DETECTING DEADLOCKS IN CCS AGENTS USING PETRI NET REDUCTION TECHNIQUES

by

Panagiotis Rondogiannis
Ptihion in Computer Engineering and Informatics
University of Patras, 1989

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. M. H. M. Cheng, Supervisor (Dept. of Computer Science)

[redacted]
Dr. M. Levy, Departmental Member (Dept. of Computer Science)

[redacted]
Dr. N. Dimopoulos, Outside Member (Dept. of Elect. & Comp. Eng.)

[redacted]
Dr. K. F. Li, External Examiner (Dept. of Elect. & Comp. Eng.)

Table of Contents

List of Figures

Acknowledgment

I would like to thank my supervisor, Dr. M. H. M. Cheng of the Department of Computer Science, University of Victoria, for his encouragement, patience, and advice during the course of this research and for his help in the preparation of this manuscript. I would also like to thank Dr. G. Pavlides of the Department of Computer Engineering and Informatics, Univerity of Patras, Greece, who first gave me the chance to get involved with research in Computer Science. I also want to thank Evaggelia Kontopidi for numerous discussions and useful suggestions during the preparation of this thesis. Finally, I would like to express my gratitude to my parents for their moral support and encouragement throughout my studies.

Στους γονείς μου

Chapter 1

Introduction

Concurrency is one of the major challenges facing computer science, both in theory and in practice. The wide variation in structure and architecture of concurrent machines has given rise to an equally wide variation of programming methods and languages. Although this situation has often caused confusion, progress in theoretical computer science has brought significant understanding and has provided simple and elegant solutions. In this thesis, we consider two different formal approaches to concurrency, namely Process Algebras and Petri nets.

Process Algebras use algebraic expressions for the specification of concurrent systems. They rely on a small set of basic operators which correspond to primitive notions of concurrent systems. The operators are used to build systems from more elementary ones. The main contributions in this area are Milner's Calculus of Communicating Systems (CCS) [1],[2], Hoare's Communicating Sequential Processes (CSP) [3], [4] and Algebra of Communicating Processes (ACP) by Bergstra and Klop [5].

On the other hand, Petri nets are a graphical and mathematical tool applicable to many systems. As a graphical tool, Petri nets can be used as a

visual communication aid similar for example to flow charts. In addition, Petri nets are equipped with *tokens* which are used to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it can be used to set up models governing the behavior of systems. Historically speaking, the concept of Petri nets has its origin in Carl Adam Petri's dissertation [8]. Since then, Petri nets have been used extensively in areas such as operating systems, performance evaluation, communication protocols, concurrent programs and in numerous other applications ([15]-[20]). Comprehensive introductions to Petri net theory can be found in [6],[7],[9], and the most recent results in the field are published annually in *Advances in Petri Nets* ([10]-[14]).

In recent years, Process Algebras and Petri nets began to influence each other and to converge. More specifically, recent research has been conducted in finding for programs of Process Algebras, finite representations in terms of Petri nets ([21]-[26]). 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. Moreover, the representation by a well-known machine model supplies an aid for the implementation of Process Algebras.

In this thesis we suggest that such a representation implies other significant benefits as well. Petri nets have been studied thoroughly for years, and many sophisticated techniques exist that facilitate their analysis. Therefore, one can apply Petri net analysis methods to reason about Process Algebra programs. In this thesis, we address the problem of *deadlock detection* in a Process Algebra program, by developing a deadlock detection algorithm for the corresponding Petri net. To illustrate the practicality of the proposed

approach, we implemented a Prolog system that performs the deadlock detection algorithm. Our overall approach serves the following broader goals:

1. It relates two well known models of concurrent computation, namely Process Algebras and Petri nets, and suggests that analysis methods for the latter can be used to reason about the former. In this sense, this thesis presents the first *practical* results in a mainly theoretical area of research.

2. It provides an efficient algorithm for deadlock detection in Process Algebra programs. As far as we know, not many interesting results exist in this area.

3. It establishes a theoretical framework that can be used as a basis for developing deadlock detection algorithms for existing concurrent programming languages.

4. It suggests that Prolog is a very convenient means for describing the kind of applications that are discussed in this thesis.

The rest of this introduction gives a chapterwise summary of the thesis contents. Chapter 2 introduces the specific Process Algebra that will be used throughout this thesis, namely CCS. CCS is introduced by an example and then it is formally defined and its transitional semantics is given. Chapter 3 gives an introduction to the theory of Petri nets. Chapter 4 deals with defining concurrent semantics for CCS with the use of Petri nets. In other words, a procedure is described that transforms a CCS program into a Petri net. In Chapter 5, we suggest that the Petri net representation of a CCS program can be used to detect deadlocks in the program. Petri net reduction techniques are used to reduce the size of the net, making it in this way

easier to analyze. We prove that the reductions have the important property of preserving the deadlock information of the initial Petri net. The proposed algorithm has the important characteristic that it avoids exploring the whole state space of the program. Chapter 6 describes the Prolog system that implements the proposed algorithm. The system can be used to detect deadlocks in CCS programs but also to correct programs that contain hidden deadlocks. Chapter 7 concludes the thesis. It summarizes the main contributions of our work and presents the most interesting questions that require further investigation.

Chapter 2

CCS: Definitions and Transitional Semantics

2.1 Introduction to CCS Notation

Complex systems consist of a number of components that act and interact with each other. These components are called *agents*. The behavior of an agent consists of two kinds of actions: *communications* with other agents and actions that occur independently or *concurrently* with the actions of the other agents. Consider for example a system consisting of two agents, a vending machine and a customer [3]. When the customer inserts a coin in the machine, a communication between these two agents takes place. A communication also takes place when the machine delivers a product. On the other hand, an internal action of the machine (e.g. a noise), can not be considered as a communication with the customer, although it may be a communication between two parts of the machine. In the following, this simple example is used to introduce the basic notation of CCS.

Let C denote a customer agent who repeatedly inserts coins and accepts products from the vending machine M. Then the customer's behavior can

be described in an algebraic way as follows:

$$C \stackrel{\text{def}}{=} \overline{\texttt{coin}}.C'$$

$$C' \stackrel{\text{def}}{=} \texttt{product}.C$$

or simply:

$$C \stackrel{\text{def}}{=} \overline{\texttt{coin}}.\texttt{product}.C$$

In the above notation, labels are used (i.e., $\overline{\texttt{coin}}, \texttt{product}$) to represent actions of the agent. The overbar has the meaning that $\overline{\texttt{coin}}$ is an action *delivered* by the agent. On the other hand, product represents an action *accepted* by the agent. The dot '.' is the first combinator of the calculus, called *prefix*. In general, a.P denotes an agent whose behavior is to perform action a and then proceed according to the definition of P. Clearly, the customer agent has been described using a recursive definition, which expresses the intuitive fact that the behavior of the customer is an infinite sequence of actions

$$\overline{\texttt{coin}}, \texttt{product}, \overline{\texttt{coin}}, \texttt{product}, \ldots$$

The recursive definition of the customer can be expressed more formally as:

$$C \stackrel{\text{def}}{=} \mu X : (\overline{\texttt{coin}}.\texttt{product}.X)$$

which may be pronounced "the customer C is defined as the agent X such that $X = \overline{\texttt{coin}}.\texttt{product}.X$". In the above so-called μ−expression, X is a variable and can be renamed without any semantic effect. Using these conventions, the corresponding agent for the vending machine is:

$$M \stackrel{\text{def}}{=} \mu X : (\texttt{coin}.\overline{\texttt{product}}.X)$$

Actions like coin, $\overline{\texttt{coin}}$ and product, $\overline{\texttt{product}}$ are called *complementary* actions.

A modification of the above example allows the introduction of the second combinator of the calculus. A sophisticated vending machine SM offers an alternative to its customers: it accepts two kinds of coins namely $1p$ and $2p$ and delivers cheap and expensive products respectively. Its behavior can be described by:

$$SM \stackrel{\text{def}}{=} \mu X : (\texttt{1p}.\overline{\texttt{cheap}}.X + \texttt{2p}.\overline{\texttt{expensive}}.X)$$

The new combinator '+' is called *summation.* In general, an agent $P + Q$ behaves either like P or like Q. As soon as one performs its first action, the other one is discarded. Often the environment will only permit one of these alternatives. But if both alternatives are permitted, then $P + Q$ is non-deterministic; that is, it may behave like P on one occasion and like Q on another.

The system S, composed of the customer agent C and the vending machine M working in parallel, can be represented as

$$S \stackrel{\text{def}}{=} (\mu X : (\overline{\texttt{coin}}.\texttt{product}.X)) \mid (\mu Y : (\texttt{coin}.\overline{\texttt{product}}.Y))$$

using the binary combinator '|' (*parallel composition*). Intuitively, the agent $P \mid Q$ is a system in which P and Q may proceed independently and concurrently but may also interact through complementary actions (e.g. coin, $\overline{\texttt{coin}}$). In the above example, the pair $(\overline{\texttt{coin}}, \texttt{coin})$ represents the handshake that takes place when the customer inserts a coin in the vending machine. Similarly, $(\texttt{product}, \overline{\texttt{product}})$ represents the handshake that takes place when the machine delivers a product to the customer.

Clearly, further customers could be added to the system above, to share the use of the vending machine. But there are cases that it is necessary to internalize an action so that no further agents may be connected to it. This means that this action becomes unavailable for external communication. This suggests the use of the *restriction* operator. Restriction takes a parameter which is a set L of actions and is a unary operator on agents which we write as $\backslash L$ in postfixed position. In the specific example, if the vending machine was designed to interact with a specific customer and none else, the actions $\mathtt{coin}$, $\overline{\mathtt{coin}}$, $\mathtt{product}$ and $\overline{\mathtt{product}}$ should be internalized. This is written as:

$$S\backslash\{\mathtt{coin},\mathtt{product}\}$$

This is a new agent which can no more interact with its environment through actions $\mathtt{coin}$, $\overline{\mathtt{coin}}$, $\mathtt{product}$, $\overline{\mathtt{product}}$. The restriction operator $\backslash L$ internalizes both the actions in L and their complements.

Let $\mathtt{l}$ be an arbitrary action and $\overline{\mathtt{l}}$ its complement. A function f from actions to actions is a *relabelling function* if it respects complements; that is, whenever $f(\mathtt{l}) = \mathtt{l}'$ then $f(\overline{\mathtt{l}}) = \overline{\mathtt{l}'}$. For each relabelling function f, the relabelling combinator $[f]$ postfixed to an agent, has the effect of relabelling the actions of the agent as dictated by f. The notation $\mathtt{l}'_1/\mathtt{l}_1,\ldots,\mathtt{l}'_n/\mathtt{l}_n$ will be used for the relabelling function f for which $f(\mathtt{l}_i) = \mathtt{l}'_i$ and $f(\overline{\mathtt{l}_i}) = \overline{\mathtt{l}'_i}$, for $i = 1,\ldots,n$ and otherwise $f(\mathtt{l}) = \mathtt{l}$. The relabelling combinator serves to allow an agent to be represented as instance of another agent. In this way, one can design for example a vending machine B that accepts only $\mathtt{2p}$ and delivers only $\mathtt{expensive}$ products, as a special instance of the vending machine M:

$$B \stackrel{\text{def}}{=} M[\mathtt{2p}/\mathtt{coin},\mathtt{expensive}/\mathtt{product}]$$

The above completes the informal introduction to the CCS notation.

2.2 The Basic Language

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 $\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots$ and their complements $\overline{\mathtt{a}}, \overline{\mathtt{b}}, \overline{\mathtt{c}}, \ldots$ 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 $(\mathtt{b}, \overline{\mathtt{b}})$ of complementary actions by the components of the agent. It is sufficient to introduce this single *perfect* action τ, to represent all such handshakes. 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 *Relabelling* (f a relabelling function)
7. $\mu X : E$, a *Recursion* ($X \in \mathcal{X}$)

In the above definition, **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. Formally:

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

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

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.

Finally, 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.*

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.

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

2.3 Transitional Semantics

The basic idea for providing transitional semantics for CCS is that one may think of expressions like C and C' in the initial definition of the customer agent as standing for the different possible states of an agent. Rather than distinguishing between the concepts of agent and state, it is convenient to identify them, so that both agent and state will always be understood to mean an agent in some state. Since a transition from state to state is accomplished by an action, it is reasonable to write $P \xrightarrow{\alpha} Q$ to denote a transition from a state P to a state Q via an action α. For example, corresponding to the definition of the customer C, we have the following transitions:

$$C \xrightarrow{\overline{\texttt{coin}}} C'$$

$$C' \xrightarrow{\texttt{product}} C$$

and indeed these transitions define the behavior of a customer just as precisely as its equational definition.

Formally, in giving meaning to the basic language, the general notion of a *labelled transition system*

$$(S, T, \{\xrightarrow{t} : t \in T\})$$

is used. The transition system consists of a set S of *states*, a set T of *transition labels*, and a *transition relation* $\xrightarrow{t} \subseteq S \times S$ for each $t \in T$. In the specific transition system, S will be the set of agents, and T will be Act, the set of actions. The semantics for the set of agents is given by the definition of each transition relation $\xrightarrow{\alpha}$ over the set. Transitions are derived using rules of the

form:

$$T_1, \ldots, T_m \quad \textbf{implies} \quad T \quad \textbf{where} \quad \textit{condition}$$

with $T_1, \ldots, T_m, T$ denoting transitions. What the above rule states is that if $T_1, \ldots, T_m$ are transitions satisfying the *condition* then T is also a transition. The complete set of rules is given below. The names **Act**, **Sum**, **Com**, **Res**, **Rel** and **Rec** indicate that the rules are associated respectively with Prefix, Summation, Parallel Composition, Restriction, Relabelling and Recursion. In the following, $\mathtt{l}$ ranges over $\mathcal{L}$ and α over Act.

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

(Sum$_1$) $E_1 \xrightarrow{\alpha} E_1'$ **implies** $E_1 + E_2 \xrightarrow{\alpha} E_1'$

(Sum$_2$) $E_2 \xrightarrow{\alpha} E_2'$ **implies** $E_1 + E_2 \xrightarrow{\alpha} E_2'$

(Com$_1$) $E_1 \xrightarrow{\alpha} E_1'$ **implies** $E_1|E_2 \xrightarrow{\alpha} E_1'|E_2$

(Com$_2$) $E_2 \xrightarrow{\alpha} E_2'$ **implies** $E_1|E_2 \xrightarrow{\alpha} E_1|E_2'$

(Com$_3$) $E_1 \xrightarrow{\mathtt{l}} E_1'$ **and** $E_2 \xrightarrow{\overline{\mathtt{l}}} E_2'$ **implies** $E_1|E_2 \xrightarrow{\tau} E_1'|E_2'$

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

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

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

The rule for Prefix states that $\alpha.E$ goes to state E after action α has occurred. The two rules for Summation can be read as follows: if any one summand of the sum $E_1 + E_2$ has an action, then the whole sum also has that action. The intuition behind the first Parallel Composition rule **Com$_1$**

is that if E_1 can do an action alone then it can also do it in the context $E_1|E_2$, leaving E_2 undisturbed. Similar explanation applies to $\mathbf{Com_2}$. The third Parallel Composition rule $\mathbf{Com_3}$ implies that if E_1 and E_2 can perform complementary actions, then a handshake can take place. This handshake is denoted by τ, the perfect action. The rule for Restriction implies that if E can perform an action α then $E\backslash L$ can perform the same action α only if α does not belong to L. The Relabelling rule states that if E can perform an action α then $E[f]$ can perform $f(\alpha)$. The rule for Recursion determines the transitions of the agent $\mu X : E$ by considering the transitions obtained after a proper unwinding of the agent. Finally, the inactive agent **0** has no transitions as no rule applies to it. In the following, the transition rules are illustrated with examples:

Example 2.2 Consider the agent **a.0|b.0**. From the **Act** rule we have $\mathtt{a.0} \xrightarrow{\mathtt{a}} \mathtt{0}$. Therefore using $\mathbf{Com_1}$ we get the transition:

$$\mathtt{a.0|b.0} \xrightarrow{\mathtt{a}} \mathtt{0|b.0}$$

Similarly, using $\mathbf{Com_2}$ we get the transition

$$\mathtt{a.0|b.0} \xrightarrow{\mathtt{b}} \mathtt{a.0|0}$$

From the two new states **0|b.0** and **a.0|0**, we get two more transitions using $\mathbf{Com_2}$ and $\mathbf{Com_1}$ respectively:

$$\mathtt{0|b.0} \xrightarrow{\mathtt{b}} \mathtt{0|0}$$

$$\mathtt{a.0|0} \xrightarrow{\mathtt{a}} \mathtt{0|0}$$

Clearly, **0|0** can perform no further action. Therefore, the agent **a.0|b.0** can either perform an action **a** followed by **b** or an action **b** followed by **a**.

Example 2.3 Consider the agent $\mu X : (\mathtt{a}.X)$. Clearly, this is an agent which continuously performs an action a. We verify this fact by examining what transitions are applicable. Using the **Rec** rule we conclude that the transitions of the agent can be obtained by considering the transitions of

$$\mathtt{a}.X\{(\mu X : (\mathtt{a}.X))/X\} \quad = \quad \mathtt{a}.(\mu X : (\mathtt{a}.X))$$

Using the **Act** rule we get that

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

Consequently, $\mu X : (\mathtt{a}.X) \xrightarrow{\mathtt{a}} \mu X : (\mathtt{a}.X)$. Therefore, after an a action, the resulting agent is the same as the initial one and can perform the same action forever.

Example 2.4 Consider the agent $\mathtt{a}.\mathbf{0} + \mathtt{b}.\mathbf{0}$. The only rules that are applicable are $\mathbf{Sum_1}$ and $\mathbf{Sum_2}$ and they give respectively:

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

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

Therefore, the agent $\mathtt{a}.\mathbf{0} + \mathtt{b}.\mathbf{0}$ can either perform a and then stop or perform b and then stop.

Example 2.5 As a final example, consider the agent $\mathtt{a}.\mathbf{0}|\overline{\mathtt{a}}.\mathbf{0}$. Clearly, $\mathbf{Com_1}$, $\mathbf{Com_2}$ and $\mathbf{Com_3}$ are all applicable and give:

$$\mathtt{a}.\mathbf{0}|\overline{\mathtt{a}}.\mathbf{0} \xrightarrow{\mathtt{a}} \mathbf{0}|\overline{\mathtt{a}}.\mathbf{0}$$
$$\mathtt{a}.\mathbf{0}|\overline{\mathtt{a}}.\mathbf{0} \xrightarrow{\overline{\mathtt{a}}} \mathtt{a}.\mathbf{0}|\mathbf{0}$$
$$\mathtt{a}.\mathbf{0}|\overline{\mathtt{a}}.\mathbf{0} \xrightarrow{\tau} \mathbf{0}|\mathbf{0}$$

From the new states, we can get the following two transitions:

$$\mathtt{a.0|0} \xrightarrow{\mathtt{a}} \mathtt{0|0}$$

$$\mathtt{0|\overline{a}.0} \xrightarrow{\overline{\mathtt{a}}} \mathtt{0|0}$$

The above are the only possible transitions of the specific agent.

2.4 Derivatives and Derivation Trees

Whenever $E \xrightarrow{\alpha} E'$, we call the pair (α, E') an *immediate derivative* of E, we call α an action of E and E' an *α-derivative* of E.

Analogously, whenever $E \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} E'$, we call $(\alpha_1 \cdots \alpha_n, E')$ a *derivative* of E, we call $\alpha_1 \cdots \alpha_n$ an *action sequence* of E and we call E' an $\alpha_1 \cdots \alpha_n -$ *derivative* of E.

It is convenient to collect the derivatives of an agent E into the *derivation tree* of E. The nodes of the tree are expressions while all the immediate derivatives of a node are represented by outgoing arcs. Figure 2.1 shows the derivation trees for all the examples given in the previous section. The derivation tree can be thought of as a very simple machine-like model of agents, in the sense that it describes how the agent would behave in practice. However, *it has the drawback that it can not represent whether parts of an agent work independently or concurrently.* Consider for example the derivation tree of the agent $\mathtt{a.0} \mid \mathtt{b.0}$, which is represented in Figure 2.1(a). It is clear that the tree does not represent the *concurrency* between the two components $\mathtt{a.0}$ and $\mathtt{b.0}$ of the initial agent. Instead, it uses the *interleaving model of concurrency* where concurrency is a special case of sequentiality. What the figure actually implies is that the given agent can either perform a

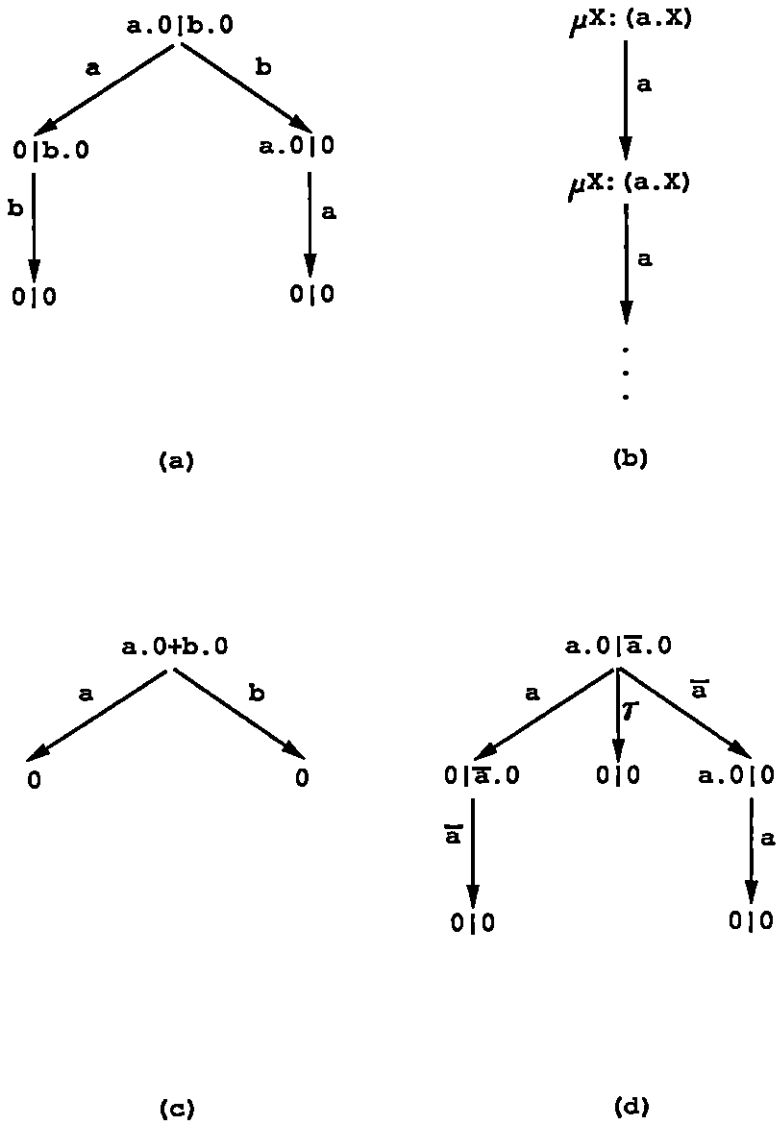


Figure 2.1: Examples of Derivation Trees

followed by b or $\overline{\mathtt{b}}$ followed by a. At the level of machines, we do not like such a reduction because it hides whether components of the machine can work independently or not. Therefore, we need an extension of derivation trees which represents the inherent parallelism of agents explicitly. This extension is Petri nets and is described in the next two chapters.

Chapter 3

Petri Nets

Petri nets are a tool for the study of systems. Petri net theory allows a system to be modelled by a Petri net, a mathematical representation of the system. Analysis of the Petri net can then, hopefully, reveal important information about the structure and dynamic behavior of the modelled system. This information can be used to evaluate the modelled system and suggest improvements or changes. Thus, the development of a theory of Petri nets is based on their application in the modelling and design of systems.

3.1 Definitions

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 adopted here has turned out to be very useful in relating Petri nets with process algebras such as CCS. Let *Act* be the set of actions, as this was defined in section 2.2. Then:

Definition 3.1 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. $T \subseteq \Delta(Pl) \times Act \times \Delta(Pl)$ is a set of *transitions.*
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 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 3.1 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 ◯ 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 3.1. The initial marking M_0 is represented by putting a dot • into the circle of each place in M_0.

Example 3.2 Using the above conventions, the net of Example 3.1 is represented as shown in Figure 3.2.

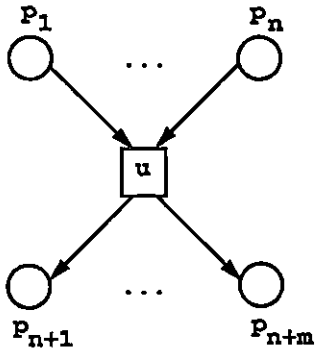


Figure 3.1: Graphical Representation of a Transition

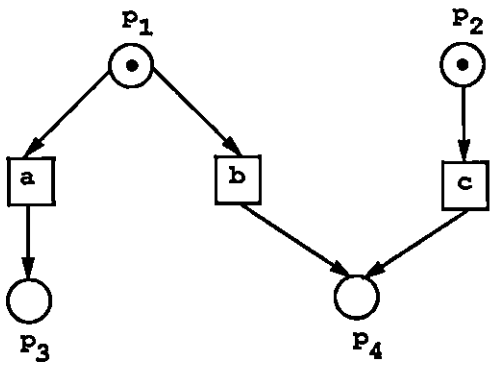


Figure 3.2: The Petri net of Example 3.1

3.2 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. To describe this situation, the notion of a *multiset* is used, i.e., a set where multiple occurrences of elements are allowed. 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 3.2 A *marking* M of a Petri net $R = (Pl, T, M_0)$ is a multiset over Pl.

Let $\sqsubseteq$, $\sqcup$ and $-$ denote multiset inclusion, union and difference respectively. Then the execution of a transition is defined as follows:

Definition 3.3 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 3.3 In Example 3.2, 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 3.3(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 3.3(b).

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

Definition 3.4 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 $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n = M$.

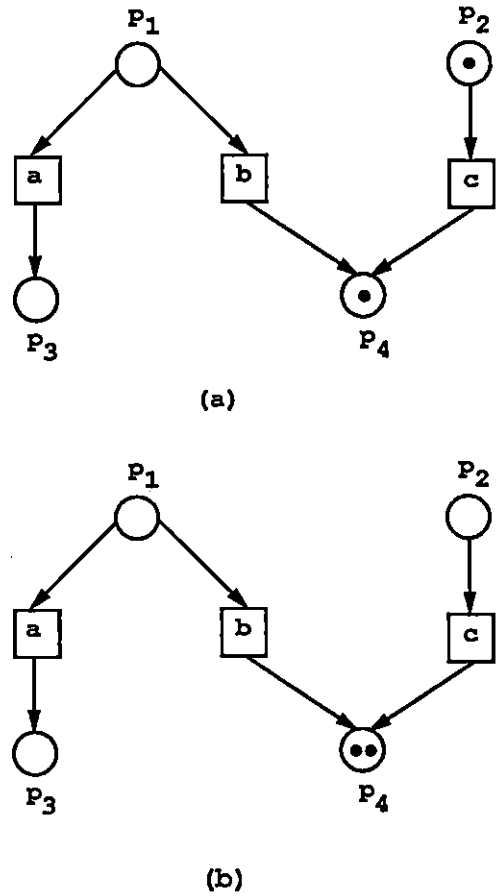


Figure 3.3: Executions of Transitions

In Figure 3.3(b), place p_4 contains two tokens. A safe Petri net is one in which such a situation does not arise:

Definition 3.5 A net R is *safe* if and only if in every reachable marking, the number of tokens per place is either zero or one.

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

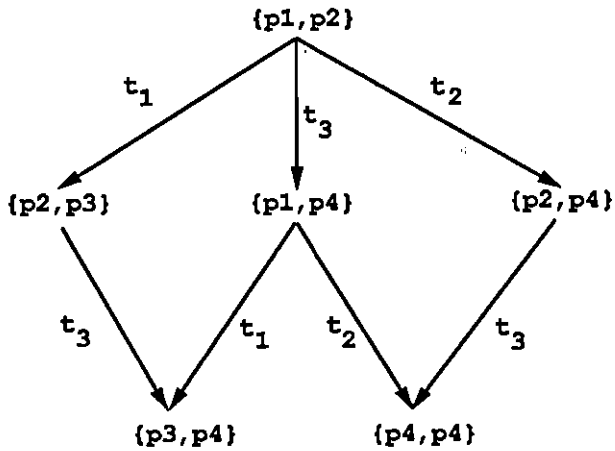


Figure 3.4: Reachability Graph

transition firing which transforms one marking into another. Formally:

Definition 3.6 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 3.4 represents the reachability graph of the Petri net in Figure 3.2. Labels t_1, t_2 and t_3 indicate the transitions with actions a, b and c respectively.

Chapter 4

Concurrent Semantics for CCS

This chapter deals with defining concurrent semantics for CCS agents using Petri nets. The main idea 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 [23] and also used in [24],[21],[22] and [25]. 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.

4.1 The Decomposition

In the following we define the *sequential components* and the *decomposition function*:

Definition 4.1 The set *Seq* of *sequential components*, with typical element G, consists of all terms generated by the following rules:

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

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 4.2 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 4.1 Consider the agent (a.0|b.0). Its decomposition gives a set of two sequential components:

$$\begin{aligned} dec(\mathtt{a.0}|\mathtt{b.0}) &= dec(\mathtt{a.0})|id \cup id|dec(\mathtt{b.0}) \\ &= \{(\mathtt{a.0})|id\} \cup \{id|(\mathtt{b.0})\} \\ &= \{(\mathtt{a.0})|id, id|(\mathtt{b.0})\} \end{aligned}$$

Example 4.2 Consider the agent (a.0|b.0)|c.0. Its decomposition gives a set of three sequential components:

$$\begin{aligned} dec(((\mathtt{a.0}|\mathtt{b.0})|\mathtt{c.0})) &= dec((\mathtt{a.0}|\mathtt{b.0}))|id \cup id|dec(\mathtt{c.0}) \\ &= \{((\mathtt{a.0})|id)|id, (id|(\mathtt{b.0}))|id\} \cup \{id|(\mathtt{c.0})\} \\ &= \{((\mathtt{a.0})|id)|id, (id|(\mathtt{b.0}))|id, id|(\mathtt{c.0})\} \end{aligned}$$

4.2 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 4.3 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) $\{\alpha.E\} \xrightarrow{\alpha} dec(E)$

(Sum$_1$) $\{E_1\} \xrightarrow{\alpha} I_1$ **implies** $\{E_1 + E_2\} \xrightarrow{\alpha} I_1$

(**Sum$_2$**) $\{E_2\} \xrightarrow{\alpha} I_2$ **implies** $\{E_1 + E_2\} \xrightarrow{\alpha} I_2$

(**Com$_1$**) $I \xrightarrow{\alpha} I'$ **implies** $I|id \xrightarrow{\alpha} I'|id$

(**Com$_2$**) $I \xrightarrow{\alpha} I'$ **implies** $id|I \xrightarrow{\alpha} id|I'$

(**Com$_3$**) $I_1 \xrightarrow{1} I_1'$ **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** $\{\mu X : E\} \xrightarrow{\alpha} I'$

The above definition is now illustrated with some examples.

Example 4.3 Consider the agent $\mu X : (\mathtt{a}.X)$. We construct its corresponding Petri net. The initial marking of the net is

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

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

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

The corresponding Petri net is shown in Figure 4.1(a).

Example 4.4 The following example shows the main difference between the transitional and the concurrent semantics. Consider the process term $\mathtt{a.0}|\mathtt{b.0}$. From example 4.1

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

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

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

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

The corresponding Petri net is shown in Figure 4.1(b). This Petri net explicitly shows the inherent parallelism of the agent in contrast to the transitional semantics of Chapter 2.

Example 4.5 Consider the agent $\mathtt{a}.\mathbf{0} + \mathtt{b}.\mathbf{0}$. The initial marking of the corresponding Petri net is

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

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

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

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

The corresponding Petri net is shown in Figure 4.1(c).

Example 4.6 As a final example on the concurrent semantics, consider the agent $\mathtt{a}.\mathbf{0}|\overline{\mathtt{a}}.\mathbf{0}$. Using the decomposition function *dec* we get the initial marking of the corresponding Petri net:

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

Starting from the initial marking, three transitions are applicable:

$$\{(\mathtt{a.0})|id\} \xrightarrow{\mathtt{a}} \{\mathtt{0}|id\}$$
$$\{id|(\overline{\mathtt{a}}.\mathtt{0})\} \xrightarrow{\overline{\mathtt{a}}} \{id|\mathtt{0}\}$$
$$\{(\mathtt{a.0})|id\,,\, id|(\overline{\mathtt{a}}.\mathtt{0})\} \xrightarrow{\tau} \{\mathtt{0}|id\,,\, id|\mathtt{0}\}$$

The corresponding Petri net is shown in Figure 4.1(d).

As a last remark on the concurrent semantics of agents, we have the following theorem [25, page 143]:

Theorem 4.1 For every agent E, the corresponding Petri net $\mathcal{R}[E]$ is safe.

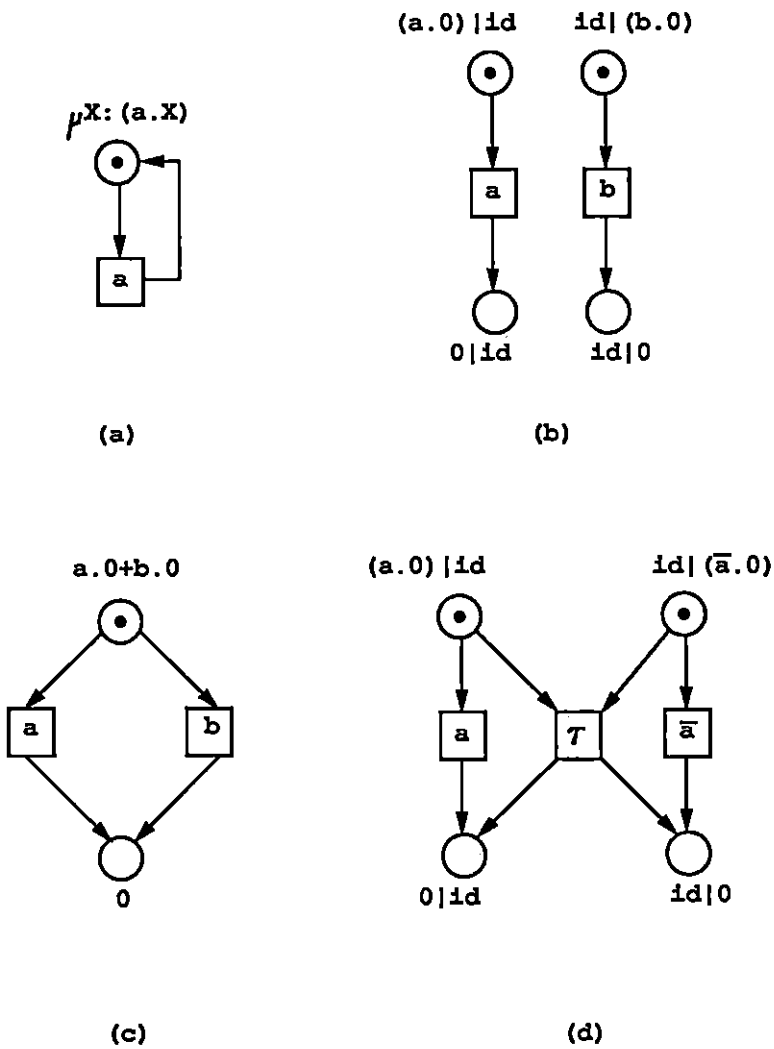


Figure 4.1: Examples of Concurrent Semantics

Chapter 5

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 [3] that "proof of absence of deadlock, even for quite simple processes, will remain the responsibility of the designer of concurrent systems". In this chapter, 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 sections, 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.

5.1 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 5.1 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 [25, page 143]. Therefore, instead of examining the states of the agent, one can examine the markings of the net. This means that:

Definition 5.2 A CCS agent contains a *deadlock* if and only if the corresponding Petri net contains a *deadlock marking.*

Deadlock markings of a net can be identified by examining its reachability graph:

Theorem 5.1 A sink node in the reachability graph of a Petri net indicates the existence of a deadlock marking.

Proof: As a node of the reachability graph indicates a reachable marking of the Petri net, a node without any outgoing edges indicates a marking of the net where no transition can fire, i.e., a deadlock marking. ■

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

Theorem 5.2 The reachability graph of a safe Petri net is finite.

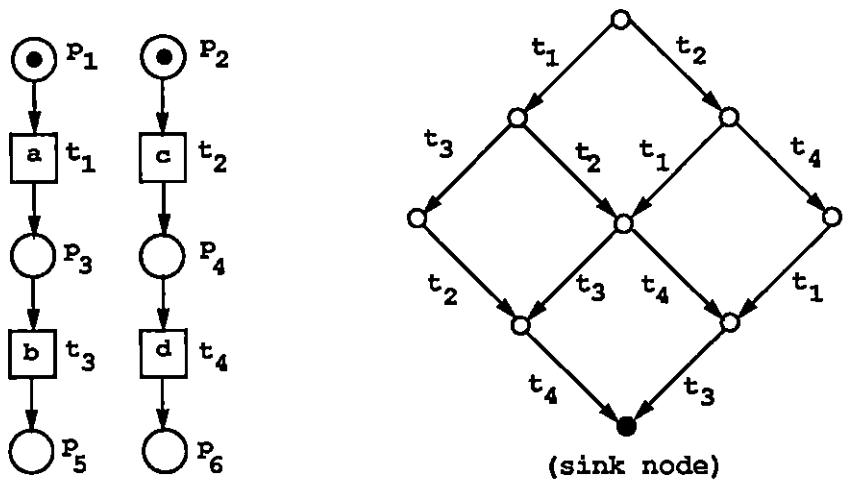


Figure 5.1: Petri Net of $a.b.\mathbf{0}|c.d.\mathbf{0}$ and its Reachability Graph.

Proof: A safe Petri net can not hold more than one token in each of its places. Therefore, the possible markings do not exceed 2^n where n is the number of places of the net. As the number of vertices of the reachability graph is equal to the number of the reachable markings, we conclude that it is finite. ■

The above theorems 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 5.1 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 5.1, 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

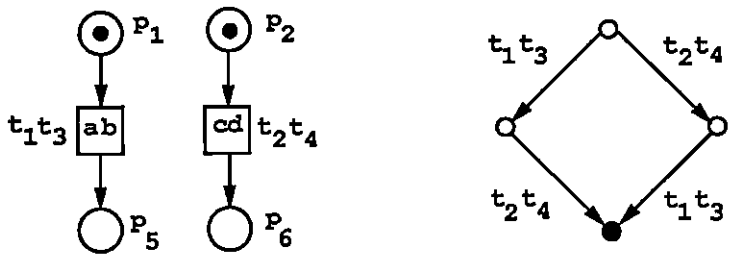


Figure 5.2: Reduced Petri Net of $a.b.0|c.d.0$ and its Reachability Graph.

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 5.2, 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.

5.2 Reduction Techniques

In order to ease the analysis of Petri nets, a number of transformations have been proposed in the literature ([27] – [34]), which simplify 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.*

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, or in other words the behavior of the net remains the same after the removal of this kind of places. 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 can be easily understood by the accompanying figures.

Definition 5.3 Let $R = (Pl, T, M_0)$ be a 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. $\forall t \in (T - (\{h\} \cup F)),\ p \notin pre(t) \wedge p \notin post(t)$
   (Except for h and the transitions belonging to F, no other transition is connected to p).

7. $p \notin M_0$.
   (Place p holds no token initially).

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

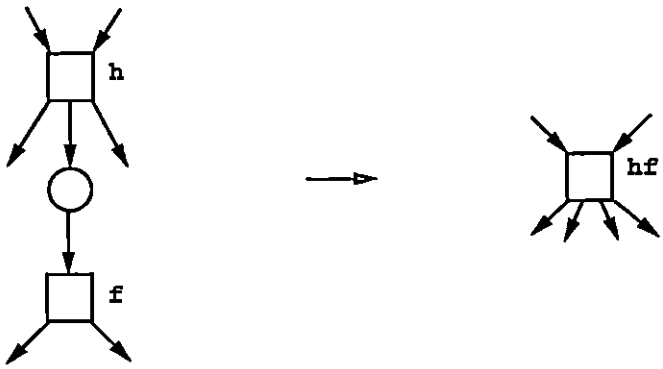
Definition 5.4 Let $R = (Pl, T, M_0)$ be a Petri net, and let $F \subseteq T$, $h \in T$ and $p \in Pl$ satisfy the conditions of Definition 5.3. Then, the system resulting by the post-fusion of F and h is $R' = (Pl', T', M_0)$, with:

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

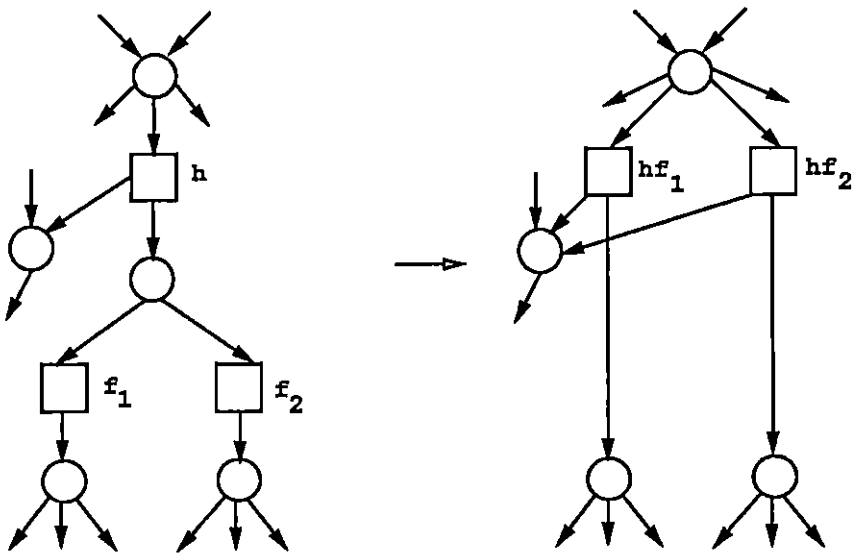
$$\begin{array}{lll} \{hf_i : & f_i \in F \text{ and} & \\ & pre(hf_i) & = pre(h), \\ & post(hf_i) & = (post(h) - \{p\}) \cup post(f_i), \\ & act(hf_i) & = act(h)act(f_i)\ \} \end{array}$$

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 5.4 is illustrated in Figure 5.3(a) for the case $F = \{f\}$ and in Figure 5.3(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 $f_i \in F$.



(a)



(b)

Figure 5.3: Examples of Post-Fusion of Transitions

Definition 5.5 Let $R = (Pl, T, M_0)$ be a 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 belonging to 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 5.6 Let $R = (Pl, T, M_0)$ be a Petri net, and let $F \subseteq T$, $h \in T$ and $p \in Pl$ satisfy the conditions of Definition 5.5. 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:

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

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 5.6 is illustrated in Figure 5.4(a) for the case $F = \{f\}$ and in Figure 5.4(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 5.7 Let $R = (Pl, T, M_0)$ be a 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).

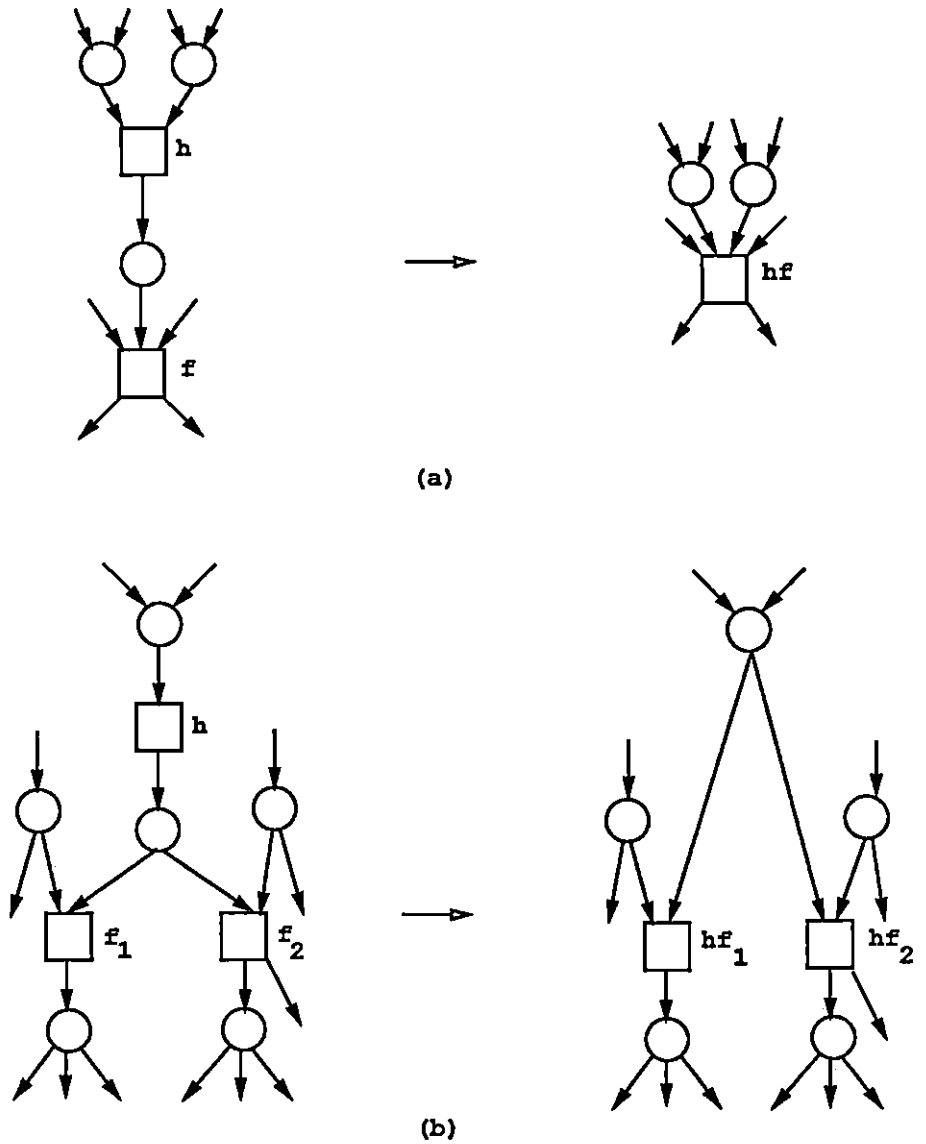


Figure 5.4: Examples of Pre-fusion

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}).

Definition 5.8 Let $R = (Pl, T, M_0)$ be a Petri net, and let $p, p_0, ..., p_{n-1} \in Pl$ and $t_0, ..., t_n \in T$ satisfy the conditions of Definition 5.7. 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 5.5(a) and 5.5(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 [28, pages 361-367]. Formally:

Theorem 5.3 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

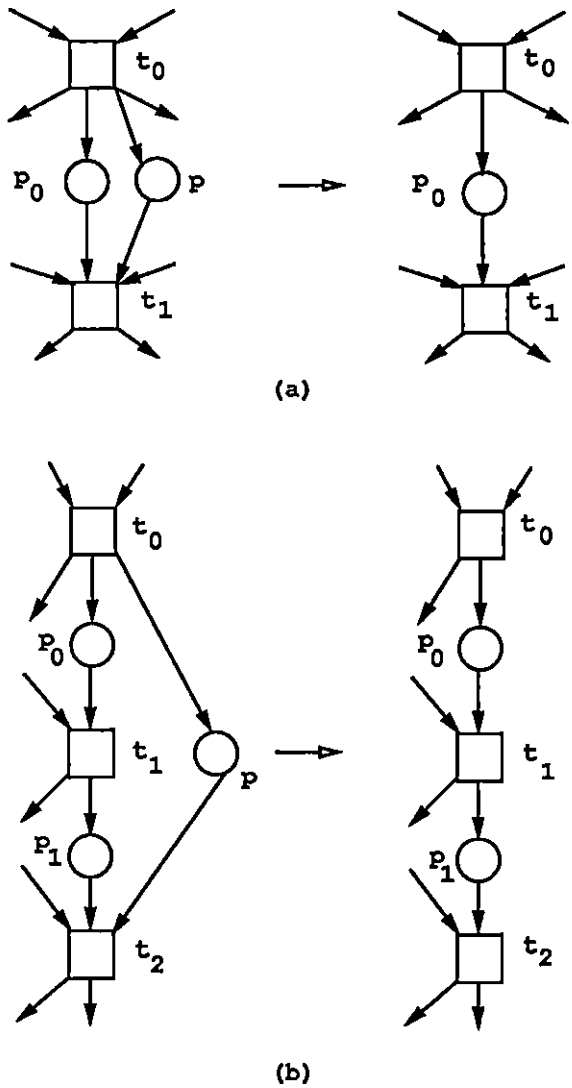


Figure 5.5: Redundant Places

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.

5.3 Extending the Theory of Reductions

In this section, we extend the theory of reductions in order to get an algorithm for detecting deadlocks in CCS agents. More specifically, we consider *sequences* of reductions, not just single reductions. As we aim at detecting *all* the deadlocks of a CCS agent, we need a stronger version of Theorem 5.3 which will ensure that no deadlock is lost or added during the reductions. On the other hand, we do not just need to detect that a deadlock exists: we are interested in finding out 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. In the following we formalize these ideas. 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 5.3, R_i is also safe. Before showing that R_0 and R_i contain the same number of deadlocks, we need the following theorems:

Theorem 5.4 Let R_1 be the Petri net resulting from 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. If $t = f_i$, then h has already fired in R_0, which means that hf_i can fire in R_1, and therefore R_1 is not deadlocked. ■

Corollary 5.1 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 5.1 Consider the following process term:

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

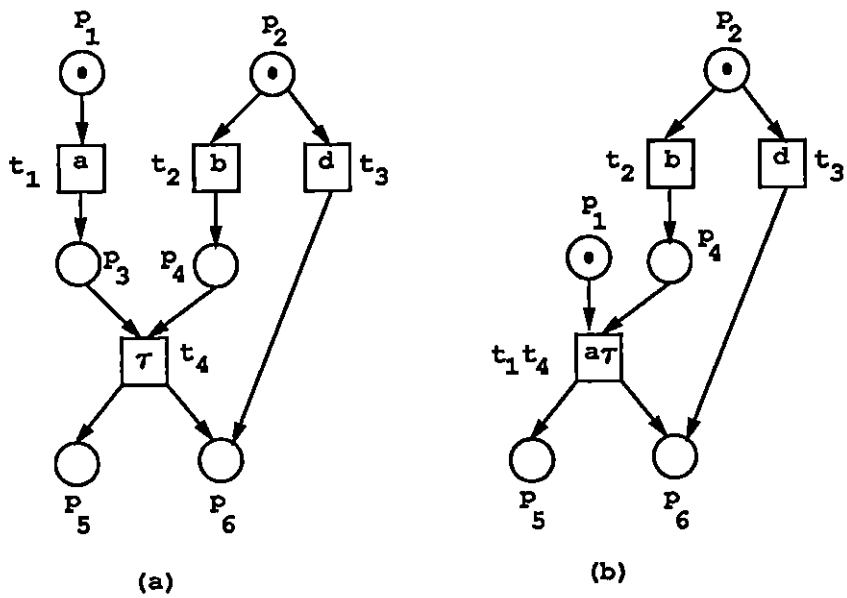


Figure 5.6: Illustration of Theorem 5.5

that yields the Petri net of Figure 5.6(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 5.6(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 5.5 Let R_1 be the Petri net resulting from 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 $(M' - pre(h)) \sqcup \{p\}$.

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. If $t = f_i$, then h must have been enabled. Therefore, hf_i is enabled in R_1 as well and R_1 is not deadlocked. If $t = h$, then none of the f_i can be enabled, because if this was the case, then the corresponding hf_i would be able to fire in R_1, which implies that R_1 is not deadlocked. Hence, the only possible transition that can fire in R_0 after the occurrence of σ, is h. Therefore, we have two cases: 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 5.1, 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 5.6 Let R_i be the Petri net resulting from R_0 after a sequence of pre-fusions of transitions, and let σ be a sequence of transitions leading R_i to deadlock. Then, σ can be extended to lead R_0 to deadlock.

Proof: Let F_m and h_m denote the transitions that are pre-fused in the m-th step of the reductions. Then, by Theorem 5.5, if σ leads R_i to deadlock, then either σh_{i-1} or σ leads R_{i-1} to deadlock. This means that σw_{i-1} leads R_{i-1} to deadlock, where $w_{i-1} = h_{i-1}$ or w_{i-1} is empty. Continuing in this way, we get that $\sigma w_{i-1} w_{i-2} ... w_0$ leads R_0 to deadlock. Therefore, σ can be extended to lead 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.

Theorem 5.7 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 5.2 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.

We can now prove that the reduction techniques do not change the number of deadlock markings of a Petri net. The following theorems, one for each technique, show this claim:

Theorem 5.8 Let R_1 be the Petri net resulting from R_0 after a post-fusion of transitions F with transition h. Let δ_0 and δ_1 be the number of deadlock markings in R_0 and R_1 correspondingly. Then, $\delta_0 = \delta_1$.

Proof: We show that R_0 and R_1 contain exactly the same number of deadlock markings by proving that for every deadlock marking of R_0 there exists a *distinct* deadlock marking of R_1 and vice versa.

Let p be the place that is eliminated by this reduction. Let M be a deadlock marking of R_0 and let σ be a sequence of transitions that leads to M. Clearly, $p \notin M$ because whenever p holds a token, some $f_i \in F$ can fire. The fact that $p \notin M$ implies that for every firing of h, there exists in σ a corresponding firing of some $f_i \in F$ after h. We can reorder σ so as to have the corresponding pairs (h, f_i) consecutive. In this way, we get σ' which when applied to R_0 leads to the same marking M as σ. But then, we can apply σ' on R_1, where a consecutive pair of transitions (h, f_i) of σ' corresponds to firing transition hf_i of R_1. In this way we get marking M of R_1. This is clearly a deadlock marking of R_1, because if any transitions could fire, then some transition would be able to fire in R_0 under marking M. Therefore, every deadlock marking M of R_0 is also a distinct deadlock marking of R_1.

Let M' be a deadlock marking of R_1. Then by Theorem 5.4, M is also a deadlock marking of R_0. Therefore, every deadlock marking of R_1 is also a distinct deadlock marking of R_0.

From the above, we conclude that R_0 and R_1 have the same number of deadlocks, i.e, $\delta_0 = \delta_1$. ■

Theorem 5.9 Let R_1 be the Petri net resulting from R_0 after a pre-fusion of transitions F with transition h. Let δ_0 and δ_1 be the number of deadlock

markings in R_0 and R_1 correspondingly. Then, $\delta_0 = \delta_1$.

Proof: We show that R_0 and R_1 contain exactly the same number of deadlock markings by proving that for every deadlock marking of R_0 there exists a *distinct* deadlock marking of R_1 and vice versa.

Let p be the place that is eliminated by this reduction. Let M be a deadlock marking of R_0 and let σ be a sequence of transitions that leads to M. In contrast to post-fusion, p may belong to M. This means that we may have a firing of h which is not necessarily followed by some $f_i \in F$. We can reorder σ to have all the corresponding pairs (h, f_i) consecutive getting σ' in this way. This can be done because we can postpone all the h firings to occur just before the corresponding f_i firings. If $p \notin M$, σ' leads R_1 to the same deadlock marking M. If on the other hand $p \in M$, then we do not get the same marking M, but $M' = (M - \{p\}) \sqcup pre(h)$ by Theorem 5.5. Therefore, every deadlock marking M of R_0 has a corresponding marking M' of R_1. What remains to be shown is that given two distinct deadlock markings M_1 and M_2 of R_0, M_1' and M_2' are distinct. If both M_1 and M_2 contain p or if they both do not, then clearly M_1' and M_2' are distinct. If M_1 contains p and M_2 does not, then $M_1' = (M_1 - \{p\}) \sqcup pre(h)$ and $M_2' = M_2$. But M_2' does not contain the whole $pre(h)$ because if it did, h would be able to fire under M_2 in R_0. Therefore, every deadlock marking M of R_0, has a distinct deadlock marking of R_1.

On the other hand, let M' be a deadlock of R_1 and σ' be a sequence of transitions leading to M'. Then by Theorem 5.5, either σ' leads R_0 to deadlock marking $M = M'$ or $\sigma' h$ leads R_0 to deadlock marking $M = (M' - pre(h)) \sqcup \{p\}$. Therefore, every deadlock marking of R_1 has a corresponding

marking of R_0. What remains to be shown is that given two markings M_1' and M_2' of R_1, M_1 and M_2 are distinct. If both M_1' and M_2' contain $pre(h)$ or they both do not, then clearly $M_1 \neq M_2$. If M_1' contains $pre(h)$ and M_2' does not, then M_1 contains p while M_2 does not. Therefore, every deadlock marking of R_1 has a corresponding distinct marking of R_0.

From the above, we conclude that R_0 and R_1 have the same number of deadlocks, i.e, $\delta_0 = \delta_1$. ■

Theorem 5.10 Let R_1 be the Petri net resulting from R_0 after eliminating redundant place p. Let δ_0 and δ_1 be the number of deadlock markings in R_0 and R_1 correspondingly. Then, $\delta_0 = \delta_1$.

Proof: We show that R_0 and R_1 contain exactly the same number of deadlock markings by proving that for every deadlock marking of R_0 there exists a *distinct* deadlock marking of R_1 and vice versa.

Let M be a deadlock marking of R_0. By the definition of a redundant place, the behavior of the net does not change after its elimination. Therefore, the same sequence of transitions that led to M in R_0 can be repeated in R_1 leading to a deadlock marking M'. If M contains place p then $M' = M - \{p\}$, otherwise $M' = M$. Therefore, each deadlock marking of R_0 has a corresponding deadlock marking of R_1. What must be shown is that given two deadlock markings M_1 and M_2 of R_0, they have distinct corresponding deadlock markings in R_1, i.e., $M_1' \neq M_2'$. If both M_1 and M_2 contain p or if they both do not contain p, we clearly have $M_1' \neq M_2'$. If M_1 contains p and M_2 does not, then M_1 (as can be seen from Figure 5.5) contains at least one more place that does not belong to M_2. Therefore, $M_1' \neq M_2'$.

Let on the other hand M' be a deadlock marking of R_1. If M' contains a place p_i (see Figure 5.5), then the corresponding deadlock marking in R_0 is $M = M' \sqcup \{p\}$, otherwise $M = M'$. Consider two distinct deadlock markings M_1' and M_2' of R_1. We show that $M_1 \neq M_2$. If both of M_1' and M_2' do not contain any place p_i or if they both contain the same place p_i, then clearly $M_1 \neq M_2$. If one of them contains a p_i and the other does not, then again $M_1 \neq M_2$ because one of them contains p while the other does not.

From the above, we conclude that R_0 and R_1 have the same number of deadlocks, i.e, $\delta_0 = \delta_1$. ■

Corollary 5.3 Let R_i be the Petri net resulting from R_0 after a sequence of reductions. Let δ_0 and δ_i be the number of deadlock markings in R_0 and R_i correspondingly. Then, $\delta_0 = \delta_i$.

5.4 The Algorithm

In this section, the proposed approach for deadlock detection in CCS agents is presented. 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. In order to reduce the number of states that have to be searched for deadlock, a sequence of transformations is applied on the net. Initially, redundant places are removed. From Figure 5.5, 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. When the final irreducible net is obtained, its 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 described 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: Eliminate Redundant Places;
WHILE there exist post-fusable transitions F_m and h_m **DO**
Apply post-fusion rule;
END;
Initialize H to empty;
WHILE there exist pre-fusable transitions F_m and h_m **DO**
Apply pre-fusion rule;
$H := h_m \circ H$;
END;
Let H be equal to $h_i \circ h_{i-1} \circ \cdots \circ h_1$, 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 p_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 p_n **DO**
  Follow the path in the initial Petri net R_0;
  Let M_n be the marking where the path leads;
  FOR $j = i$ **TO** 1 **DO**
    IF h_j can fire in R_0 **THEN**
      $p_n := p_n \circ h_j$;
  END;
  Output p_n;
END;

Before illustrating the algorithm by an example, some comments are necessary:

First, the algorithm always terminates. This is due to the fact that 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.

Second, the reductions can be performed very efficiently. In the case of a fusion of transitions, one must consider a place of the net and check if the transitions in its preset and postset satisfy the required conditions. However, the number of transitions in the preset and the postset of a place, is very small in practice, and of course bounded by the number of transitions in the whole net. Therefore, fusions can be performed very efficiently. On the other hand, 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 5.7. These properties can be easily validated by a depth first search algorithm that starts from t_0. During the depth first search, all the places visited, are examined to ensure that they have just one

input and one output. Clearly, redundant places can be detected efficiently as well.

Concluding, we should point out 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 performing the reductions on the net, than trying to find techniques that would operate on the reachability graph directly.

5.5 An Example

As an example of the applicability of the above method we use the classical "Dining Philosophers" problem. It is well known from the literature that there exists an incorrect solution 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:

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

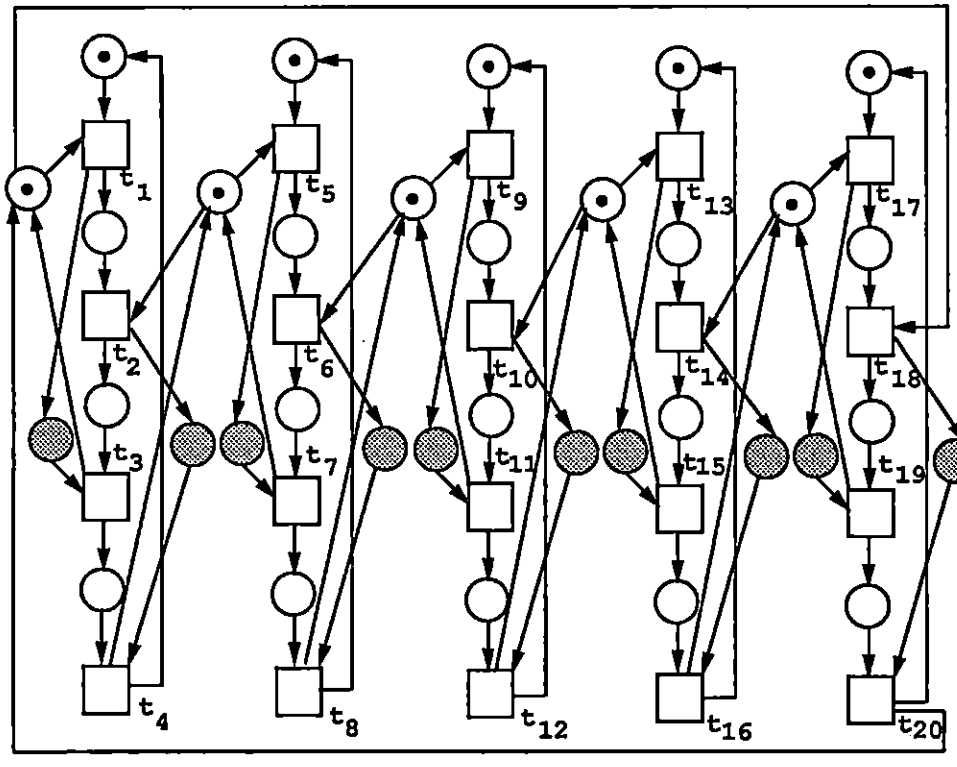


Figure 5.7: The Dining Philosophers

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 5.7. In this Figure, the darkened places are redundant places and are discovered before the WHILE loop of the algorithm is entered. The removal of these places reduces the Petri net to the one shown in Figure 5.8. 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

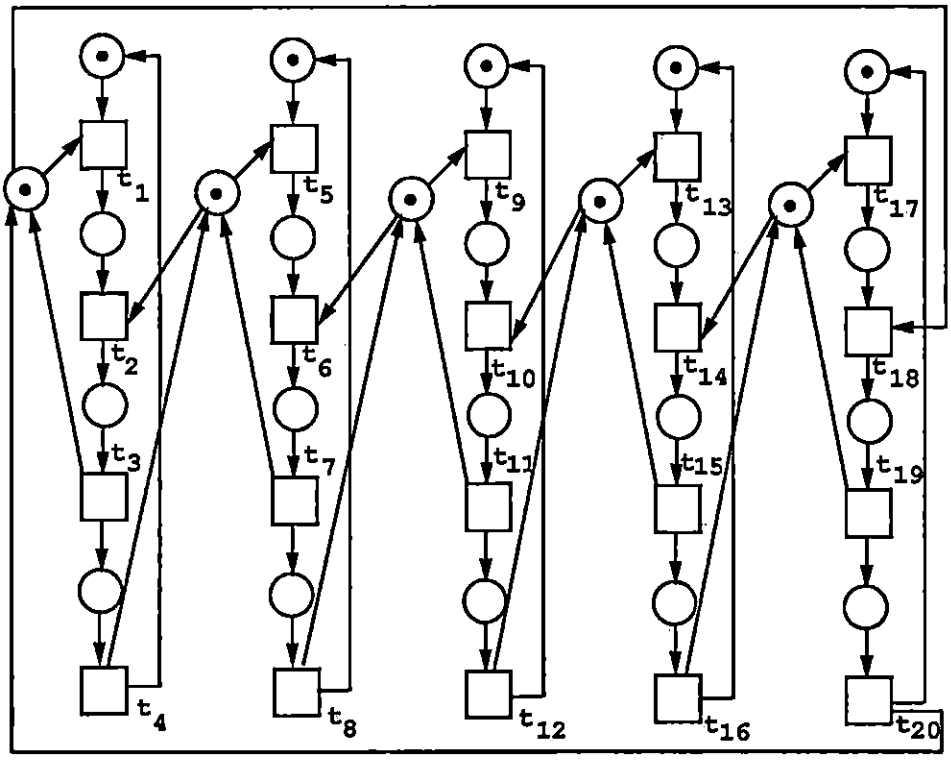


Figure 5.8: The Net after Elimination of the Redundant Places

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 5.9. 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 $\sigma = t_1t_5t_9t_{13}t_{17}$ 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.

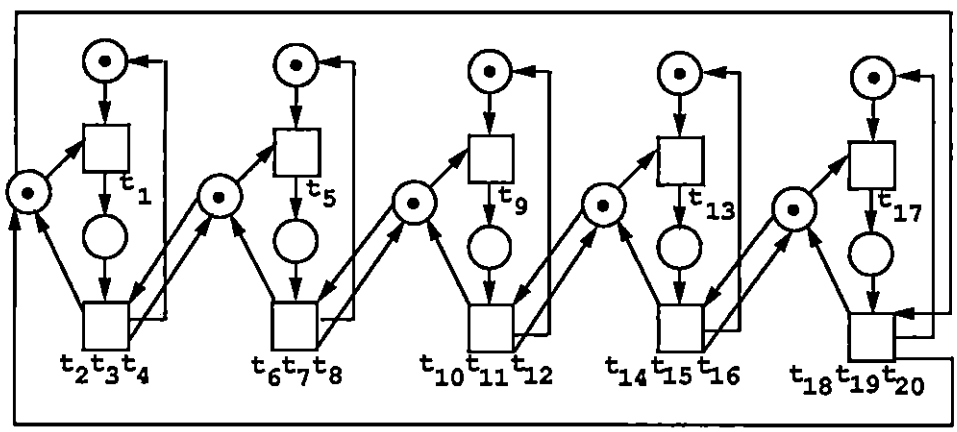


Figure 5.9: The Net after Postfusion of Transitions

Chapter 6

An Implementation of the Algorithm

In this chapter we describe a Prolog system which can be used to detect deadlocks in CCS agents. Given an agent E, the system produces the corresponding Petri net $\mathcal{R}[E]$. The Petri net is then reduced and the deadlocks are located. The system is interesting because:

- This is the first – to our knowledge – implementation of the concurrent semantics of CCS, and has given us the chance to experiment with agents of considerable size.
- The implementation can be used to evaluate the proposed deadlock detection algorithm.

In the following, the structure of the system is described and its use is demonstrated by examples.

6.1 System Description

The theory that was developed in the previous chapters has been implemented using the Prolog programming language. The reasons that led to this decision are:

- A major characteristic of Prolog programming is that it allows the programmer to concentrate on telling the computer *what* the problem is, rather than giving it a description of *how* to solve it. The theoretical nature of the material presented in the previous chapters would make the use of another programming language very tedious.
- Many transformations and definitions of the previous chapters, can be directly translated to Prolog programs.

The system takes as input an agent E and produces as output a set of transition sequences that lead to deadlocks in $\mathcal{R}[E]$. The interaction of the user and the system is performed with a query of the form:

$$? - \texttt{detect(E, Deadlock\_Paths)}.$$

In the above query, `E` is explicitly specified by the user. On the other hand, the argument `Deadlock_Paths` is acting as the output of the program. `Detect` is the main relation of the program and it in fact performs the algorithm of Chapter 5. It is defined as follows:

```
detect(E,Deadlock_Paths):-
        convert(E,Petri_Net),
        reduce(Petri_Net,Reduced_Petri_Net,H),
        reachability(Reduced_Petri_Net,Graph),
        analysis(Graph,Paths),
        extend(Petri_Net,Paths,Deadlock_Paths,H).
```

We should note here that each one of the above relations corresponds to a distinct step of the algorithm of chapter 5. We comment on each relation, avoiding to give low level implementation details:

1. Relation `convert(E,Petri_Net)` is true when `Petri_Net` is the concurrent semantics of agent `E`. The implementation of this relation consists of two main steps:

   - The initial marking M_0 of $\mathcal{R}[E]$ is computed using the relation `dec(E,M0)`. This can be done by directly implementing the function *dec* of Definition 4.2.
   - Starting from M_0, the system investigates what transitions are applicable. It is remarkable that the rules for the concurrent semantics of CCS can be directly translated into Prolog programs. Consider for example the general rule:

     $$I_1 \stackrel{\alpha_1}{\rightarrow} I_1', ..., I_m \stackrel{\alpha_m}{\rightarrow} I_m' \quad \textbf{implies} \quad I \stackrel{\alpha}{\rightarrow} I' \quad \textbf{where} \quad \textit{condition}$$

     This can be written as:

```
transition(I,A,Ip):-
        Cond,
        transition(I1,A1,I1p),
                .
                .
                .
        transition(IM,AM,IMp).
```

     where `Cond` is a piece of code that examines the validity of *condition*. When all the possible transitions have been investigated, the Petri net is complete, and the system proceeds to the next step.

2. Relation `reduce(Petri_Net, Reduced_Petri_Net,H)` is true when, after the application of the transformation rules, `Petri_Net` has been reduced to `Reduced_Petri_Net`. Here, `H` is used to hold all the h_m's that are pre-fused during the reductions. The high level description of this phase is:

```
reduce(Petri_Net,Reduced_Petri_Net,H):-
        eliminate_redundant_places(Petri_Net,Net1),
        postfuse_transitions(Net1,Net2),
        prefuse_transitions(Net2,Reduced_Petri_Net,H).
```

   The relations `eliminate_redundant_places`, `postfuse_transitions` and `prefuse_transitions` implement the corresponding reductions.

3. The relation `reachability(Reduced_Petri_Net,Graph)` is true when `Graph` is the reachability graph that corresponds to the reduced Petri net. This relation is implemented by starting from the initial marking M_0 of the net and examining all those markings that can be reached by firing a single transition. The new markings are then examined and this is repeated until all the possible markings have been considered. This procedure is guaranteed to terminate as the reachability graph of a safe net is finite.

4. The relation `analysis(Graph,Paths)` computes for each sink node of the reachability graph, a path that starts from the root and leads to the node. This is done using a depth first search algorithm.

5. The relation `extend(Petri_Net,Paths,Deadlock_Paths,H)` is used to find the paths that lead to deadlock in the initial Petri net. If `H` is empty, this means that the pre-fusion rule has not been used, and then

`Deadlock_Paths` will be the same as `Paths`. If `H` is not empty, then the `Paths` are extended to `Deadlock_Paths`, i.e., those paths that lead to deadlock in the initial Petri net.

6.2 Example Application of the System

In the following, we illustrate the use of the Prolog system with a "real world" example. At the same time, we show how our approach can aslo be used to correct a system that contains a deadlock state. The *gas station* system, consists of three agents working in parallel: a *customer*, an *operator* and a *pump*. The definitions of the agents are given below:

$$\begin{array}{lcl} \mathit{Customer} & \stackrel{\text{def}}{=} & \mu \mathtt{X} : (\overline{\mathtt{prepay}}.\overline{\mathtt{pumpstart}}.\mathtt{pumpfinish}.\mathtt{givechange}.\mathtt{X}) \\ \mathit{Pump} & \stackrel{\text{def}}{=} & \mu \mathtt{Y} : (\mathtt{activate}.\mathtt{pumpstart}.\overline{\mathtt{charge}}.\overline{\mathtt{pumpfinish}}.\mathtt{Y}) \\ \mathit{Operator} & \stackrel{\text{def}}{=} & \mu \mathtt{Z} : (\mathtt{prepay}.\overline{\mathtt{activate}}.\overline{\mathtt{givechange}}.\mathtt{charge}.\mathtt{Z}) \end{array}$$

Let L be the following set of actions:

$$\{\mathtt{prepay}, \mathtt{pumpstart}, \mathtt{pumpfinish}, \mathtt{givechange}, \mathtt{activate}, \mathtt{charge}\}$$

The whole system can be expressed as the composition of the above three agents after restricting them to L:

$$\mathit{Gas\_Station} \stackrel{\text{def}}{=} (\mathit{Customer} \mid \mathit{Pump} \mid \mathit{Operator}) \backslash L$$

When the above system is given as an input to the Prolog implementation we get the following output:

```
Performing Step 1:  Translating Agent into a Petri Net.
    The Transition Function of the Net is:
        t1=([p0,p1],t,[p2,p3])
```

```
        t2=([p4,p5],t,[p0,p6])
        t3=([p7,p8],t,[p4,p1])
    The Initial Marking of the Net is:
        M0=[p7,p5,p8]

Performing Step 2:  Reducing the Initial Petri Net.
    Eliminating Redundant Place:  p1
    Post-Fusing Transitions :  t2 with t1
    Pre-Fusing Transitions  :  t3 with <t2,t1>
    The Transition Function of the Reduced Net is:
        <t3,t2,t1>=
        ([p7,p5,p8],<t,t,t>,[p2,p3,p6])
    The Initial Marking of the Reduced Net is:
        M10=[p7,p5,p8]

Performing Step 3:  Computing the Reachability Graph of
the Reduced Net.
    The Reachability Graph has 2 node(s).

Performing Step 4:  Analyzing the Reachability Graph.
    The Graph contains deadlock state(s)!

Performing Step 5:  Reporting the Deadlock Paths.
    The paths that lead to Deadlocks are:
        path1=<t3,t2,t1>
```

The above results are illustrated in Figure 6.1. Clearly, a deadlock has been detected. Using the information that the system has provided, one can change the program so as to become deadlock-free. Tracing the execution of the three transitions t3, t2 and t1 one realizes that they correspond to the consecutive handshakes ($\overline{\texttt{prepay}}$, prepay), ($\overline{\texttt{activate}}$, activate) and ($\overline{\texttt{pumpstart}}$, pumpstart). After the execution of these handshakes, the system deadlocks as there exists a circular wait: the customer waits for the pump

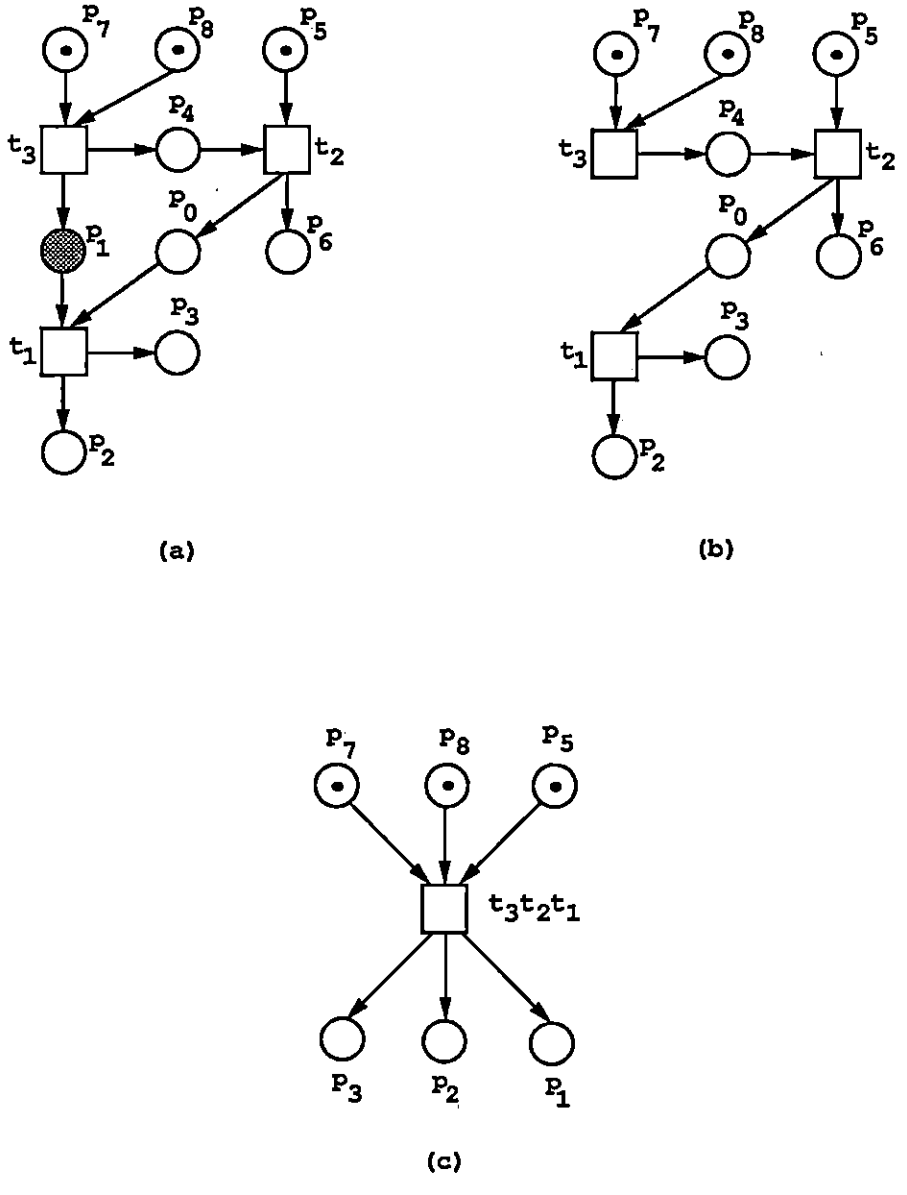


Figure 6.1: The gas station example

to finish, the pump waits for the operator to read the charging information and the operator waits for the customer to take his change. Clearly, this situation has been caused because the operator attempts to give change before actually reading the charging information. Changing the order of charge and $\overline{\texttt{givechange}}$ in the operator definition solves the problem. This can be seen by examining the new system:

```
Performing Step 1:  Translating Agent into a Petri Net.
    The Transition Function of the Net is:
        t1=([p0,p1],t,[p2,p3])
        t2=([p4,p5],t,[p0,p6])
        t3=([p7,p8],t,[p4,p1])
        t4=([p9,p10],t,[p5,p7])
        t5=([p11,p6],t,[p9,p8])
        t6=([p2,p3],t,[p10,p11])
    The Initial Marking of the Net is:
        M0=[p2,p3,p6]

Performing Step 2:  Reducing the Initial Petri Net.
    Eliminating Redundant Place:  p1
    Eliminating Redundant Place:  p5
    Eliminating Redundant Place:  p8
    Eliminating Redundant place:  p10
    Post-Fusing Transitions :  t2 with t1
    Post-Fusing Transitions :  t3 with <t2,t1>
    Post-Fusing Transitions :  t4 with <t3,t2,t1>
    Post-Fusing Transitions :  t5 with <t4,t3,t2,t1>
    Pre-Fusing Transitions  :  t6 with <t5,t4,t3,t2,t1>
    The Transition Function of the Reduced Net is:
        <t6,t5,t4,t3,t2,t1>=
        ([p2,p3,p6],<t,t,t,t,t,t>,[p2,p3,p6])
    The Initial Marking of the Reduced Net is:
        M10=[p2,p3,p6]

Performing Step 3:  Computing the Reachability Graph of
the Reduced Net.
```

```
    The Reachability Graph has 1 node(s).

Performing Step 4:  Analyzing the Reachability Graph.
    No sink nodes have been found.  The Agent does not
    contain any Deadlock states.
```

What the above example suggests, is that deadlock detection is not the only function that the system can perform: it can also be used to correct an implementation that contains deadlocks as well as to prove that an implementation is deadlock free.

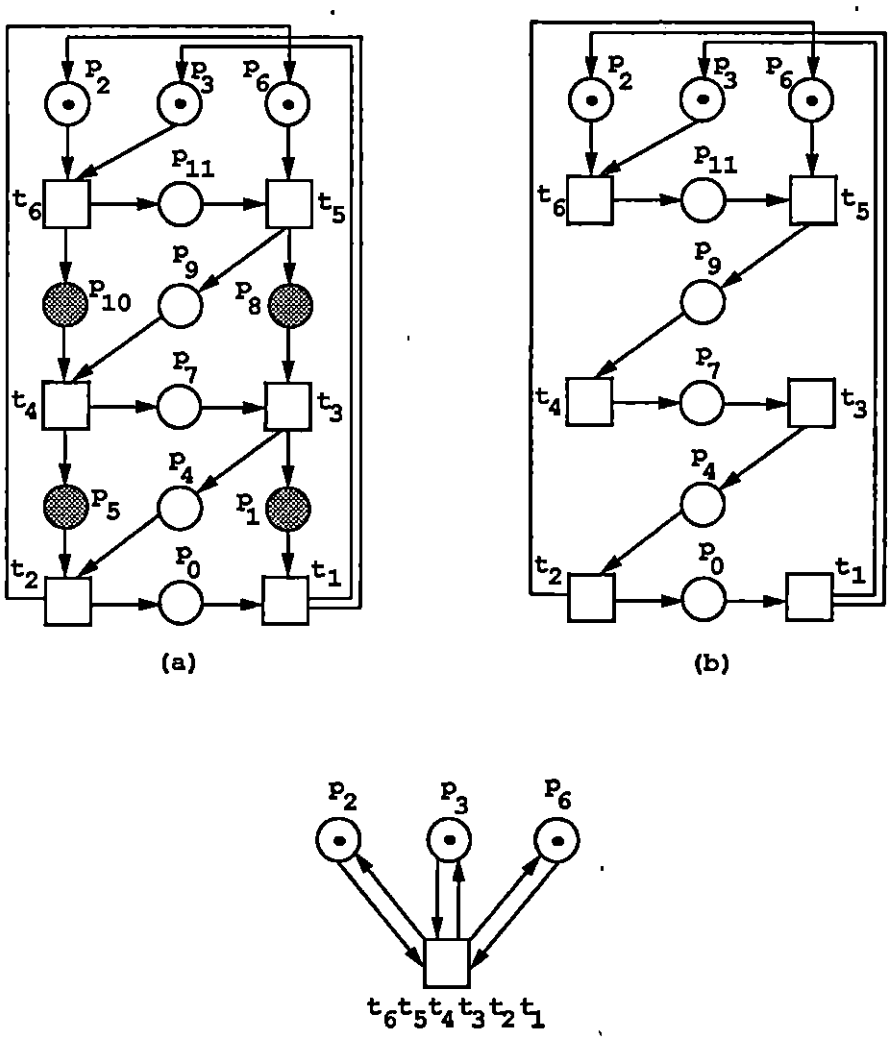


Figure 6.2: The correct system

Chapter 7

Concluding Remarks

7.1 Contributions

Reachability analysis is a widely used method for analyzing concurrent systems. Unfortunately, the number of states of even moderate systems, is often huge and unmanageable even for computers. This problem is usually described as the "state space explosion problem". To deal with state space explosion, one must effectively reduce the size of the state space, yet still be able to determine the system's properties.

In this thesis, an algorithm for deadlock detection in an abstract concurrent language has been developed. Our approach is based on efficiently reducing the state space of the programs of the language by properly transforming their corresponding Petri net representation into a simpler one. The contributions of our work can be summarized as follows:

1. It is the first – to our knowledge – practical use of the concurrent semantics of CCS. We should note here that the primary goal of representing CCS agents by Petri nets, was to transfer methods from net theory to the process algebra field [36]. However, no relevant work has

appeared until now.

2. We suggest that Prolog provides a natural way for describing the concurrent semantics of CCS. An independent approach to the same problem using Pascal, faced considerable difficulties [35]. We should also note that experimentation with a prototype implementation in Prolog, gave us some of the ideas presented in Chapter 5.

3. The proposed approach can be modified in order to apply to existing "concrete" concurrent programming languages such as Occam and ADA.

Although the results we have obtained are encouraging, there are many open questions that should be examined.

7.2 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 a set of deadlock preserving transformations *directly* on CCS agents, without transforming them into Petri nets?

3. Can we characterize a subset of CCS which gives completely reducible Petri nets using the transformations we have described? Can we characterize a subset of CCS that gives non-reducible nets? Can the answers

to these questions be used to define other more efficient transformations?

4. What other practical applications may result from the representation of a concurrent program by a Petri net?

Answers to the above questions will help us in defining more efficient methods for the analysis of concurrent programs.

Bibliography

[1] R. Milner, A calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, 1980.

[2] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

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

[4] C. A. R. Hoare, "Communicating Sequential Processes," Communications of ACM, vol. 21, pp. 666-677, 1978.

[5] J. A. Bergstra and J. W. Klop, "Algebra for Communicating Processes with Abstraction," Journal of Theoretical Computer Science, vol. 37, pp. 77-121, 1985.

[6] J. L. Peterson, Petri Net Theory and the Modelling of Systems, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

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

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

[9] T. Murata, "Petri Nets: Properties, Analysis and Applications," Proceedings of the IEEE, vol. 77, no. 4, April 1989.

[10] G. Rozenberg, Ed. Advances in Petri Nets 1984, Lecture Notes in Computer Science, vol. 188, New-York: Springer-Verlag, 1985.

[11] G. Rozenberg, Ed. Advances in Petri Nets 1985, Lecture Notes in Computer Science, vol. 222, New-York: Springer-Verlag, 1986.

[12] G. Rozenberg, Ed. Advances in Petri Nets 1987, Lecture Notes in Computer Science, vol. 266, New-York: Springer-Verlag, 1987.

[13] G. Rozenberg, Ed. Advances in Petri Nets 1988, Lecture Notes in Computer Science, vol. 340, New-York: Springer-Verlag, 1988.

[14] G. Rozenberg, Ed. Advances in Petri Nets 1989, Lecture Notes in Computer Science, vol. 424, New-York: Springer-Verlag, 1990.

[15] J. D. Noe, "A Petri net model of the CDC 6400," Proc. ACM/SIGOPS Workshop on Systems Performance Evaluation, pp. 362-378, 1971.

[16] Proc. Int. Workshop on Petri Nets and Performance Models, Madison, WI, August 24-26, 1987.

[17] G. Berthelot and R. Terrat, "Petri nets theory for the correctness of protocols," IEEE Trans. Commun., vol. COM-30, no. 12, pp. 2497-2505, Dec. 1982.

[18] 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, pp. 314-326, Mar. 1989.

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

[20] K. Lautenbach and H. A. Schmid, "Use of Petri nets for proving correctness of concurrent process systems," Proc. IFIP Congress 74, pp. 187-191, 1974.

[21] E. R. Olderog, "Operational Petri net semantics for CCSP," in: G. Rozenberg Ed., Advances in Petri Nets 1987, Lecture Notes in Computer Science, vol. 266, pp. 196-223, Springer-Verlag, 1987.

[22] E. R. Olderog, Nets, Terms and Formulas: Three Views of Concurrent Processes and their relationship, Habilitationsschrift, Univ. Kiel, 1988/89.

[23] P. Degano, R. DeNicola and U. Montanari, "CCS is an (augmented) contact-free C/E system," in: M. Venturini Zilli, Ed., Math. Models for the semantics of Parallelism, Lecture Notes in Computer Sci., vol. 280, Springer-Verlag, 1987.

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

[25] D. Taubner, Finite Representations of CCS and TCSP Programs by automata and Petri Nets, Lecture Notes in Computer Science, vol. 369, Springer-Verlag, 1989.

[26] D. Taubner, "Representing CCS Programs by Finite Predicate/ Transition Nets," Acta Informatica, 27, pp. 533-565, 1990.

[27] 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, vol. 84, pp. 277-290, New York: Springer-Verlag, 1980.

[28] G. Berthelot, "Checking properties of nets using transformations," in: G. Rozenberg, Ed., Advances in Petri Nets 1985, Lecture Notes in Computer Science, vol. 222, pp. 19-40, New York: Springer-Verlag, 1986.

[29] 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, vol. 254, pp. 359-376, Springer-Verlag, 1987.

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

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

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

[33] T. A. Chu, "A method of abstraction for Petri nets," Proc. Int. Workshop on Petri Nets and Performance Models, Madison, WI, August 24-26, 1987, pp. 164-173.

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

[35] E. R. Olderog, Personal Communication.

[36] R. DeNicola, Personal Communication.

VITA

Surname:	Rondogiannis	Given Names:	Panagiotis
Place of Birth:	Athens, Greece	Date of Birth:	31 Aug. 1966

Educational Institutions Attended:

University of Victoria	1990 to 1991
University of Patras, Greece	1984 to 1989

Degrees Awarded:

Ptihion in Comp. Eng. and Informatics	University of Patras	1989

Honours and Awards:

Greek Institute of Scholarships Award	1984 and 1985
First Scholarship of the Hellenic-Canadian Association	1990
University of Victoria Fellowship	1990-1992
Technical Chamber of Greece Scholarship	1991
Graduate Teaching Award	1991

Publications:

1. P. Rondogiannis, G. Pavlides, A. Levy, "Distributed Algorithm for Communication Deadlock Detection," Information and Software Technology, Butterworth-Heinemann Ltd., vol. 33, no. 7, pp. 483-488, Sept. 1991.
2. P. Rondogiannis, G. Pavlides, A. Levy, "SEPDS: Communication Deadlock Detection," Technical Report 90.04.11, Department of Comp. Eng. and Inform., Univ. of Patras, Greece.
3. P. Rondogiannis, "Deadlock detection in Distributed systems," Thesis for the Ptihion in Comp. Eng. and Inform., Univ. of Patras, Greece, July 1989.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Detecting Deadlocks in CCS Agents Using Petri Net Reduction Techniques

Author: 

PANAGIOTIS RONDOGIANNIS

(Name in Block Letters)

November 26, 1991

(Date)