), a deadlock detection algorithm which is based on Petri net reduction techniques. The Petri net model of a CCS program is transformed into a simpler one which contains a smaller number of states, without losing however any deadlock information. An implementation of the proposed technique confirms that net-based verification of processes is a promising area of research.

1 Introduction

Petri nets (PN) are a graphical and mathematical tool that has been used extensively as a fundamental model of concurrent systems. Their theory is quite well understood and many sophisticated techniques exist that facilitate their analysis. On the other hand, Process Algebras use algebraic expressions for the specification of concurrent systems. They rely on a small set of basic operators which can be used to build systems from more elementary ones.

Recently, the two formalisms began to influence each other and to converge. More specifically, research has been conducted in finding for Process Algebra programs finite representations in terms of Petri nets [9, 10, 21, 22, 29, 11]. The main reason for such an investigation is that Petri net representations explicitly model parallelism and can in this way be understood as a concurrent semantics for Process Algebras. In this work, we suggest that such a representation implies other significant benefits as well. We propose for a specific Process Algebra (Milner's Calculus of Communicating Systems or CCS) [17, 18], a deadlock detection algorithm which is based on Petri net analysis techniques. A CCS program is first transformed into an equivalent safe Petri net. Using Petri net specific reductions, the net is simplified into one with a fewer number of states while preserving any deadlock information. The sequences of transitions that lead to deadlocks in the original CCS program, can be uncovered from the reduced net.

Related work Taylor [30] proposes an algorithm that addresses several problems related to the analysis of Ada programs. The detection of deadlock situations is performed using a state enumeration approach. More recent work [27, 20, 32, 33, 25], is based on the representation of the concurrent program (e.g., Ada) by a Petri net. In [27], the source program is modeled by a Petri net, and deadlock information is obtained through examination of the state-space of the net. An attempt that elegantly uses many results from Petri net theory in order to reduce the cost of the state enumeration approach, is given in [20]. However, the proposed method of using Petri net invariants still has a significant overhead and the reduction in the number of states is not always satisfactory. More recent work of the same group [32, 33], reaches independently to similar results as ours [25]. Both approaches have as their starting point the theory of Petri net reductions, as it is described in [7]. The context is however different: we start with an abstract model of concurrency (CCS), while they use a more concrete concurrent programming language (Ada). There are both advantages and disadvantages in the two approaches. Our method is based on the theoretically well-established concurrent semantics of CCS. This fact has an important consequence: the nets that our algorithm manipulates are very concise. In fact, the number of states of a

Petri net equals the number of states of the corresponding CCS program. On the other hand, the authors in [32, 33] propose some Ada-specific reductions, which can be more effective for specific problems. Other more recent approaches to the same problem exist: Valmari [35] uses the idea of a *stubborn set* in order to reduce the overhead due to state enumeration. An approach based on partial orders is taken by Godefroid in [14]. Finally, in [15] a theory of *optimal simulations* is developed, which however does not appear to apply to general concurrent systems.

We should emphasize here that the problem has been shown to be intractable [31] and therefore a universally "good" solution does not exist. One method may succeed in a case where another method may fail, and vice versa. We believe that a deadlock detection system, in order to be practical, should have a variety of methods at its disposal and use them as needed.

Contributions The main contributions of our work can be summarized as follows:

1. To our knowledge, it is one of the first practical uses of the concurrent semantics of CCS. It is our belief that the representation of Process Algebra programs by Petri nets may offer other benefits as well.

2. We reply affirmatively to the question posed in [22, pp. 259]: "Do structure theorems of net theory imply the success of certain strategies for process verification and construction?"

3. We present a deadlock detection algorithm which can be used to alleviate the state space explosion that appears in the analysis of concurrent programs.

The paper is self-contained and provides an introduction to CCS, Petri nets and their interelationship. The presented material is organized as follows. The next section introduces CCS and Petri nets. Section 3 concentrates on a subset of the research on the concurrent semantics of CCS. This semantics is our basis of translating CCS programs into equivalent Petri nets. We describe our approach for detecting deadlocks in CCS programs by analyzing the corresponding Petri net, in section 4. In section 5, we conclude giving statistical results of our approach and suggest pointers for future research.

2 CCS and Petri Nets

2.1 Simple CCS

In this section, the syntax of the CCS calculus is formally defined. Let $\mathcal{X}$ be the set of agent *variables* with typical elements $X, Y, ...$ Let $\mathcal{A}$ be an infinite set of *names* denoting actions, and $\overline{\mathcal{A}}$ a set of *co-names*. Typical elements of $\mathcal{A}$ are $\mathsf{a}, \mathsf{b}, \mathsf{c}, ...$ and their complements $\overline{\mathsf{a}}, \overline{\mathsf{b}}, \overline{\mathsf{c}}, ...$ are elements of $\overline{\mathcal{A}}$. Let $\mathcal{L} = \mathcal{A} \bigcup \overline{\mathcal{A}}$ be the set of action *labels*. Whenever a composite agent $P \mid Q$ is considered, we use τ to represent *any* pair $(\mathsf{b}, \overline{\mathsf{b}})$ of complementary actions by the components of the agent. In fact, τ is a distinguished *silent* action which is considered to be unobservable outside a system. The set of *actions* is defined as $Act = \mathcal{L} \bigcup \{\tau\}$. Let α and β range over Act.

Definition 2.1 The set $\mathcal{E}$ of *agent expressions* is the smallest set which includes $\mathcal{X}$ and contains the following expressions, where E, E_i are already in $\mathcal{E}$:

1. $\mathbf{0}$, the *inactive agent*
2. $\alpha.E$, a *Prefix* ($\alpha \in Act$)
3. $E_1 + E_2$, a *Summation*
4. $E_1 \mid E_2$, a *Parallel Composition*
5. $E \backslash L$, a *Restriction* ($L \subseteq \mathcal{L}$)
6. $E[f]$, a *Relabeling* (f a relabeling function)
7. $\mu X : E$, a *Recursion* ($X \in \mathcal{X}$)

In the above definition, $\mathbf{0}$ denotes an agent capable of no action whatever. In the following we will not consider the whole CCS calculus. We restrict our attention to a significant subset of CCS which does not allow agent expressions with initial parallelism in a summation or recursion context. This subset is called *simple* CCS and was first introduced in [11]. The structure of simple CCS is similar to the structure of other concurrent programming languages and therefore its expressive power is adequate for specifying practical problems. The formal definition follows:

Definition 2.2 The set $\mathcal{S}$ of *simple agent expressions* with typical element S, is the subset of agent expressions that is generated by the following grammar:

$$S ::= F \mid X \mid S|S \mid S\backslash L \mid S[f]$$
$$F ::= \mathbf{0} \mid \alpha.S \mid F+F \mid \mu X : F$$

Example 2.1 The agent expression $(b.\mathbf{0}|c.\mathbf{0})+(d.\mathbf{0})$ is not simple because it contains initial parallelism in a summation context. The agent expression $(a.\mathbf{0})+(b.\mathbf{0})$ is simple.

Additionally, we are interested in those expressions in which the agent variables do not appear free.

Definition 2.3 An occurrence of an agent variable X in an agent expression E is said to be *bound* if it occurs in E within a subexpression of the form $\mu X : Q$. Otherwise the occurrence is said to be *free*. The set of free variables in E is denoted by $free(E)$.

For agent expressions E_1, E_2 and variable X, $E_1\{E_2/X\}$ denotes the result of substituting E_2 for every free occurrence of X in E_1.

Definition 2.4 An agent expression E is *closed* if it contains no free agent variables.

A final restriction that we impose on the agent expressions ensures that the Petri nets obtained through the concurrent semantics are finite. This requirement will be further discussed in section 3.2.

Definition 2.5 An agent expression E is *regular* [22] if:

1. For every subexpression $E_1|E_2$ of E, $free(E_1) = free(E_2) = \emptyset$.

2. For every subexpression $E_1\backslash L$ of E, $free(E_1) = \emptyset$.

3. For every subexpression $E_1[f]$ of E, $free(E_1) = \emptyset$.

In the following, we consider only those agent expressions that are *simple*, *regular* and *closed*. We will often refer to them as *agents* (or programs) and will denote them using the same notation as for agent expressions (i.e., E and E_i).

The semantics of CCS as defined by Milner in [18] is based on the notion of a labeled transition system, where concurrency is viewed as a special case of sequentiality—concurrency is achieved by *interleaving*. To capture the intended concurrency in CCS expressions, we adopt a non-interleaving semantics of CCS based on Petri nets in section 3.

2.2 Safe Petri Nets

In the literature, several different versions of Petri nets have been proposed, depending on the level of detail at which one wishes to describe a specific system. The notation we use in this paper, allows us to relate Petri nets with Process Algebras, as well as to accommodate the nets that result from the reductions that will be introduced later on. Let Act be the set of actions, as this was defined in section 2.1 and Act^* the set of non-empty sequences of actions. Then:

Definition 2.6 A *Petri net* (or simply *net*) over Act is a structure $R = (Pl, T, M_0)$ where:

1. Pl is a possibly infinite set of *places*.

2. T is a set over $\Delta(Pl) \times Act^* \times \Delta(Pl)$ and represents the *transition relation*.

3. $M_0 \in \Delta(Pl)$ is the *initial marking*.

Here $\Delta(Pl)$ denotes the set of all non-empty, finite subsets of Pl. An element $(I, u, O) \in T$ is called a transition and will usually be written as $I \xrightarrow{u} O$. For a transition $t = I \xrightarrow{u} O$, its *preset* or *input* is given by $pre(t) = I$, its *postset* or *output* is $post(t) = O$ and its *action* (also called *action sequence*) is $act(t) = u$. Similarly, for $p \in Pl$, $pre(p)$ denotes the set of transitions that have p in their postset and $post(p)$ is the set of transitions that have p in their preset.

Example 2.2 The following is an example of a Petri net $R = (Pl, T, M_0)$:

$$\begin{aligned} Pl &= \{p_1, p_2, p_3, p_4\} \\ T &= \{(\{p_1\}, a, \{p_3\}),\ (\{p_1\}, b, \{p_4\}),\ (\{p_2\}, c, \{p_4\})\} \\ M_0 &= \{p_1, p_2\} \end{aligned}$$

Petri nets are usually represented graphically in the following way: places $p \in Pl$ are represented as circles $\bigcirc$ with their name p outside, and transitions $t = \{p_1, \ldots, p_n\} \xrightarrow{u} \{p_{n+1}, \ldots, p_{n+m}\}$ are represented as boxes $\boxed{u}$ carrying the label u inside and connected via directed arcs to the places in $pre(t)$ and $post(t)$ as shown in Figure 1. The initial marking M_0 is represented by putting a dot $\bullet$ into the circle of each place in M_0.

Example 2.3 Using the above conventions, the net of Example 2.2 is represented as shown in Figure 2.

2.3 Execution Rules for Petri Nets

The dynamic behavior of a Petri net, is accomplished through the execution of transitions. Although in the initial marking of a Petri net only single tokens are allowed for each $p \in M_0$ (i.e., M_0 is a set), the execution of transitions may result in places having more than one tokens. Therefore, in a marking M of a Petri net, a place p appears as many times as the number of tokens that it holds. Formally:

Definition 2.7 A *marking* M of a Petri net $R = (Pl, T, M_0)$ is a multiset over Pl.

where a *multiset* is a set in which multiple occurrences of elements are allowed. Let $\sqsubseteq$, $\sqcup$ and $-$ denote multiset inclusion, union and difference respectively. Then the execution of a transition is defined as follows:

Definition 2.8 Let $R = (Pl, T, M_0)$ be a net, $t = I \xrightarrow{u} O$ a transition of R and M be a marking of R. Then:

1. Transition t is *enabled* at M if $I \sqsubseteq M$.

2. If enabled at M, the *execution* or *firing* of t transforms M into a new marking M_1 of R, and $M_1 = (M - I) \sqcup O$. In symbols: $M \xrightarrow{t} M_1$.

Example 2.4 In Example 2.3, initially all transitions are enabled. The execution of the transition carrying action b results in the marking $M_1 = \{p_2, p_4\}$. The new situation is represented graphically in Figure 3(a). Under the new marking, the transition carrying action c is still enabled. Its execution results in the marking $M_2 = \{p_4, p_4\}$. The new situation is represented in Figure 3(b).

A marking that can be reached by successive executions of transitions is called a reachable marking. Formally:

Definition 2.9 A *reachable marking* is a marking M for which there exist intermediate markings $M_1, \ldots, M_n$ and transitions $t_1, \ldots, t_n$ with $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n = M$.

In Figure 3(b), place p_4 contains two tokens. A safe Petri net is one in which such a situation does not arise:

Definition 2.10 A net R is *safe* if and only if in every reachable marking, the number of tokens per place is either zero or one.

In the following, all the Petri nets considered will be assumed to be safe, except if explicitly stated otherwise. The *reachability graph* is a tool that has been used for the analysis of Petri nets. Intuitively, a node of the reachability graph corresponds to a reachable marking of the Petri net, and an edge between two nodes corresponds to a transition firing which transforms one marking into another. Formally:

Definition 2.11 The *reachability graph* of a Petri net $R = (Pl, T, M_0)$, is a graph $RG = (V, E)$ where $V = \{M \,:\, M \text{ is reachable from } M_0\}$ and $E = \{(M_1, M_2) \,:\, M_1, M_2 \in V \wedge \exists t \in T,\ M_1 \xrightarrow{t} M_2\}$.

Figure 4 represents the reachability graph of the Petri net in Figure 2. Labels t_1, t_2 and t_3 indicate the transitions with actions a, b and c respectively.

3 Concurrent Semantics for Simple CCS

This section presents a concurrent semantics for simple CCS agents using Petri nets. A concurrent semantics of the whole CCS is still an interesting research problem. However, for simple CCS all the existing semantic results coincide. Moreover, as we noted earlier, the structure and expressive power of simple CCS is adequate for specifying practical problems. In the following, the terms CCS and simple CCS are used interchangeably.

The main idea of a non-interleaving semantics of CCS is to decompose each agent E into a set $\{C_1, ..., C_n\}$ of sequential components, which can be thought of as working concurrently. Sequential components correspond to places of Petri nets. Consequently, net transitions are now of the form

$$\{C_1, ..., C_n\} \xrightarrow{u} \{C_{n+1}, ..., C_{n+m}\}$$

where $\{C_1, ..., C_{n+m}\}$ are sequential components and u is an action of the agent. The decomposition idea was first proposed in [9] and also used in [10],[21],[22], [29]and [11]. In the following, the details of the decomposition are presented and subsequently the transition relation of the Petri net is defined by induction on the syntactic structure of sequential components. Our presentation follows the ideas of [11].

3.1 The Decomposition

In the following the *sequential components* and the *decomposition function* are formally defined:

Definition 3.1 The set *Seq* of *sequential components*, with typical element G, consists of all terms generated by the following rules:

$$G ::= \mathbf{0} \ | \ \alpha.E \ | \ E+E \ | \ id|G \ | \ G|id \ | \ E\backslash L \ | \ E[f] \ | \ \mu X : E$$

where E is an agent and $\backslash L$, $[f]$ have their standard CCS meaning.

A sequential component is in fact a subagent of an agent equipped with some extra information concerning the environment in which the subagent operates. There exists an operator on sequential components for all CCS operators apart from parallel composition. This one is replaced by two unary operators, $|id$ and $id|$, which are used as tags for showing which parts of an agent are working concurrently. In the following, we use I, I_1, $I_2, ...$ to denote sets of sequential components. An agent can be decomposed into a set of sequential components using the following function *dec*:

Definition 3.2 Function *dec* decomposes a CCS agent into a set of sequential components as follows:

$$
\begin{array}{lcl}
dec(\mathbf{0}) & = & \{\mathbf{0}\} \\
dec(\alpha.E) & = & \{\alpha.E\} \\
dec(E_1 + E_2) & = & \{E_1 + E_2\} \\
dec(E_1|E_2) & = & dec(E_1)|id \cup id|dec(E_2) \\
dec(E\backslash L) & = & dec(E)\backslash L \\
dec(E[f]) & = & dec(E)[f] \\
dec(\mu X : E) & = & \{\mu X : E\}
\end{array}
$$

where $\backslash L$, $id|$, $|id$ and $[f]$ are extended to apply also to sets, and in this case they are understood elementwise.

Example 3.1 Consider the agent $(\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0})$. Its decomposition gives a set of two sequential components:

$$
\begin{array}{lcl}
dec(\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0}) & = & dec(\mathsf{a}.\mathbf{0})|id \cup id|dec(\mathsf{b}.\mathbf{0}) \\
 & = & \{(\mathsf{a}.\mathbf{0})|id\} \cup \{id|(\mathsf{b}.\mathbf{0})\} \\
 & = & \{(\mathsf{a}.\mathbf{0})|id\,,\ id|(\mathsf{b}.\mathbf{0})\}
\end{array}
$$

Example 3.2 Consider the agent $(\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0})|\mathsf{c}.\mathbf{0}$. Its decomposition gives a set of three sequential components:

$$
\begin{array}{lcl}
dec(((\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0})|\mathsf{c}.\mathbf{0})) & = & dec((\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0}))|id \cup id|dec(\mathsf{c}.\mathbf{0}) \\
 & = & \{((\mathsf{a}.\mathbf{0})|id)|id\,,\ (id|(\mathsf{b}.\mathbf{0}))|id\} \cup \{id|(\mathsf{c}.\mathbf{0})\} \\
 & = & \{((\mathsf{a}.\mathbf{0})|id)|id\,,\ (id|(\mathsf{b}.\mathbf{0}))|id\,,\ id|(\mathsf{c}.\mathbf{0})\}
\end{array}
$$

3.2 Concurrent Semantics

The next definition gives the concurrent semantics for CCS agents using Petri nets. More specifically, to each agent E, a Petri net $\mathcal{R}[E]$ is assigned. The places of the net are sequential components, its initial marking consists of the initial decomposition $dec(E)$ of E and its transitions are derived using rules analogous to those of the transitional semantics.

Definition 3.3 Let E be an agent. Then, its *concurrent semantics* is defined as the Petri net

$$\mathcal{R}[E] = (Seq, Tr, dec(E))$$

where Tr is the set of transitions derived by the following rules:

(Act) $\{\alpha.E\} \xrightarrow{\alpha} dec(E)$

(Sum₁) $\{E_1\} \xrightarrow{\alpha} I_1$ **implies** $\{E_1 + E_2\} \xrightarrow{\alpha} I_1$

(Sum₂) $\{E_2\} \xrightarrow{\alpha} I_2$ **implies** $\{E_1 + E_2\} \xrightarrow{\alpha} I_2$

(Com₁) $I \xrightarrow{\alpha} I'$ **implies** $I|id \xrightarrow{\alpha} I'|id$

(Com₂) $I \xrightarrow{\alpha} I'$ **implies** $id|I \xrightarrow{\alpha} id|I'$

(Com₃) $I_1 \xrightarrow{1} I_1'$ **and** $I_2 \xrightarrow{\overline{1}} I_2'$ **implies** $I_1|id \cup id|I_2 \xrightarrow{\tau} I_1'|id \cup id|I_2'$

(Res) $I \xrightarrow{\alpha} I'$ **implies** $I\backslash L \xrightarrow{\alpha} I'\backslash L$ **where** $\alpha, \overline{\alpha} \notin L$

(Rel) $I \xrightarrow{\alpha} I'$ **implies** $I[f] \xrightarrow{f(\alpha)} I'[f]$

(Rec) $\{E\{(\mu X : E)/X\}\} \xrightarrow{\alpha} I'$ **implies** $\{\mu X : E\} \xrightarrow{\alpha} I'$

Remark: Notice that the set Seq in the definition of $\mathcal{R}[E]$, is infinite. As we are interested only in finite nets, we restrict attention to the subset of Seq that consists of all places reachable from the initial marking. See also [22] for a further discussion on this point.

The above definition is now illustrated with some examples.

Example 3.3 Consider the agent $\mu X : (\mathsf{a}.X)$. We construct its corresponding Petri net. The initial marking of the net is

$$M_0 \;=\; dec(\mu X : (\mathsf{a}.X)) \;=\; \{\mu X : (\mathsf{a}.X)\}$$

The only rule that can be used is the one for **Rec**, which gives the only possible transition:

$$\{\mu X : (\mathsf{a}.X)\} \xrightarrow{\mathsf{a}} \{\mu X : (\mathsf{a}.X)\}$$

The corresponding Petri net is shown in Figure 5(a).

Example 3.4 The following example shows the main difference between the transitional and the concurrent semantics. Consider the process term $\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0}$. From example 3.1

$$M_0 \;=\; dec(\mathsf{a}.\mathbf{0}|\mathsf{b}.\mathbf{0}) \;=\; \{(\mathsf{a}.\mathbf{0})|id,\, id|(\mathsf{b}.\mathbf{0})\}$$

The only rules that are applicable are $\mathbf{Com_1}$ and $\mathbf{Com_2}$, which give two transitions:

$$\{(\mathsf{a}.\mathbf{0})|id\} \xrightarrow{\mathsf{a}} \{\mathbf{0}|id\}$$
$$\{id|(\mathsf{b}.\mathbf{0})\} \xrightarrow{\mathsf{b}} \{id|\mathbf{0}\}$$

The corresponding Petri net is shown in Figure 5(b). This Petri net explicitly shows the inherent parallelism of the agent.

Example 3.5 Consider the agent $\mathsf{a}.\mathbf{0}+\mathsf{b}.\mathbf{0}$. The initial marking of the corresponding Petri net is

$$M_0 \;=\; dec(\mathsf{a}.\mathbf{0}+\mathsf{b}.\mathbf{0}) \;=\; \{\mathsf{a}.\mathbf{0}+\mathsf{b}.\mathbf{0}\}$$

Starting from M_0 and using the rules $\mathbf{Sum_1}$ and $\mathbf{Sum_2}$, we get the following two possible transitions:

$$\{\mathsf{a}.\mathbf{0}+\mathsf{b}.\mathbf{0}\} \xrightarrow{\mathsf{a}} \{\mathbf{0}\}$$
$$\{\mathsf{a}.\mathbf{0}+\mathsf{b}.\mathbf{0}\} \xrightarrow{\mathsf{b}} \{\mathbf{0}\}$$

The corresponding Petri net is shown in Figure 5(c).

Example 3.6 As a final example on the concurrent semantics, consider the agent $\mathsf{a}.\mathbf{0}|\bar{\mathsf{a}}.\mathbf{0}$. Using the decomposition function *dec* we get the initial marking of the corresponding Petri net:

$$M_0 \;=\; dec(\mathsf{a}.\mathbf{0}|\bar{\mathsf{a}}.\mathbf{0}) \;=\; \{(\mathsf{a}.\mathbf{0})|id,\, id|(\bar{\mathsf{a}}.\mathbf{0})\}$$

Starting from the initial marking, three transitions are applicable:

$$\{(\mathsf{a}.\mathbf{0})|id\} \xrightarrow{\mathsf{a}} \{\mathbf{0}|id\}$$
$$\{id|(\bar{\mathsf{a}}.\mathbf{0})\} \xrightarrow{\bar{\mathsf{a}}} \{id|\mathbf{0}\}$$
$$\{(\mathsf{a}.\mathbf{0})|id\,,\; id|(\bar{\mathsf{a}}.\mathbf{0})\} \xrightarrow{\tau} \{\mathbf{0}|id\,,\; id|\mathbf{0}\}$$

The corresponding Petri net is shown in Figure 5(d).

It is important at this point to note that the subset of CCS that we consider, always results in finite Petri nets, i.e., nets with a finite number of places. More specifically, it can be shown [22, page 63] that:

Theorem 3.1 For every closed regular agent expression E, the corresponding Petri net $\mathcal{R}[E]$ is finite.

Considering only regular processes is not a significant restriction for practical purposes. Many interesting concurrent programs are finite state in nature, but their state space is often huge. This makes verification of this kind of systems a problem in its own right [2]. In this paper, we consider many well-known examples which can be described by the subset of CCS we have considered. As a last remark on the concurrent semantics of agents, we have the following theorem [29, page 143]:

Theorem 3.2 For every agent E, the corresponding Petri net $\mathcal{R}[E]$ is safe.

4 Detecting Deadlocks in CCS Agents

A deadlock in a CCS agent is a state of the agent in which no further action is possible. In general, it is not easy to find the deadlocks without exploring all possible states of an agent. However, the number of states can be quite large, making it difficult or impossible even for a computer to examine all of them. This has led to the observation [12] that "proof of absence of deadlock, even for quite simple processes, will remain the responsibility of the designer of concurrent systems". In this section, a technique is developed that allows detection of all deadlocks of a CCS agent. The proposed approach uses Petri net analysis techniques in order to reduce the states that should be examined for deadlock.

In the following, the notion of deadlock in a CCS agent is formalized, and the deadlock detection algorithm, which is based on Petri net analysis and transformation techniques, is presented and illustrated by examples.

4.1 Introduction

In order to use techniques from Petri net theory for detecting deadlocks in CCS agents, the notion of deadlock in a Petri net should first be defined:

Definition 4.1 A reachable marking M of a Petri net $R = (Pl, T, M_0)$ is a *deadlock marking* if and only if no transition $t \in T$ is enabled in M.

The translation of a CCS agent to a Petri net actually maps the states of the agent to markings of the net [29, page 143]. Therefore, instead of examining the states of the agent, one can examine the markings of the net. This means that:

Definition 4.2 A CCS agent contains a *deadlock* if and only if the corresponding Petri net contains a *deadlock marking*.

We should note here that the notion of deadlock we adopt is rather general, as it does not distinguish from the possibility of correct termination. This point will be further exemplified in section 4.6. Deadlock markings of a net can be identified by examining its reachability graph:

Proposition 4.1 A sink node in the reachability graph of a Petri net indicates the existence of a deadlock marking.

Proof: Immediate by the definition of the reachability graph. ■

In general, the reachability graph of a Petri net is not finite. However, in the case of safe and finite Petri nets—which are the ones we are interested in here—the graph is finite:

Proposition 4.2 The reachability graph of a safe and finite Petri net is finite.

Proof: Immediate by the fact that the number of vertices of the reachability graph is equal to the number of reachable markings. ■

The above propositions suggest a straightforward way to detect deadlocks in CCS agents: the agent is transformed into a safe Petri net, its reachability graph is computed and finally sink nodes are located which correspond to the deadlocks of the agent. This method is illustrated in Figure 6 for the CCS agent `a.b.0|c.d.0`.

Unfortunately, the procedure just described, does nothing more than examining all possible states of the CCS agent, and is clearly inefficient for this reason. However, looking at the Petri net of Figure 6, we see that when transition t_1 fires it causes transition t_3 to fire after some time. The same is the case for transitions t_2 and t_4. This means that each of these pairs of transitions can be represented by a single transition and consequently we can modify the Petri net to look like the one shown in Figure 7, without affecting its deadlock states. We can then examine the reachability graph of the reduced Petri net which is smaller than the initial one but still contains the required deadlock information. In the following section, we formalize the transformations we are going to use in order to reduce the size of the reachability graph, and thus simplify in this way the task of deadlock detection.

4.2 Reduction Techniques

In order to ease the analysis of Petri nets, a number of transformations have been proposed in the literature ([4, 5, 6, 7, 16, 8, 19, 28, 34, 3, 36]), which simplify (or refine) the net while preserving some of its important properties. In general, different transformations preserve different properties of the initial Petri net, and aim at different goals. The transformations we have adopted preserve safeness and deadlock freedom and allow, as we are going to show, an elegant treatment of deadlock detection in CCS agents. More specifically, we use *post-fusion* and *pre-fusion* of transitions, as well as *elimination of redundant places*

[6]. However the reductions we adopt are simplified versions of the ones that appear in [6]. This is because we aim at an algorithm for deadlock detection that should be easily implementable and efficient.

Before giving formal definitions, we discuss the intuition behind the transformations. Elimination of redundant places consists of the removal of places whose marking is always sufficient to allow firings of transitions connected to them. This kind of transformation does not modify the functioning of the net. On the other hand, fusions of transitions have been defined in order to make indivisible some transition sequences representing actions which may fire more or less at the same time. They are based on the fact that *it is not mandatory for a transition to fire as soon as it can fire.* Although the formal definitions of the transformations that are given below seem complicated, their intuition is simple and can be easily understood by the accompanying figures.

Definition 4.3 Let $R = (Pl, T, M_0)$ be a safe Petri net. A non-empty subset F of T is *post-fusable* with $h \in T$ if and only if there exists a place $p \in Pl$ such that the following conditions are satisfied:

1. $\forall f \in F,\ pre(f) = \{p\}$.
   (The only input of f is p).

2. $\forall f \in F,\ p \notin post(f)$.
   (Place p is not an output of f).

3. $\exists f \in F,\ |post(f)| > 0$.
   (There exists a transition in F which has at least one output place).

4. $p \notin pre(h)$.
   (Place p is not an input of h).

5. $p \in post(h)$.
   (Place p is an output of h).

6. $\forall t \in (T - \{h\} - F),\ p \notin pre(t) \wedge p \notin post(t)$

   (Except for h and the transitions in F, no other transition is connected to p).

7. $p \notin M_0$.

   (Place p holds no token initially).

8. $\forall f \in F, post(f) \cap post(h) = \emptyset$.

   (The transitions in F do not have any common output place with h).

Whenever the above conditions hold, the Petri net can be modified according to the following transformation:

Definition 4.4 Let $R = (Pl, T, M_0)$ be a safe Petri net, and let $F \subseteq T$, $h \in T$ and $p \in Pl$ satisfy the conditions of Definition 4.3. Then, the system resulting by the post-fusion of F and h is $R^{'} = (Pl^{'}, T^{'}, M_0)$, with:

1. $Pl^{'} = Pl - \{p\}$

2. $T^{'} = (T - \{h\} - F) \cup F^{'}$ with $F^{'}$ defined as:

$$\begin{aligned} \{hf_i :\ & f_i \in F \ and \\ & pre(hf_i) &=& \ pre(h), \\ & post(hf_i) &=& \ (post(h) - \{p\}) \cup post(f_i), \\ & act(hf_i) &=& \ act(h)act(f_i)\ \} \end{aligned}$$

where hf_i denotes the concatenation of h with f_i and $act(h)act(f_i)$ denotes the concatenation of $act(h)$ with $act(f_i)$.

Remark 1: In definition 4.3, condition 8 is used to ensure that no multiple arcs will be created from a transition to a place, during the post-fusion. For a more general definition of post-fusion, the reader is referred to [7].

Remark 2: In general, it is possible that during a post-fusion, a transition is created that already exists in the net. There are two ways to cope with this situation. The first one is to assume that transitions are multisets, and replace the union in step 2 of definition

4.4 by a multiset union. The other way (which is the one we adopt) is to assume that transitions are sets, and make sure that by disposing the duplicate transitions, no deadlock information is lost. It is possible for two transitions with the same preset, the same postset and the same action sequence, to contain different deadlock information. This is the case when the action sequences of the two transitions contain silent actions τ. An occurrence of τ in the action sequences of the two transitions may correspond to different pairs of events $(a, \overline{a})$ and $(b, \overline{b})$ with $a \neq b$. Therefore, we adopt the convention that two transitions are the same, if they have the same preset, the same postset and their action sequences are indistinguishable when the silent actions that occur in them are replaced by the pairs of events that they correspond to.

Definition 4.4 is illustrated in Figure 8(a) for the case $F = \{f\}$ and in Figure 8(b) for the case $F = \{f_1, f_2\}$. The intuition behind post-fusion is that whenever h fires, it is mandatory for one of the $f_i \in F$ to fire after some time. This is because the firing of h puts a token in place p which enables the transitions in F. Therefore, it is reasonable to remove the intermediate place p and fuse h with each $f_i \in F$.

Definition 4.5 Let $R = (Pl, T, M_0)$ be a safe Petri net. A non-empty subset F of T is *pre-fusable* with $h \in T$ if and only if there exists a place $p \in Pl$ such that the following conditions are satisfied:

1. $post(h) = \{p\}$.
   (The only output of h is p).

2. $p \notin pre(h)$.
   (Place p is not an input of h).

3. $|pre(h)| > 0$.
   (Transition h has at least one input).

4. $\forall f \in F,\ p \in pre(f)$.
   (Every transition of F has p in its input).

5. $\forall f \in F,\ p \notin post(f)$.

   (No transition of F has p in its output).

6. $\forall t \notin (\{h\} \cup F), p \notin pre(t) \wedge p \notin post(t)$.

   (Except for h and the transitions in F, no other transition is connected to p).

7. $p \notin M_0$

   (Place p holds no token initially.)

8. $\forall q \in pre(h),\ \forall t \neq h,\ q \notin pre(t)$.

   (Transition h does not share its input).

Whenever the above conditions hold, the Petri net can be modified according to the following transformation:

Definition 4.6 Let $R = (Pl, T, M_0)$ be a safe Petri net, and let $F \subseteq T$, $h \in T$ and $p \in Pl$ satisfy the conditions of Definition 4.5. Then, the system resulting by the pre-fusion of F and h is $R' = (Pl', T', M_0)$, with:

1. $Pl' = Pl - \{p\}$

2. $T' = (T - \{h\} - F) \cup F'$ with F' defined as:

$$
\begin{array}{lllll}
\{hf_i : & f_i \in F \ and & & \\
 & pre(hf_i) & = & (pre(f_i) - \{p\}) \cup pre(h) \\
 & post(hf_i) & = & post(f_i), \\
 & act(hf_i) & = & act(h)act(f_i)\ \}
\end{array}
$$

where hf_i denotes the concatenation of h with f_i and $act(h)act(f_i)$ denotes the concatenation of $act(h)$ with $act(f_i)$.

Definition 4.6 is illustrated in Figure 9(a) for the case $F = \{f\}$ and in Figure 9(b) for the case $F = \{f_1, f_2\}$. The intuition behind pre-fusion is that whenever h fires, some $f_i \in F$ *may* fire after some time. However, this is not guaranteed as in the case of post-fusion.

It is this difference that makes, as we are going to see, the treatment of pre-fusion more difficult than that of post-fusion.

The last category of reductions, namely *elimination of redundant places*, is introduced below:

Definition 4.7 Let $R = (Pl, T, M_0)$ be a safe Petri net. A place $p \in Pl$ is called *redundant* if and only if there exist transitions $t_0, ..., t_n$ and places $p_0, ..., p_{n-1}$ such that the following conditions are satisfied:

1. $pre(p) = \{t_0\}$
   (The only input of p is t_0).

2. $post(p) = \{t_n\}$
   (The only output of p is t_n).

3. $pre(p_i) = \{t_i\},\ i = 0, .., n-1$
   (The only input of each p_i is t_i).

4. $post(p_i) = \{t_{i+1}\},\ i = 0, ..., n-1$
   (The only output of each p_i is t_{i+1}).

5. $p, p_0, \ldots, p_{n-1} \notin M_0$
   (The places $p, p_0, \ldots, p_{n-1}$ are not initially marked).

Definition 4.8 Let $R = (Pl, T, M_0)$ be a safe Petri net, and let $p, p_0, ..., p_{n-1} \in Pl$ and $t_0, ..., t_n \in T$ satisfy the conditions of Definition 4.7. Then, the net resulting from the elimination of p is $R' = (Pl', T', M_0)$, with:

1. $Pl' = Pl - \{p\}$

2. $T' = (T - \{t_0, t_n\}) \cup \{t_0', t_n'\}$ where

$$\begin{aligned} t_0' &= (pre(t_0), act(t_0), post(t_0) - \{p\}) \\ t_n' &= (pre(t_n) - \{p\}, act(t_n), post(t_n)) \end{aligned}$$

Figures 10(a) and 10(b) illustrate the above definitions for $n = 1$ and $n = 2$ correspondingly. The above transformations preserve the safeness and deadlock freedom of a Petri net [5, pages 361-367]. Formally:

Theorem 4.1 Let $R = (Pl, T, M_0)$ be a Petri net and $R^{'} = (Pl^{'}, T^{'}, M_0)$ be the net resulting from a sequence of the above transformations. Then, $R^{'}$ is deadlock-free (safe) if and only if R is deadlock-free (safe).

What the above theorem suggests in our case is that given an agent P, one need not analyze the reachability graph of the corresponding Petri net R. It suffices to analyze the reachability graph of the net $R^{'}$ that results from R after applying to it a sequence of the above transformations. If $R^{'}$ is found to contain a deadlock, the same will be the case for R. On the other hand, proving that $R^{'}$ is deadlock-free implies that R is deadlock-free.

4.3 Identifying Sequences that Lead to Deadlock

In general, it is not sufficient just to detect that a deadlock exists; it is also crucial to know what sequences of transitions (or actions) have led to deadlock. This is very important for the designer of a system, because it can help to identify the flaws in the design and possibly correct them. This section investigates the relation between the sequences of actions that lead to deadlock in the reduced Petri net and those that lead to deadlock in the initial CCS program. Certain of the results given in the following have been motivated by the work in [7], which examines the preservation of properties in a more general setting. Let $R_0 = (Pl_0, T_0, M_{00})$ denote the initial Petri net and $R_i = (Pl_i, T_i, M_{0i})$ be the resulting net after a sequence of i reductions ($i \geq 0$) on R_0. We assume R_0 is safe. By Theorem 4.1, R_i is also safe. Our first result below suggests that if the reductions are all post-fusions then a sequence of transitions that leads to deadlock in the reduced Petri net also leads to deadlock in the initial one.

Theorem 4.2 Let R_1 be the net resulting from the safe Petri net R_0 after a post-fusion of transitions F with transition h, and let σ be a sequence of transitions leading R_1 to a deadlock marking $M^{'}$. Then σ leads R_0 to a deadlock marking M, and $M = M^{'}$.

Proof: Every sequence of transitions of R_1 is also a sequence of transitions of R_0, since every transition of R_1 is either a transition of R_0 or can be decomposed in two transitions that can fire one after another in R_0. Also, the firing of any $t \neq hf_i$ in R_1 has the same effect as the firing of t in R_0. The firing of any hf_i, $f_i \in F$, in R_1 has the same effect as firing first h and then f_i in R_0. Therefore, a transition sequence of R_1 can be followed in R_0 and leads exactly to the same marking. Let now σ be a sequence of transitions leading R_1 to deadlock and suppose that R_0 is not deadlocked after the occurrence of σ. Then, a transition t may occur. If $t \neq h$ and $t \neq f_i$ then R_1 can also perform t, as the preset and postset of t have not been changed by the reduction. Therefore R_1 is not deadlocked. If $t = h$, then one of transitions $f_i \in F$ can also fire in R_0. But then in R_1, one of transitions hf_i can fire as well, and therefore R_1 is not deadlocked. The case $t = f_i$ does not arise because it implies that place p in R_0 contains a token; this is not possible because we know that R_0 and R_1 have the same marking after the firing of σ. ■

Corollary 4.1 Let R_i be the Petri net resulting from R_0 after a sequence of post-fusions of transitions and let σ be a sequence of transitions leading R_i to a deadlock marking $M^{'}$. Then σ also leads R_0 to a deadlock marking M, and $M = M^{'}$.

However the situation is not as straightforward when the pre-fusion rule is used. Before stating a general theorem we give an example of the problems that arise with pre-fusion.

Example 4.1 Consider the following process term:

$$(\mathsf{a.c.0}|(\mathsf{b.\overline{c}.0} + \mathsf{d.0}))\backslash\{\mathsf{c}\}$$

that yields the Petri net of Figure 11(a). It can be easily checked that the only deadlock markings are $M_1 = \{p_3, p_6\}$ and $M_2 = \{p_5, p_6\}$. The net is reduced using the pre-fusion rule on transitions t_1 and t_4, yielding the Petri net of Figure 11(b). In the reduced net, the sequence $\sigma = t_3$ leads to the deadlock marking $M = \{p_1, p_6\}$. In this case however, transition t_3 by itself does not lead to deadlock in the initial Petri net. This is because the pre-fusion rule has been used. However it leads 'very close' to deadlock because if t_1 fires after t_3 we get deadlock M_1.

The idea behind the above example will be formalized by the following two theorems:

Theorem 4.3 Let R_1 be the net resulting from the safe Petri net R_0 after a pre-fusion of transitions F with transition h and let σ be a sequence of transitions leading R_1 to a deadlock marking $M^{'}$. Then, either σ leads R_0 to the same deadlock marking $M^{'}$, or σh leads R_0 to the deadlock marking $(M^{'} - pre(h)) \sqcup \{p\}$ where $post(h) = \{p\}$.

Proof: For the same reasons as for post-fusion, every sequence of transitions of R_1 that leads to a marking $M^{'}$, is also a sequence of R_0 and leads to the same marking $M^{'}$ in R_0. However, when a sequence σ leads R_1 to deadlock, it may not immediately lead R_0 to deadlock. Consider R_0 after the occurrence of σ, and suppose that it is not deadlocked. Then, a transition t may occur. If $t \neq h$ and $t \neq f_i \in F$, then R_1 can also perform t, as the preset and postset of t have not been changed by the reduction. Therefore R_1 is not deadlocked. The case $t = f_i$ does not arise because it implies that place p in R_0 contains a token; this is not possible because we know that R_0 and R_1 have the same marking after the firing of σ. The only possible additional firing that can occur in R_0 is that of $t = h$. After that, no other firing can take place in R_0, because if any f_i was enabled, the corresponding hf_i would be able to fire in R_1.

Summarizing, two cases are only possible: either σ leads R_0 to the same deadlock marking $M^{'}$ or σh leads R_0 to a deadlock M that is one step further than $M^{'}$. By the rule of firing of a transition, $M = (M^{'} - pre(h)) \sqcup post(h)$. But $post(h) = \{p\}$, and this completes the proof. ■

Referring again to example 4.1, we see that the deadlock marking $\{p_3, p_6\}$ of the reduced Petri net can be obtained by the deadlock marking $\{p_1, p_6\}$ of the initial net. Clearly, $\{p_3, p_6\} = (\{p_1, p_6\} - \{p_1\}) \sqcup \{p_3\}$.

Theorem 4.4 Let R_i be the net resulting from the safe Petri net R_0 after a sequence of pre-fusions of transitions, and let σ be a sequence of transitions leading R_i to deadlock. Then, there exist transitions $w_i, \ldots, w_1$ such that $\sigma w_i \ldots w_1$ leads R_0 to deadlock.

Proof: Let h_m and F_m denote the transitions that are pre-fused in the m-th step of the reductions. We show that there exists a sequence $\sigma w_i \dots w_1$ that leads R_0 to deadlock, such that $\forall m \in \{1, \dots, i\}$, either $w_m = h_m$ or w_m is empty. The proof is by induction on the number i of pre-fusions. The induction basis ($i = 1$) holds trivially using Theorem 4.3. Assume that the result holds for all sequences of pre-fusions of length i or less. Then, the induction step considers the case where R_0 reduces to R_{i+1} after a sequence of $i+1$ pre-fusions. Applying the induction hypothesis on the sequence of i reductions from R_1 to R_{i+1}, we get that $\sigma w_i \dots w_2$ leads R_1 to deadlock. The path $\sigma w_i \dots w_2$ leads R_0 to exactly the same markings as R_1. The only additional transition that may fire is h_1, and therefore there exists a sequence $\sigma w_i \dots w_1$ that leads R_0 to deadlock. ■

The above theorem suggests that if the h_m's are stored during the reductions, one can later restore the sequences that lead to deadlock in R_0, by extending the sequences that lead to deadlock in R_i. This can be done by checking whether after σ has been followed in R_0, transitions $h_i, \dots, h_1$ can fire (in this order).

Theorem 4.5 Let R_1 be the Petri net resulting from R_0 after eliminating redundant place p and let σ be a sequence of transitions leading R_1 to a deadlock marking M'. Then σ also leads R_0 to a deadlock marking M, and $M \supseteq M'$.

Proof: The elimination of a redundant place does not alter the firing conditions of transitions. Therefore, R_1 will have the same behavior as R_0 and σ leads R_0 to deadlock. A deadlock marking of R_0 will either be the same as that of R_1 or it will additionally contain the eliminated place p. ■

Corollary 4.2 Let R_i be the Petri net resulting from R_0 after a sequence of eliminations of redundant places. Then if σ leads R_i to deadlock, it also leads R_0 to deadlock.

4.4 The Algorithm

Based on the theory presented, the proposed approach for deadlock detection in CCS agents can be developed. An informal description of the algorithm is given below:

The input of the algorithm, a CCS agent, is initially transformed into its corresponding Petri net. This is accomplished in the following way. The function *dec* is first computed, and this results in a set of places that hold the initial marking of the net. Places are then considered individually or in pairs, to apply the rules of the concurrent semantics. It should be noted that we do not need to consider more than two places at a time, as communication in CCS allows at most two actions to be composed in parallel. The application of the transition rules results in new places, which must again be considered together with the already existing ones. This process terminates as the subset of CCS we are considering results in finite nets.

In order to reduce the number of states that have to be searched for deadlock, a sequence of transformations is applied on the resulting net. Initially, redundant places are removed. From Figure 10, it is obvious that the elimination of redundant places reduces in general the number of inputs and outputs of transitions. This fact increases the probability that the new net will contain pre-fusable or post-fusable transitions. The Petri net is then searched for fusable transitions. This process is repeated until a final irreducible net is obtained.

Then, the net's reachability graph is computed, the possible deadlocks are identified and paths leading to those deadlocks are detected. However, this is not enough. We are interested in the paths that lead to deadlock in the initial Petri net, not the reduced one. Such information would allow the designer of a system to identify the deadlocks, and modify the system in order to avoid them. Thus, we use the theory that was developed in the previous section to extend the paths that have been found, getting in this way the paths that lead to deadlock in the initial Petri net. The algorithm is presented below.

Input: An Agent P.
Output: Paths Leading to Deadlocks in the Corresponding Petri Net.

```
Step 1:  Transform the Agent into the Corresponding
         Petri Net R_0 = (Pl_0, T_0, M_00);

Step 2:  Initialize H to empty;
         WHILE the net is not completely reduced DO

            Eliminate Redundant Places;

            WHILE there exist post-fusable transitions F_m and h_m DO
               Apply post-fusion rule;
            END;

            WHILE there exist pre-fusable transitions F_m and h_m DO
               Apply pre-fusion rule;
               H := h_m ∘ H;
            END;

         END

         /* H is now equal to h_i ∘ h_{i-1} ∘ ··· ∘ h_1, where i is the
         number of pre-fusions of transitions that have occurred. */

Step 3:  Obtain the reachability graph of the reduced Petri net;

Step 4:  For each sink node n of the graph, find a path σ_n which starts
         from the root of the graph and leads to n;

Step 5:  IF the pre-fusion rule has not been used THEN
            Output the set of paths found;
         ELSE
            FOR each path σ_n DO
               Follow the path in the initial Petri net R_0;
               Let M_n be the marking where the path leads to;
               FOR j = i TO 1 DO
                  IF h_j can fire in R_0 THEN
                     σ_n := σ_n ∘ h_j;
                     Update M_n;
                  END;
               END;
               Output σ_n;
            END;
```

In the following sections, we discuss the applicability and the characteristics of the algorithm.

4.5 An Example

As an example of the applicability of the above method we use the classical "Dining Philosophers" problem. In the literature, a well-known incorrect solution is often discussed that may lead to a deadlock situation. In this solution, each philosopher picks up his left fork first, then his right one, he eats and then puts the forks down in the same order that he picked them up. The deadlock occurs in the case where all philosophers have picked their left forks up and then wait indefinitely to get the right ones. The above solution can be described in CCS in the following way: A philosopher PH_i $i = 1, \ldots, 5$, can be described by the following process term:

$$\mu\mathtt{PH_i}\colon (\overline{\mathtt{i\_picks\_up\_fork\_i}}).(\overline{\mathtt{i\_picks\_up\_fork\_(i \oplus 1)}}).$$
$$(\overline{\mathtt{i\_puts\_down\_fork\_i}}).(\overline{\mathtt{i\_puts\_down\_fork\_(i \oplus 1)}}).\mathtt{PH_i}$$

where $\oplus$ denotes modulo 5 addition. On the other hand, each fork $\mathtt{F_i}$, $i = 1, \ldots, 5$, can be represented as:

$$\mu\mathtt{F_i}\colon (\mathtt{i\_picks\_up\_fork\_i}).(\mathtt{i\_puts\_down\_fork\_i}).\mathtt{F_i}+$$
$$((\mathtt{i} \ominus 1)\mathtt{\_picks\_up\_fork\_i}).((\mathtt{i} \ominus 1)\mathtt{\_puts\_down\_fork\_i}).\mathtt{F_i}$$

where $\ominus$ denotes modulo 5 subtraction. The whole system can be described by the parallel composition of the five philosophers and the five forks after properly restricting it. The Petri net corresponding to the above system, is shown in Figure 12. In this Figure, the darkened places are redundant places. Their removal reduces the Petri net to the one shown in Figure 13. It should be noted here that if the redundant places were not removed, there would not exist any post-fusable or pre-fusable transitions in the Petri net. The new Petri net has 10 pairs of post-fusable transitions. Consider for example the pair (t_2,t_3). After the post-fusion of the two transitions, a new transition is created, namely t_2t_3, which is post-fusable with t_4. The post-fusion creates transition $t_2t_3t_4$. Repeating the same procedure for all the pairs of post-fusable transitions, we get the Petri net of Figure 14. This net is not further reducible. Its reachability graph has 32 nodes compared to the 242 nodes of the reachability graph of the initial Petri net. Analysis of the reachability graph reveals a sink node which corresponds to the deadlock mentioned previously. As a last remark on this example, consider the sequence $\sigma = t_1 t_5 t_9 t_{13} t_{17}$ of transition firings that leads the reduced Petri net to deadlock. One can easily observe that the same sequence leads the initial Petri net to deadlock.

4.6 Comments on the Algorithm

This section discusses aspects concerning the proposed algorithm such as complexity, confluence as well as possible extensions.

4.6.1 Complexity

The basic principle behind the algorithm is that the Petri net representation of a CCS agent is usually very small compared to the size of the corresponding reachability graph. In this sense, it is worth to first perform reductions on the net (which may significantly prune the state space), instead of directly performing reachability analysis.

The algorithm terminates because each time one of the proposed transformations is applied, the size of the Petri net is reduced: each place elimination removes one place and each fusion of transitions reduces the number of transitions by one and also removes one place. As we have a finite initial Petri net, the reductions terminate in a finite number of steps.

As we mentioned before, static deadlock detection in concurrent systems is intractable in the general case [31]. Therefore, the worst case performance of every relevant algorithm is exponential while the average case performance is in general impossible to predict. A practical deadlock detection system should use a set of different algorithms, which possibly prune in different ways the set of initial states of the program. The following propositions examine the time complexity of verifying that the conditions of the adopted reductions hold.

Proposition 4.3 Let $R = (Pl, T, M_0)$ be a net. Given a place $p \in Pl$ with $pre(p) = \{h\}$ and $post(p) = F$, it is possible to verify the conditions of post-fusion in time $O(|F| + (|post(h)| - 1) * \Sigma_{f_i \in F}|post(f_i)|)$ in the worst-case.

Proof: It can be easily verified that each one of the conditions of Definition 4.3 except for the last one, can be performed in time $O(|F|)$, in the worst case. The last condition can be verified by checking that each place in the postset of h (except for p), does not belong to the postset of any of the $f_i \in F$. The result follows immediately from the above two remarks. ■

Proposition 4.4 Let $R = (Pl, T, M_0)$ be a net. Given a place $p \in Pl$ with $pre(p) = \{h\}$ and $post(p) = F$, it is possible to verify the conditions of pre-fusion in time $O(|F| + |pre(h)|)$ in the worst-case.

Proof: It can be easily verified that each one of the conditions of Definition 4.5 except for the last one, can be performed in time $O(|F|)$, in the worst case. The last condition can be verified by checking that each place in the preset of h has only one output. This can be done in time proportional to $|pre(h)|$. ■

Proposition 4.5 Let $R = (Pl, T, M_0)$ be a net. Given a place p of R, it is possible to verify the conditions of redundant place in time $O(max(|Pl| + |T|, e))$ in the worst-case, where e is the number of edges in the Petri net.

Proof: Detecting a redundant place consists in finding a place with only one input transition t_0 and only one output transition t_n, such that t_0 and t_n satisfy the properties of Definition 4.7. These properties can be validated by a depth first search algorithm that starts from t_0. During the search, all the places visited are examined to ensure that they have just one input and one output. The result follows from the fact that depth first search takes time $O(max(n, e))$ in a graph with n nodes and e edges [13]. ■

4.6.2 Effect of Reduction Order

In transformation-based systems, confluence is usually a desired property: it is convenient to know that two different transformation sequences will finally end up with the same irreducible result. In [7], a class of nets is exhibited, which is confluent under a specific set of reductions. However, in the same reference it is also shown that confluence does not hold in general for Petri net reductions.

In our case, there are additional reasons that can prevent confluence. First, as for practical reasons the reductions we adopt are simplified versions of the ones in [6], it is possible to get after two sequences of transformations two nets that are not further reducible by our algorithm, but which would be equivalent if the more general reductions of [6] were adopted. This is illustrated by the following example:

Example 4.2 Consider the following process term:

$$(\mathsf{a}.(\mathsf{b}.0|(\mathsf{c}.0+\mathsf{d}.0)))$$

As shown in Figures 15(a) and 15(b) respectively, there exist two different sequences of reductions that result in two different nets that are irreducible (with respect to the simplified transformations we have adopted).

There is an additional difficulty in reasoning about confluence in our case. The nets that the algorithm manipulates are not arbitrary. They represent the concurrent semantics of CCS programs, and it would be useful to have a characterization of their class when confluence is considered. However, such a characterization still remains an open problem [22].

As a final remark on reduction sequences, we should mention that the outer WHILE loop in Step 2 of the algorithm is necessary in order to ensure that the resulting net is irreducible. In general, the application of a reduction may alter the structure of the net in such a way that a previous reduction becomes applicable again. This can be easily verified for the net of the process

$$\mathsf{a}.(\mathsf{b}.\mathsf{c}.0|\mathsf{d}.\overline{\mathsf{c}}.0|\overline{\mathsf{b}}.0)\backslash\{\mathsf{c},\mathsf{b}\}$$

which is shown in Figure 16. After eliminating place `p` and performing two post-fusions, place `q` becomes redundant.

4.6.3 General Remarks

The algorithm can be a extended in a number of ways and various levels of sophistication can be added. In the following, we discuss two such enchancements.

First, the deadlock model adopted is rather general. In fact, it does not discriminate between correct termination and deadlock situations. For example in Figure 6, the marking $\{p_5, p_6\}$ can be considered as a terminating state for the corresponding CCS agent if one assumes that the inactive agent indicates a termination point in the program. In this case, the algorithm can be modified as follows: every time a sink node is detected in the reachability graph, the corresponding marking is examined. If *all* the places in the marking are sequential components such as `0` or `id`|`0` and so on, (i.e. combinations of the inactive agent with the labels |`id` and `id`|), then the marking corresponds to termination. Otherwise, it corresponds to a deadlock.

The second extension can be again understood using Figure 6. The two components of the Petri net are independent from each other, and can therefore be analyzed separately. This reduces even more the number of states that have to be examined. However, it requires a preprocessing algorithm for detecting connected components of the net (see for example [13]).

5 Concluding Remarks

5.1 Statistical Results

We have developed a prototype system for the algorithm, using Prolog. The development required approximately 4 person months, and the program consists of approximately 140 predicates [26]. In the following, we illustrate the performance of the system using two well-known examples. The first one is a *scheduler* used to coordinate a set of n processes [22].

Figure 17(a) shows the corresponding Petri net for two processes. The following table presents the performance of our prototype compared to the performance of ordinary reachability analysis (RA), i.e., examination of the whole state space of the program. The time given for our system, includes the time to convert the program into a Petri net, the time to reduce the Petri net and the time to perform reachability analysis on the reduced Petri net.

	Ordinary RA		Reduction	
n	*States*	*Time (sec)*	*States*	*Time (sec)*
2	8	0.02	2	0.15
6	384	142	6	1.70
10	10240	–	10	6.58
30	$> 30 \times 10^9$	–	30	140
	$n \times 2^n$		n	

As a second example, we consider the problem of n processes that use a binary semaphore to ensure *mutual exclusion*. Figure 17(b) shows the corresponding Petri net for the case of two processes. The following table gives the results obtained for different values of n:

	Ordinary RA		Reduction	
n	*States*	*Time (sec)*	*States*	*Time (sec)*
3	32	0.6	4	1
5	192	25.5	6	3.3
10	11264	–	11	21.4
15	524288	–	16	63
	$2^n \times (n+1)$		$n+1$	

The results show that ordinary reachability analysis becomes infeasible even for small values of n. On the other hand, analysis using our approach gives very good results and does not degrade as the size of the examples increases. However, we should emphasize that this is not always the case. Consider for example the “Dining Philosophers” problem. Although a reduction in the number of states is obtained, the resulting net still has an exponential number of states in terms of the number of Philosophers involved.

5.2 Future Work

There are many interesting aspects of our work that should be further investigated. The most important of them are the following:

1. The reduction techniques we have used, are a subset of those that have been proposed in Petri net theory. Can we define other reductions that are efficiently implementable and which at the same time preserve the deadlock information of the initial Petri net?

2. Can we define similar deadlock preserving transformations *directly* on CCS agents, without transforming them into Petri nets? Our attempts in this area have not been fruitful until now. This fact possibly suggests that it is much easier to perform such transformations on the Petri nets (possibly due to their graphical nature).

3. Clearly, it would be interesting to apply the proposed technique on other concurrent programming languages (Ada for example). As we mentioned in the introduction, existing techniques operate on Petri nets whose size may be considerable. In our opinion, this is due to the fact that the transformation from programs to Petri nets is not optimal. What we propose is to model the source programs using Process Algebras (see for example [1]) and then translate the resulting Process Algebra program into a Petri net. We believe that such a procedure would certainly result into more manageable nets.

Answers to the above questions will help us define more efficient methods for the analysis of concurrent programs. Future research includes incorporating additional deadlock preserving reductions in the system and investigating the possibility of incremental deadlock detection.

Acknowledgments

We would like to thank the anonymous referees for many insightful comments and suggestions which have significantly improved the quality of the paper. We would also like to thank E. R. Olderog for useful discussions and remarks. This research was supported by the Natural Sciences and Engineering Research Council of Canada and a University of Victoria Graduate Fellowship.

References

[1] E. Astesiano, G. Serrano and E. Zucca, The draft formal definition of Ada, Consorzio per la Ricersa e le Applicazioni di Informatica, (1986).

[2] Automatic Verification Methods for Finite State Systems, J. Sifakis Ed., International Workshop 1989, Lecture Notes in Computer Science (Springer-Verlag, Berlin, New York, 1990).

[3] W. Brauer, R. Gold and W. Vogler, A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets, in: Advances in Petri Nets 1990, Lecture Notes in Computer Science (Springer-Verlag, Berlin, 1991) 1-46

[4] G. Berthelot, G. Roucairol and R. Valk, Reduction of nets and parallel programs, in: W. Brauer, Ed., Net Theory and Applications, Lecture Notes in Computer Science (Springer-Verlag, New York, 1980) 277-290.

[5] G. Berthelot, Checking properties of nets using transformations, in: G. Rozenberg, Ed., Advances in Petri Nets 1985, Lecture Notes in Computer Science (Springer-Verlag, New York, 1986) 19-40.

[6] G. Berthelot, Transformations and Decompositions of Nets, in: W. Brauer, W. Reisig and G. Rozenberg, Ed., Petri Nets: Central Models and their Properties, Lecture Notes in Computer Science, (Springer-Verlag, New York, 1987) 359-376.

[7] G. Berthelot, Transformations et analyse de reseaux de Petri - Application aux Protocoles, Thèse d'Etat, Université P. et M. Curie (1983).

[8] T.A. Chu, A method of abstraction for Petri nets, Proceedings of International Workshop on Petri Nets and Performance Models (Madison, 1987) 164-173.

[9] P. Degano, R. DeNicola and U. Montanari, CCS is an (augmented) contact-free C/E system, in: M. Venturini Zilli, Ed., Mathematical Models for the semantics of Parallelism, Lecture Notes in Computer Science (Springer-Verlag, New York, 1987) 144-165.

[10] P. Degano, R. DeNicola and U. Montanari, A Distributed operational semantics for CCS based on condition/event systems, Acta Informatica 26 (1988) 59-91.

[11] P. Degano, R. De Nicola and U. Montanari, Partial Orderings, Descriptions and Observations of Nondeterministic Concurrent Processes, in: J. W. de Bakker, W. P. de Roever, G. Rozenberg, Ed., Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Lecture Notes in Computer Science (Springer-Verlag, Berlin Heidelberg, 1989) 438-466.

[12] C.A.R. Hoare, Communicating Sequential Processes (Prentice-Hall, 1985).

[13] A. Gibbons, Algorithmic Graph Theory (Cambridge University Press, 1989)

[14] P. Godefroid, Using partial orders to improve automatic verification methods, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 3 (1991) 321-339.

[15] R. Janicki and M. Koutny, Net implementation of Optimal Simulations, in Proceedings of the 11th International Conference on Application and Theory of Petri Nets (Paris, 1990) 295-314.

[16] R. Johnsonbaugh and T. Murata, Additional methods for reduction and expansion of marked graphs, IEEE Trans. Circuit Syst., vol. CAS-28, no. 10, (1981) 1009-1014.

[17] R. Milner, A calculus of Communicating Systems, Lecture Notes in Computer Science (Springer-Verlag, New York,1980).

[18] R. Milner, Communication and Concurrency (Prentice-Hall, 1989).

[19] T. Murata and J. Y. Koh, Reduction and expansion of live and safe marked graphs, IEEE Trans. Circuits Syst., vol. CAS-27, no. 1, (1980) 68-70.

[20] T. Murata, B. Shenker and S. M. Shatz, Detection of ADA static deadlocks using Petri net invariants, IEEE Trans. Software Eng., vol. 15, no. 3, (1989) 314-326.

[21] E.R. Olderog, Operational Petri net semantics for CCSP, in: G. Rozenberg Ed., Advances in Petri Nets 1987, Lecture Notes in Computer Science (Springer-Verlag, New York, 1987) 196-223.

[22] E. R. Olderog, Nets, Terms and Formulas: Three Views of Concurrent Processes and their relationship, Habilitationsschrift, Univ. Kiel, (1988/89). (Also published by Cambridge University Press, 1991).

[23] C.A. Petri, Kommunikation mit Automaten, Bonn: Institut fur Instrumentelle Mathematik, Schriften des IIM Nr. 3, 1962.

[24] W. Reisig, Petri Nets, EATCS Monographs on Theoretical Computer Science, vol. 4 (Springer-Verlag, New York, 1985).

[25] P. Rondogiannis, Detecting deadlocks in CCS agents using Petri net reduction techniques, Master's thesis, Department of Computer Science, University of Victoria, Canada (1991).

[26] P. Rondogiannis and M.H.M. Cheng, A Prolog System for Detecting Deadlocks in Concurrent Programs, in: Proceedings of the International Conference on the Practical Applications of Prolog, London, (1992).

[27] S. M. Shatz and W. K. Cheng, A Petri net framework for automatic static analysis of Ada tasking behavior, J. Syst. Software, vol. 8 (1988) 343-359.

[28] I. Suzuki and T. Murata, A method for stepwise refinements and abstractions of Petri Nets, J. Comput. Syst. Sci., vol. 27, no. 1, (1983) 51-76.

[29] D. Taubner, Finite Representations of CCS and TCSP Programs by automata and Petri Nets, Lecture Notes in Computer Science, (Springer-Verlag, New York, 1989).

[30] R. N. Taylor, A general-purpose algorithm for analyzing concurrent programs, Communications of the ACM, 26(5) (1983) 362-376.

[31] R. N. Taylor, Complexity of Analyzing the Synchronization Structure of Concurrent Programs, Acta Informatica, 19 (1983) 57-84.

[32] S. Tu, S.M. Shatz, and T. Murata, Applying Petri net reduction to support Ada-tasking deadlock detection, in The 10th International Conference on Distributed Computing Systems, (1990).

[33] S.M. Shatz, S. Tu and T. Murata, Theory and Application of Petri Net Reduction for Ada-Tasking Deadlock Detection, Technical Report UIC-EECS 91-15, Dept. of EECS, University of Illinois at Chicago, (1991).

[34] R. Valette, Analysis of Petri nets by stepwise refinements, J. Comp. Syst. Sciences, 18 (1979) 35-46.

[35] A. Valmari, Stubborn sets for reduced state space generation, in: Proc. 10th International Conference on Application and Theory of Petri nets, vol. 2 (Bonn, 1989) 1-22.

[36] W. Vogler, Behaviour preserving refinements of Petri Nets, in: Lecture Notes in Computer Science 246 (1987) 82-93.

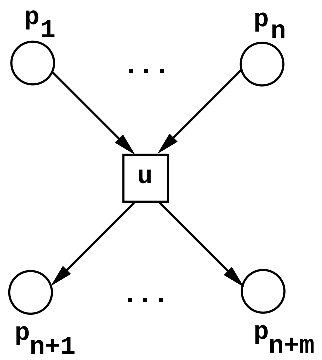


Figure 1: Graphical Representation of a Transition

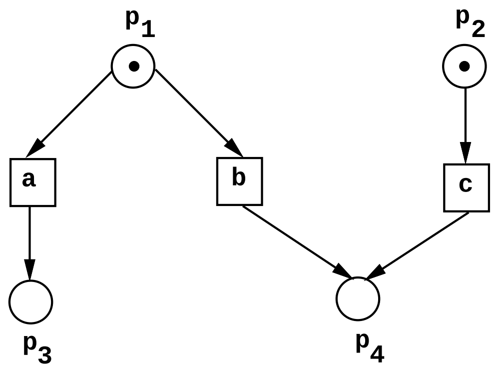


Figure 2: The Petri net of Example 2.2

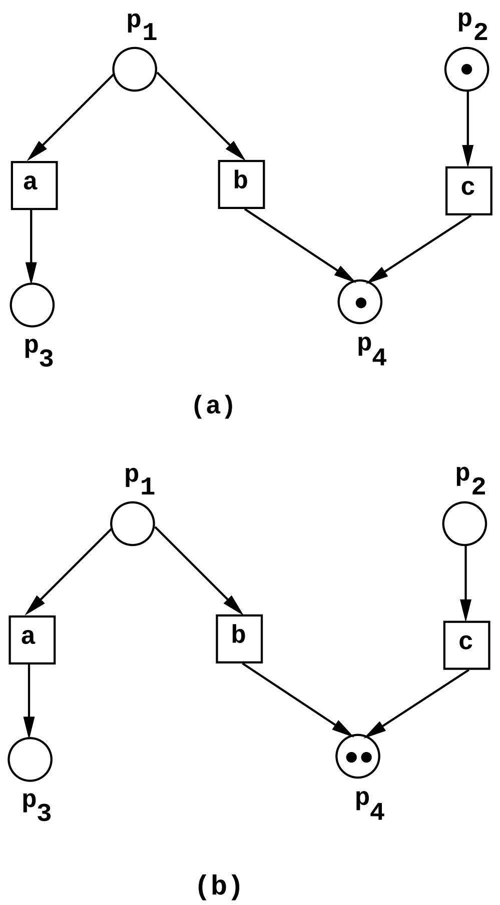


Figure 3: Executions of Transitions

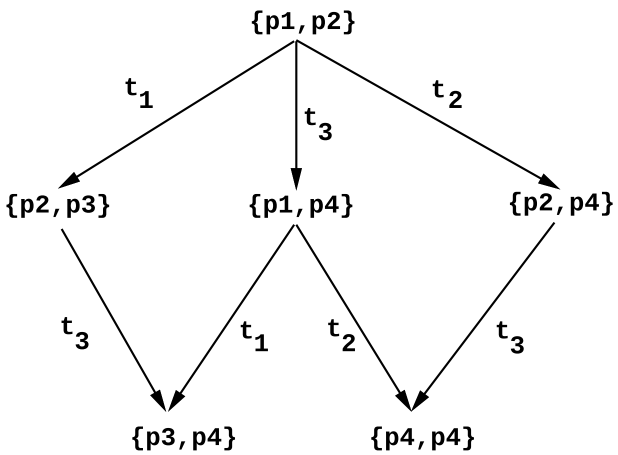


Figure 4: Reachability Graph

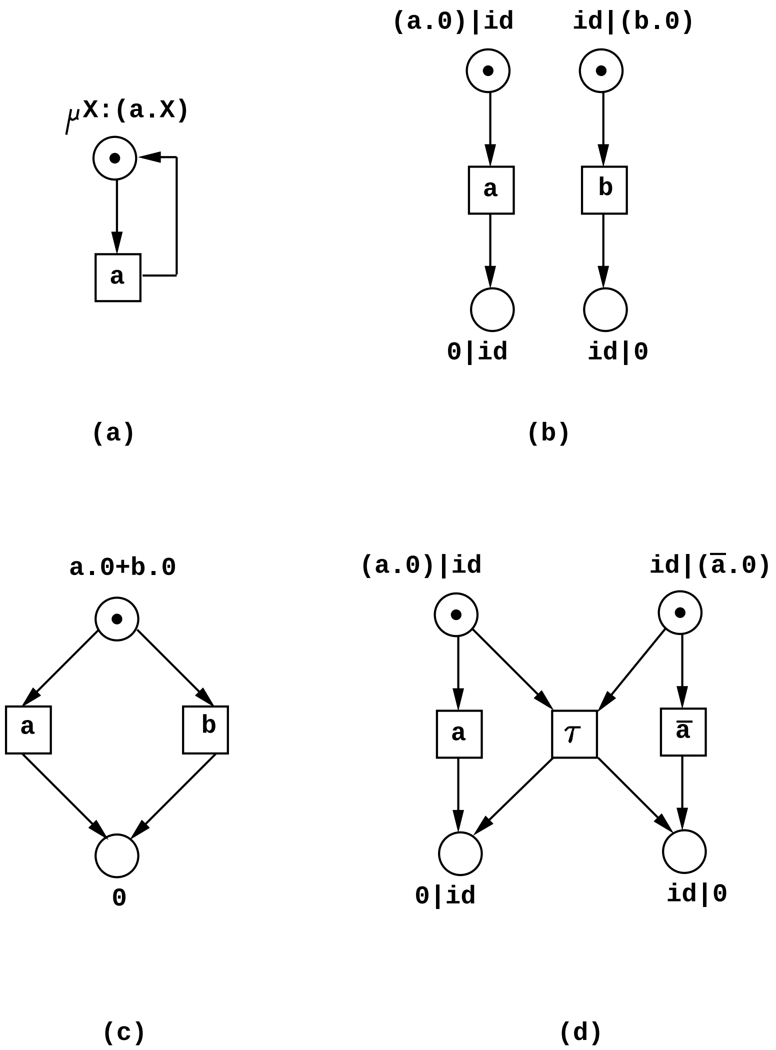


Figure 5: Examples of Concurrent Semantics

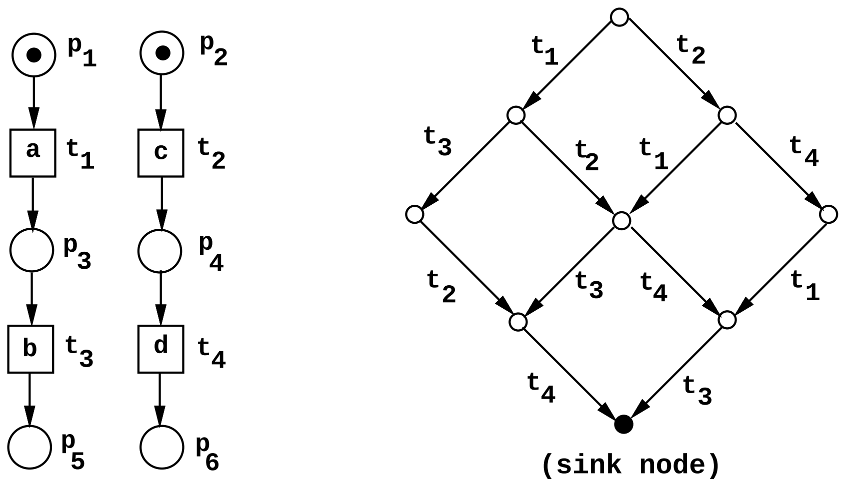


Figure 6: Petri Net of $a.b.\mathbf{0}|c.d.\mathbf{0}$ and its Reachability Graph.

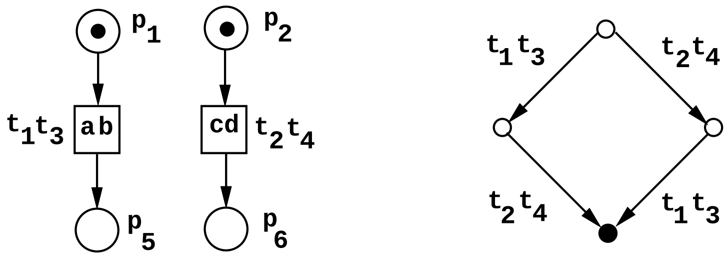


Figure 7: Reduced Petri Net of $a.b.\mathbf{0}|c.d.\mathbf{0}$ and its Reachability Graph.

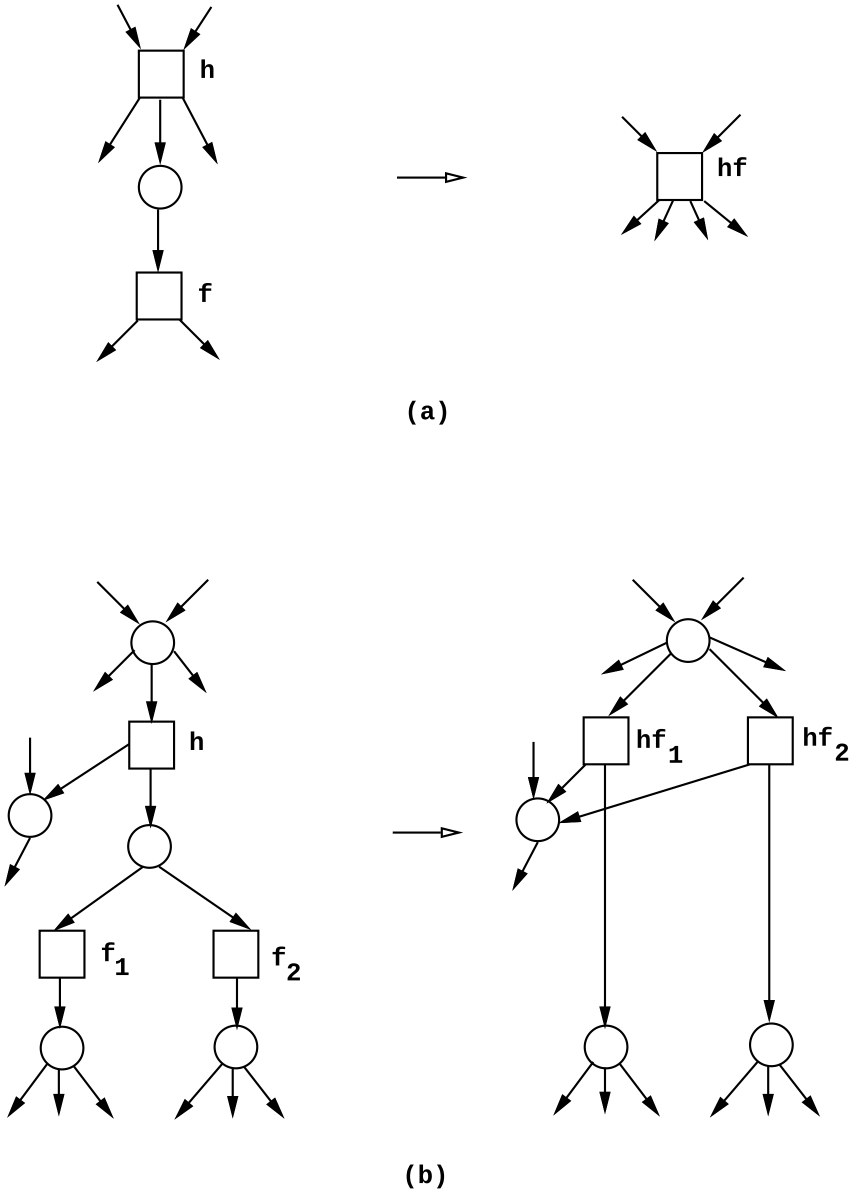


Figure 8: Examples of Post-fusion of Transitions

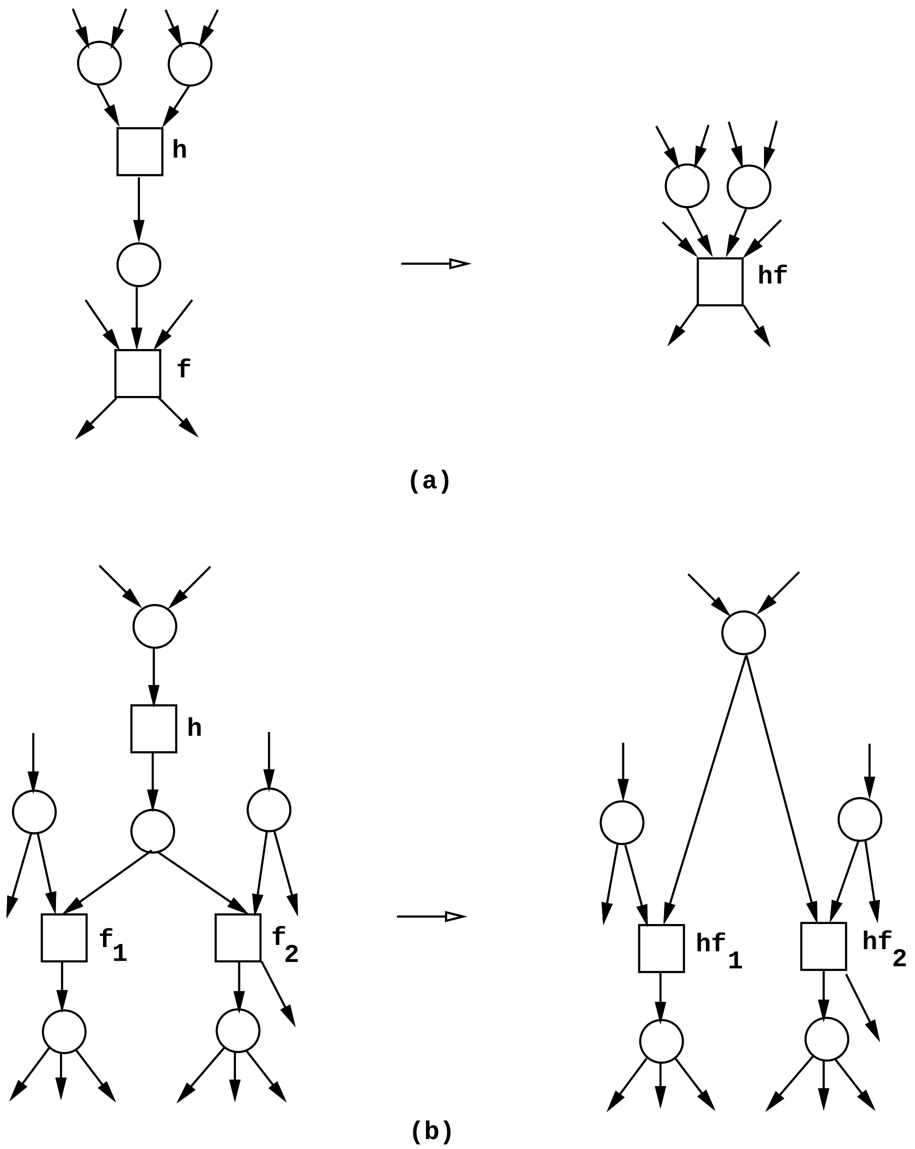
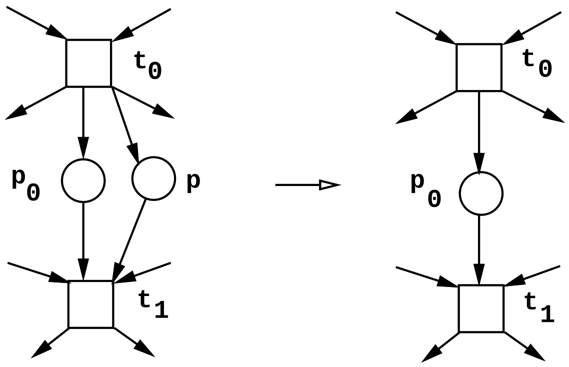
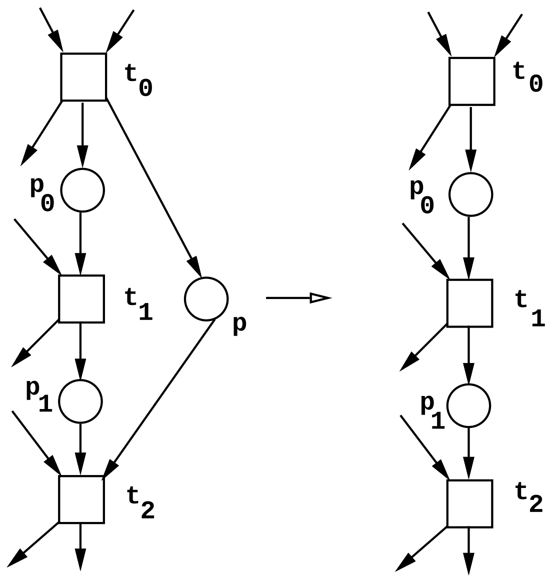


Figure 9: Examples of Pre-fusion



(a)



(b)

Figure 10: Redundant Places

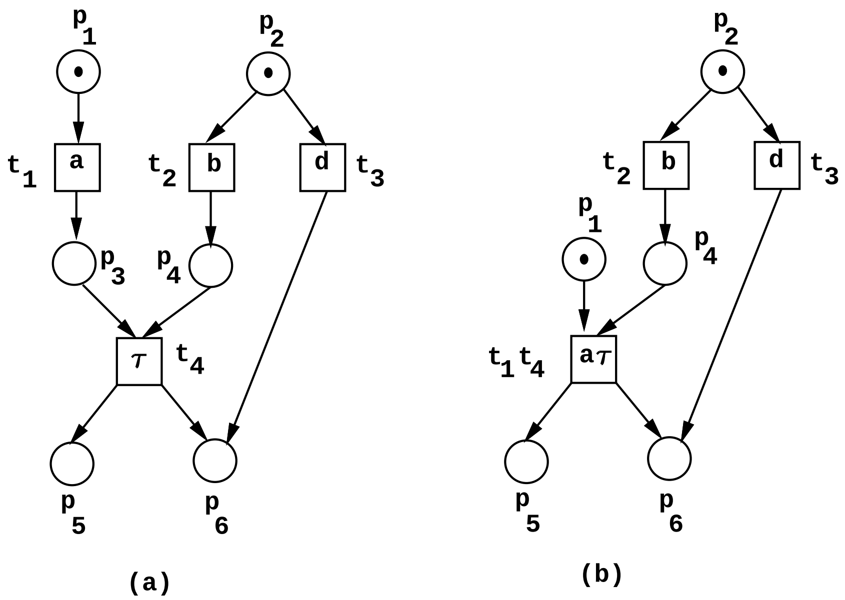


Figure 11: Illustration of Theorem 4.3

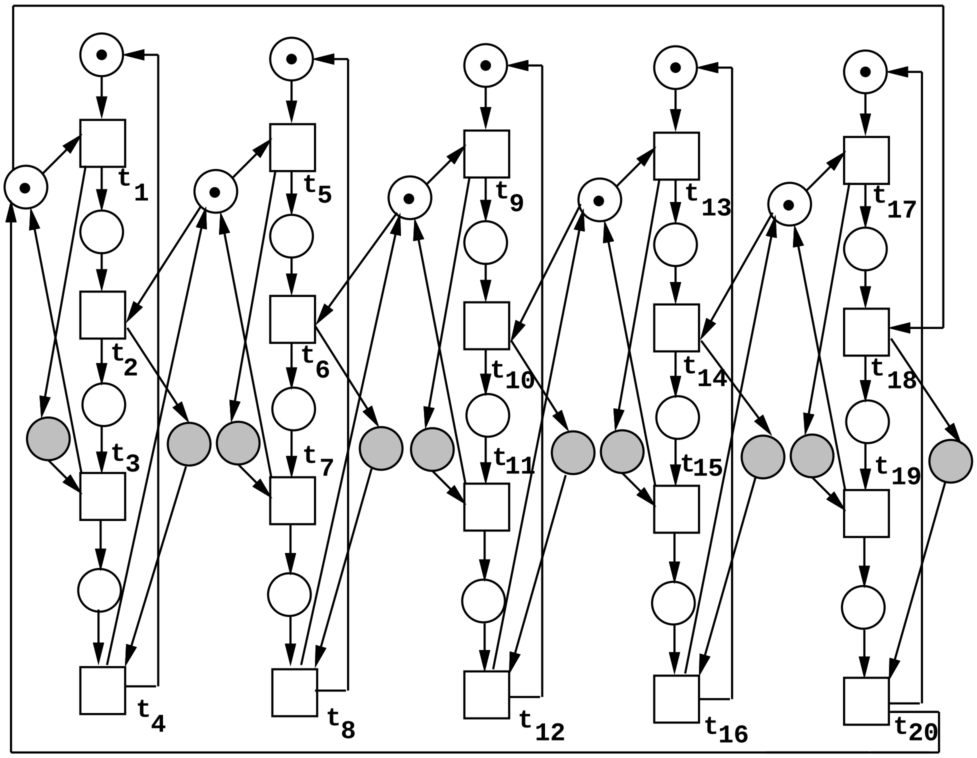


Figure 12: The Dining Philosophers

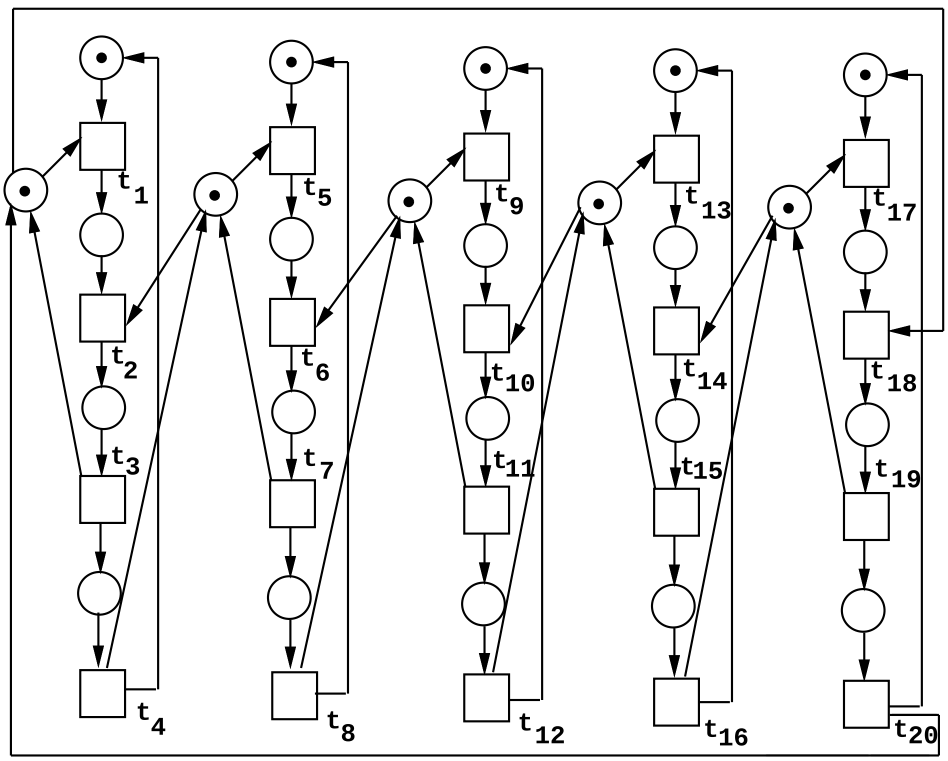


Figure 13: The Net after Elimination of the Redundant Places

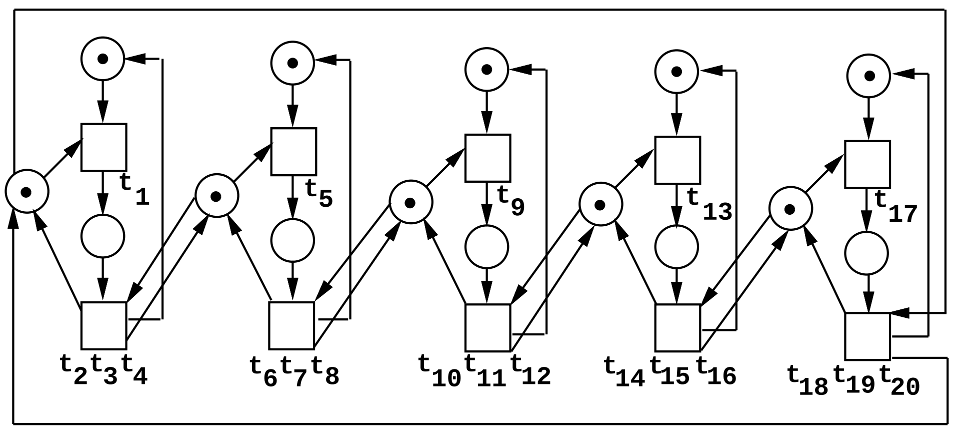


Figure 14: The Net after Post-fusion of Transitions

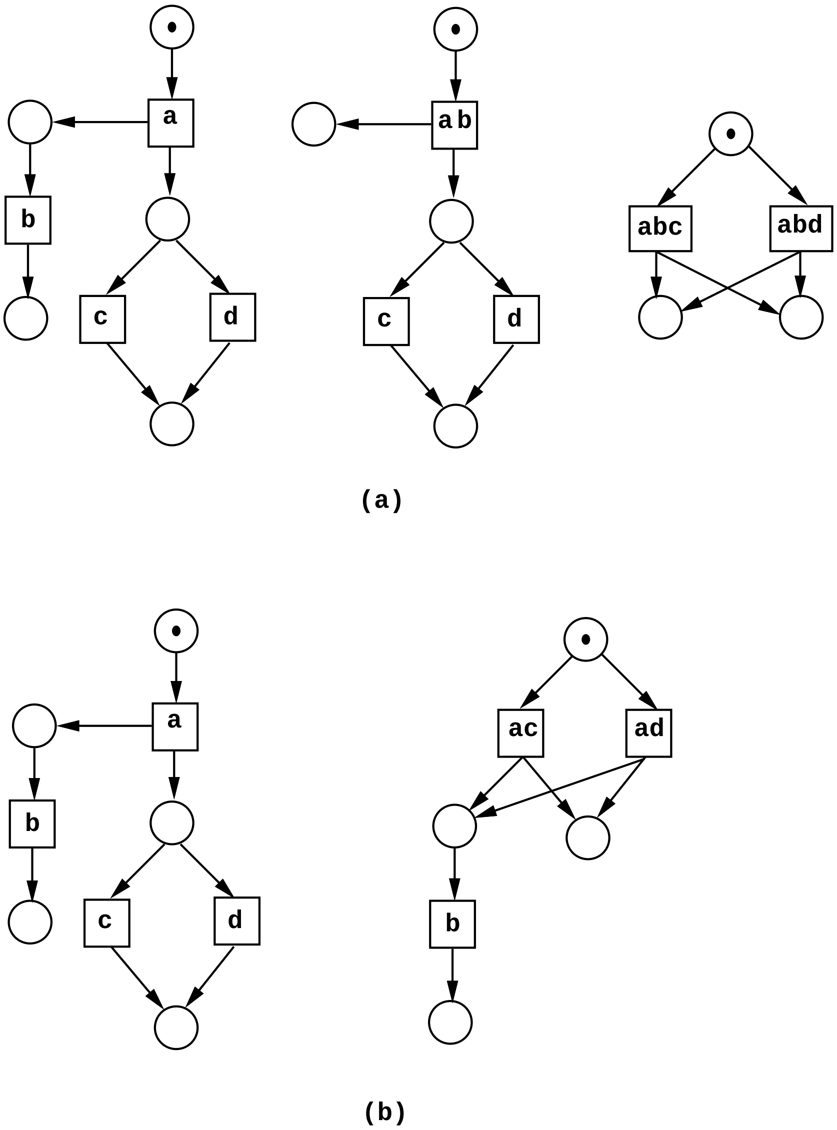


Figure 15: Two Different Reduction Sequences

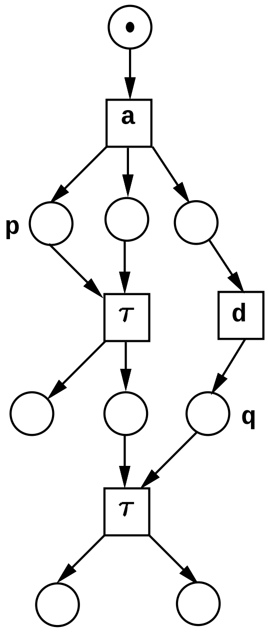
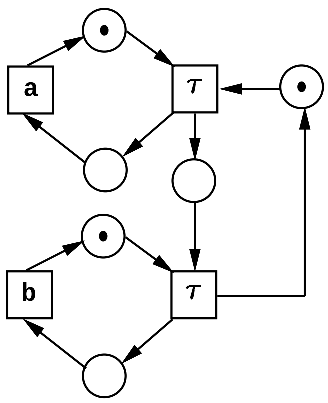
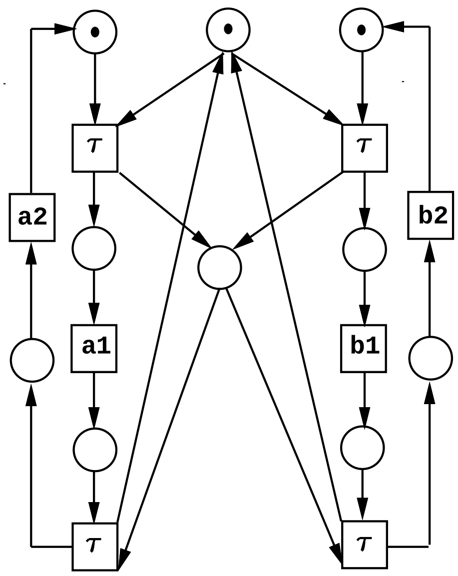


Figure 16: Creation of Redundant Place after Post-Fusions



(a)



(b)

Figure 17: (a) Scheduler (b) Mutual Exclusion