Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

W.P. Weijland

Correctness proofs for systolic algorithms: a palindrome recognizer

Department of Computer Science Report CS-R8747 October

69B71, 69D24, 69F32

Correctness proofs for systolic algorithms: a palindrome recognizer

W.P. Weijland

Dept. of Computer Science, University of Amsterdam,

P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Abstract: In designing VLSI-circuits it is very useful, if not necessary, to construct the specific circuit by placing simple components in regular configurations. Systolic systems are circuits built up from arrays of cells and therefore very suitable for formal analysis and induction methods.
In case of a palindrome recognizer a correctness proof is given using bisimulation semantics with asynchronous cooperation. The proof is carried out in the formal setting of the *Algebra of Communicating Processes* (see [BK1]), which provides us with an algebraical theory and a convenient proof system. An extensive introduction to this theory is included in this paper.
The palindrome recognizer has also been studied by Hennessy [HEN] in a setting of failure semantics with synchronous cooperation.

key words: concurrency, process algebra, systolic array, VLSI, correctness proof.
1985 Mathematics Subject Class.: 68Q35, 68Q60, 68Q10, 68Q55.
1982 CR Categories: B.7.1, D.2.4, F.3.2

1. INTRODUCTION

In the current research on (hardware) verification one of the main goals is to find strong proof systems and tools to verify the designs of algorithms and architectures. For instance, in the development of integrated circuits ('chips') the important stage of testing a prototype (to save the high costs of producing defective processors) can be dealt with much more efficiently, when a strong verification tool is available. Therefore, developping a verification theory has very high priority and is subject of study at many universities and scientific institutions.
However, working on detailed verification theories is not the only approach to this problem. Once having a basic theory, the development of case studies is of utmost importance to provide us with new ideas. Furthermore, one can focus on special design techniques, which turn out to fit conveniently in the theory. For example, because of the regular configuration of these circuits,

Report CS-R8747
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

systolic arrays are very suitable for formal analysis and induction methods (see HENNESSY [HEN], KOSSEN & WEIJLAND [KW1,KW2], MULDER & WEIJLAND [MW], REM [RE] and WEIJLAND [WE]). Indeed, systolic arrays have grown very popular in the last few years.

In this paper we will present a theory called *Algebra of Communicating Processes* (see BERGSTRA & KLOP [BK1]), which is an algebraical theory providing us with a formal description of concurrent processes. Some of the main theoretical results are presented in simple terms, to make it possible for the reader to understand how to work in ACP.
Next, a simple description of a systolic algorithm for palindrome recognition (see KUNG [KU]) will be presented. A systolic system can be looked at as a large integration of identical cells such that the behaviour of the total system strongly resembles the behaviour of the individual cells. In fact the total system behaves like one of its individual cells 'on a larger scale'. In designing VLSI-circuits it is very useful, if not necessary, to construct the specific circuit by placing simple components in regular configurations. Otherwise, one looses all intuition about the behaviour of the circuit that is eventually constructed. For this reason one may see systolic systems as a sort of regular subclass of VLSI-circuits.
Within the semantical setting of ACP, we will be able to prove *correctness* of the palindrome recognizer. Such a proof already was presented by HENNESSY [HEN] using *synchronous* ('clocked') cooperation between cells. In the following, however, we will specify an *asynchronous* version of this algorithm. We therefore construct a *delay-insensitive* (see EBERGEN [EB]) circuit, which says that temporarily cutting some of its wires, may delay its computation but cannot endanger its correct behaviour.
This paper is a revised version of the first part of KOSSEN & WEIJLAND [KW1]. Most of the improvements only concern notational problems, although some of its formalism has been changed as well. At this place I especially want to thank Jos Baeten who took the trouble to check this paper several times before it was printed and who gave so much of his support in developing its contents.

2. THE ALGEBRA OF COMMUNICATING PROCESSES

The axiomatic framework in which we present this document is ACP_τ, the Algebra of Communicating Processes with silent steps, as described in BERGSTRA & KLOP [BK2]. In this section, we give a brief review of ACP_τ.
Process algebra starts from a finite collection A of given objects, called atomic actions, atoms or steps. These actions are taken to be indivisible, usually have no duration and form the basic

building blocks of our systems. The first two compositional operators we consider are $\cdot$, denoting sequential composition, and + for alternative composition. If x and y are two processes, then $x{\cdot}y$ is the process that starts the execution of y after the completion of x, and $x+y$ is the process that chooses either x or y and executes the chosen process. Each time a choice is made, we choose from a set of alternatives. We do not specify whether the choice is made by the process itself, or by the environment. Axioms A1-5 in table 1 below give the laws that + and $\cdot$ obey. We leave out $\cdot$ and brackets as in regular algebra, so $xy + z$ means $(x{\cdot}y) + z$.
On intuitive grounds $x(y + z)$ and $xy + xz$ present different mechanisms (the moment of choice is different), and therefore, an axiom $x(y + z) = xy + xz$ is not included (see example below).

example
Playing Russian Roulette can best be described by the process *shot·dead +shot·alive*, instead of *shot·(dead+alive)*, since in the second process after the shot, one can still chose between dying and surviving. This illustrates why $x(y + z) = xy + xz$ is not included as an axiom.

We have a special constant δ denoting deadlock, the acknowledgement of a process that it cannot do anything anymore, the absence of an alternative. Axioms A6,7 give the laws for δ. Together, the axioms A1-A7 are referred to as *BPA*, which stands for *Basic Process Algebra.*
Next, we have the parallel composition operator $\|$, called merge. The merge of processes x and y will interleave the actions of x and y, except for the communication actions. In $x\|y$, we can either do a step from x, or a step from y, or x and y both synchronously perform an action, which together make up a new action, the communication action. This trichotomy is expressed in axiom CM1. Here, we use two auxiliary operators $\Lsh$ (left-merge) and $|$ (communication merge). Thus, $x \Lsh y$ is $x\|y$, but with the restriction that the first step comes from x, and $x|y$ is $x\|y$ with a communication step as the first step. Axioms CM2-9 give the laws for $\Lsh$ and $|$. On atomic actions, we assume the communication function given, obeying laws C1-3.

example
$a\|b \ = \ a \Lsh b + b \Lsh a \ + \ (a|b) \ = \ ab + ba + (a|b)$
$(ab) \Lsh c \ = \ a(b\|c) \ = \ a(bc + cb + (b|c))$
$(ab)|(cd) \ = \ (a|c)(b\|d) \ = \ (a|c)(bd + db + (b|d)).$

Finally, on the left-hand side of table 1 we have the laws for the encapsulation operator ∂_H. Here H is a set of atoms, and ∂_H blocks actions from H, renames them into δ. The operator ∂_H can be used to encapsulate a process, i.e. to block communications with the environment.

example
Suppose H={b}, then $\partial_H(a \| b) = \partial_H(ab + ba + (a|b)) = a\delta + \delta a + (a|b)$ and using axioms A6 and A7, we obtain: $= a\delta + \delta + (a|b) = a\delta + (a|b)$.

In all following tables we have $a,b,c \in A \cup \{\delta\}$, x,y,z are arbitrary processes, and $H,I \subseteq A$.

$x + y = y + x$	A1	$x \| y = x \llcorner\!\llcorner y + y \llcorner\!\llcorner x + x \mid y$	CM1
$x + (y + z) = (x + y) + z$	A2	$a \llcorner\!\llcorner x = ax$	CM2
$x + x = x$	A3	$ax \llcorner\!\llcorner y = a(x \| y)$	CM3
$(x + y)z = xz + yz$	A4	$(x + y) \llcorner\!\llcorner z = x \llcorner\!\llcorner z + y \llcorner\!\llcorner z$	CM4
$(xy)z = x(yz)$	A5	$ax \mid b = (a \mid b)x$	CM5
$x + \delta = x$	A6	$a \mid bx = (a \mid b)x$	CM6
$\delta x = \delta$	A7	$ax \mid by = (a \mid b)(x \| y)$	CM7
		$(x + y) \mid z = x \mid z + y \mid z$	CM8
		$x \mid (y + z) = x \mid y + x \mid z$	CM9
$\partial_H(a) = a \quad \text{if } a \notin H$	D1		
$\partial_H(a) = \delta \quad \text{if } a \in H$	D2	$a \mid b = b \mid a$	C1
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3	$(a \mid b) \mid c = a \mid (b \mid c)$	C2
$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	D4	$\delta \mid a = \delta$	C3

table 1. ACP.

Next, we introduce laws for Milner's silent step, denoted by the new constant τ (see MILNER [MI]). This constant can be looked at as an *internal* action, which cannot be seen from the outside. In fact, τ stands for zero or more machine steps and indicates that the machine is busy.
Suppose we see atomic actions only *start*, not end. The silent step τ denotes an invisible action, so we do not see it start, nor end. Now, it is clear that the processes a and $a\tau$ cannot be distinguished, since they both start with a and after a while they terminate. Thus, in general: $x\tau = x$.
Since τ stands for *zero or more* internal machine steps, any process τx has a possibility of starting immediately with x. So, since τx has a summand x, we have: $\tau x + x = \tau x$.

For similar reasons $a(\tau x + y)$ has a summand ax, since after this process has done a, and possibly some internal moves, it might do x without being able to choose y. So: $a(\tau x + y) = a(\tau x + y) + ax$. In table 2, the laws T1-3 are Milner's τ-laws, and TM1,2 and TC1-4 describe the interaction of τ and merge. Finally, τ_I is the abstraction operator, that renames atoms from I into τ.

$x\tau = x$	T1		
$\tau x + x = \tau x$	T2		
$a(\tau x + y) = a(\tau x + y) + ax$	T3		
$\tau \mathbin{\lfloor\!\lfloor} x = \tau x$	TM1	$\partial_H(\tau) = \tau$	DT
$\tau x \mathbin{\lfloor\!\lfloor} y = \tau(x \| y)$	TM2	$\tau_I(\tau) = \tau$	TI1
$\tau \mid x = \delta$	TC1	$\tau_I(a) = a$ if $a \notin I$	TI2
$x \mid \tau = \delta$	TC2	$\tau_I(a) = \tau$ if $a \in I$	TI3
$\tau x \mid y = x \mid y$	TC3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI4
$x \mid \tau y = x \mid y$	TC4	$\tau_I(xy) = \tau_I(x) \cdot \tau_I(y)$	TI5

table 2. The silent step τ.

The axioms of ACP in table 1, together with the axioms in table 2 above, form the system ACP_τ.

definition The set of **basic terms**, BT, is inductively defined as follows:

i. $\tau, \delta \in BT$

ii. if $t \in BT$, then $\tau t \in BT$

iii. if $t \in BT$ and $a \in A$, then $at \in BT$

iv. if $t, s \in BT$, then $t+s \in BT$.

elimination theorem (BERGSTRA & KLOP [BK2]) Let t be a closed term over ACP_τ. Then there is a basic term s such that $ACP_\tau \vdash t=s$.

The elimination theorem allows us to use induction in proofs. The set of closed terms modulo derivability (the initial algebra) forms a model for ACP_τ. However, most processes encountered in practice cannot be represented by a closed term, but will be specified recursively. Therefore, most models of process algebra also contain infinite processes, that can be recursively specified. First, we develop some terminology.

definition i) Let t be a term over ACP_τ, and x a variable in t. Suppose that the abstraction operator τ_I does not occur in t. Then we say that an occurrence of x in t is **guarded** if t has a subterm of the form a·s, with $a \in A_\delta$ (so $a \neq \tau$!) and this x occurs in s. (I.e. each variable is 'preceded' by an atom.)

ii) A **recursive specification** over ACP_τ is a set of equations $\{x = t_x : x \in X\}$, with X a set of variables, and t_x a term over ACP_τ and variables X (for each $x \in X$). No other variables may occur in t_x.

iii) A recursive specification $\{x = t_x : x \in X\}$ is **guarded** if no t_x contains an abstraction operator τ_I, and each occurrence of a variable in each t_x is guarded.

notes: i) The constant τ cannot be a guard, since the presence of a τ does not lead to unique solutions: for instance, the equation $x = \tau x$ has each process starting with a τ as a solution.

ii) A definition of guardedness involving τ_I is very complicated, and therefore, we do not give such a definition here. The definition above suffices for our purposes.

definition: On ACP_τ, we can define a **projection operator** π_n, that cuts off a process after n atomic steps are executed, by the axioms in table 2 ($n \geq 1$, $a \in A_\delta$, x,y are arbitrary processes).

$\pi_n(a) = a$	$\pi_n(\tau) = \tau$
$\pi_1(ax) = a$	$\pi_n(\tau x) = \tau \cdot \pi_n(x)$
$\pi_{n+1}(ax) = a \cdot \pi_n(x)$	
$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	

table 3. Projection.

remarks: Because of the τ-laws, we must have that executing a τ does not increase depth. A process p is **finite** if it is equal to a closed term; otherwise p is **infinite**. Note that if p is finite, there is an n such that $\pi_n(p) = p$.

projection theorem (BAETEN, BERGSTRA & KLOP [BBK2]) If the set of processes P forms a solution for a guarded recursive specification E, then $\pi_n(p)$ is equal to some closed ACP_τ-term for each $p \in P$ and $n \geq 1$, and this term does not depend on the particular solution P.

The projection theorem leads us to formulate the following two principles, which together imply that each guarded recursive specification has a unique solution (determined by its finite projections).

The **Recursive Definition Principle (RDP)** is the assumption that each guarded recursive specification has at least one solution, and the **Recursive Specification Principle (RSP)** is the assumption that each guarded recursive specification has at most one solution. In this paper, we assume RDP and RSP to be valid.

To give an example, if p is a solution of the guarded recursive specification $\{x = a\cdot x\}$, we find $\pi_n(p) = a^n$ for all $n \geq 1$, so we can put $p = a^\omega$. For more information, see [BBK1]. Abusing language, we also use the variables in a guarded recursive specification for the process that is its unique solution.

In BAETEN, BERGSTRA & KLOP [BBK1], a model is presented for ACP_τ, consisting of rooted, directed multigraphs, with edges labeled by elements of $A \cup \{\delta,\tau\}$, modulo a congruence relation called rooted $\tau\delta$-bisimulation (comparable to Milner's observational congruence, see [MI]). In this model all axioms presented in this paper hold, and also principles RDP and RSP hold.

The axioms of **Standard Concurrency** (displayed in table 4) will also be used in the sequel. A proof that they hold for all closed terms can be found in BERGSTRA & KLOP [BK2].

$$(x \mathbin{\|\!\!\_} y) \mathbin{\|\!\!\_} z = x \mathbin{\|\!\!\_} (y \| z)$$
$$(x \mid ay) \mathbin{\|\!\!\_} z = x \mid (ay \mathbin{\|\!\!\_} z)$$
$$x \mid y = y \mid x$$
$$x \| y = y \| x$$
$$x \mid (y \mid z) = (x \mid y) \mid z$$
$$x \| (y \| z) = (x \| y) \| z$$

table 4. Standard concurrency.

As one can easily see encapsulation and abstraction cannot in general be distributed over $\|$ since in a merge processes may do a communication step and thus it is of great importance which comes first, the encapsulation (or abstraction) operator or the merge. Next *conditional axioms* will be presented to state conditions for distributing τ_I and ∂_H over $\|$.

Definition: The **alphabet** of a process is the set of atomic actions that it can perform. So an alphabet is a subset of A. In order to define the alphabet function α on processes, we have the axioms in table 5 (for $a \in A$, x,y are arbitrary processes; see BAETEN, BERGSTRA & KLOP [BBK2]).

$\alpha(\delta) = \emptyset$	AB1
$\alpha(\tau) = \emptyset$	AB2
$\alpha(ax) = \{a\} \cup \alpha(x)$	AB3
$\alpha(\tau x) = \alpha(x)$	AB4
$\alpha(x + y) = \alpha(x) \cup \alpha(y)$	AB5
$\alpha(x) = \cup_{n\geq 1} \alpha(\pi_n(x))$	AB6
$\alpha(\tau_I(x)) = \alpha(x) - I$	AB7

table 5. Alphabet.

Note that $\alpha(\delta) = \alpha(\tau) = \emptyset$ is necessary by axioms A6 and T1. The axioms AB6 and AB7 can be proved from AB1-5 for *closed* terms, but are needed here to define the alphabet on general processes.
Now we can formulate the conditional axioms as is done in table 6.

$\alpha(x) \mid (\alpha(y) \cap H) \subseteq H$	$\Rightarrow$	$\partial_H(x \| y) = \partial_H(x \| \partial_H(y))$	CA1
$\alpha(x) \mid (\alpha(y) \cap I) = \emptyset$	$\Rightarrow$	$\tau_I(x \| y) = \tau_I(x \| \tau_I(y))$	CA2
$\alpha(x) \cap H = \emptyset$	$\Rightarrow$	$\partial_H(x) = x$	CA3
$\alpha(x) \cap I = \emptyset$	$\Rightarrow$	$\tau_I(x) = x$	CA4
$H = J \cup K$	$\Rightarrow$	$\partial_H(x) = \partial_J \circ \partial_K(x)$	CA5
$I = J \cup K$	$\Rightarrow$	$\tau_I(x) = \tau_J \circ \tau_K(x)$	CA6
$H \cap I = \emptyset$	$\Rightarrow$	$\tau_I \circ \partial_H(x) = \partial_H \circ \tau_I(x)$	CA7

table 6. Conditional axioms.

In [BBK2] the axioms CA1-7 have been proved to hold for all closed ACP_τ-terms. We will assume that they hold for all processes.

3. A PALINDROME-RECOGNIZER

In the following we will describe a machine which is able to recognize palindromes from strings of input symbols i.e. a machine that answers 'true' iff a given string of inputsymbols is equal to its reverse.

Suppose S is a finite set of symbols from which the input strings are built up.
The actions of sending and receiving a symbol d along some channel are written as s(d) and r(d) respectively. Moreover we have a predicate *ispal* with strings of symbols as its domain which is true iff its argument is a palindrome. Finally we write |w| for the length of the string w.
Now we can easily write down the specification of the palindrome-recognizer PAL as is done in table 7.

$$\begin{aligned} PAL(\varepsilon) &= s(true)\cdot PAL(\varepsilon) + \Sigma_{x\in S}\, r(x)\cdot s(true)\cdot PAL(x) \\ PAL(w) &= \Sigma_{x\in S}\, r(x)\cdot s(ispal(x\cdot w))\cdot PAL(x\cdot w) \qquad (\,|w| > 0\,) \end{aligned}$$

table 7. A specification of the palindrome-recognizer PAL.

The specification in table 7 describes precisely our intuition about what a palindrome-recognizer should do.
Note that the machine PAL only *receives* inputsymbols. Since it is clear that a palindrome-recognizer should not throw away any of its received information the machine described in table 7 needs to be able to contain arbitrarily long strings of symbols. In practice, however, machines are of a finite size. So from a more practical point of view we should give a specification of a machine that only works on input strings with a limited length.
This is done in table 8 where a finite machine PAL_k is specified, working exactly like the previous palindrome-recognizer but now with a limit to the length of its input. For reasons to be explained later this limit is put at length 2k instead of k.
We assume the machine PAL_k to have an in/output channel numbered k+1. So $s_{k+1}(d)$ and $r_{k+1}(d)$ will denote the actions of sending and receiving a symbol d.

The fourth equation says that if PAL_k has reached its maximum capacity it will deadlock, for instance by announcing memory-overflow.

$PAL_0(w)$	$=$	$s_1(true) \cdot PAL_0(w)$	$(0 \leq |w|)$
$PAL_{k+1}(\varepsilon)$	$=$	$s_{k+2}(true) \cdot PAL_{k+1}(\varepsilon) + \Sigma_{x \in S}\, r_{k+2}(x) \cdot s_{k+2}(true) \cdot PAL_{k+1}(x)$	
$PAL_{k+1}(w)$	$=$	$\Sigma_{x \in S}\, r_{k+2}(x) \cdot s_{k+2}(ispal(x \cdot w)) \cdot PAL_{k+1}(x \cdot w)$	$(0 < |w| < 2(k+1))$
$PAL_{k+1}(w)$	$=$	δ	$(2(k+1) \leq |w|)$

table 8. A specification of PAL_k for arbitrary natural number k.

We will now introduce an implementation of a Palindrome recognizer. This particular implementation has the look of a large chain of cells, each of which is itself a Palindrome recognizer of size 2. We will prove that a merge of k such cells gives us exactly a Palindrome-recognizer of size 2k, i.e.: satifies the specification in table 8.
Consider the cell pictured in figure 1.

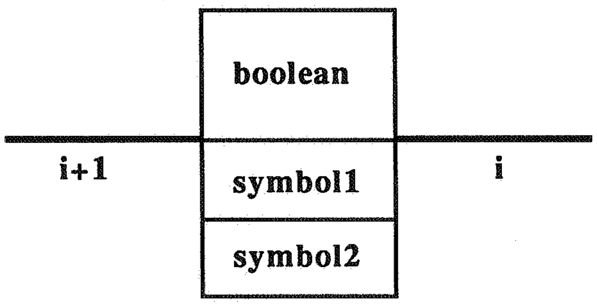

figure 1. An individual cell, C_i, of the palindrome-recognizer

The ith cell C_i has two communication channels **i** and **i+1**. Internally C_i has three storage locations, one for boolean values and two for symbols.
The cell C_i has three distinct states.

(0) In the initial state the cell carries no symbols, i.e.: carries the empty word, and since the empty word is a palindrome it can always output the boolean value true to the left. If a symbol is input from the left it is stored in the location **symbol2**, then the boolean value true is output to the left since a word consisting of a single symbol always is a palindrome. The cell now is in state one.

(1) In state one a symbol is input from the left and a boolean from the right (in any order), and stored in the remaining locations **symbol1** and **boolean.** The cell is now in state two.

(2) In state two the cell contains two symbols **symbol1** and **symbol2** forming a word that is a palindrome iff **symbol1=symbol2.** Now a boolean value **b** is output to the left, which is calculated according to the formula

$$\mathbf{b} \Leftrightarrow \mathbf{boolean}\ \mathit{and}\ \mathbf{symbol1{=}symbol2}\,.$$

Hence before deciding about its output the cell C_i *consults* messages received from its channels. Together with this boolean output the symbol in location **symbol1** is output to the right (in any order interleaved) making room for new input symbols. The cell is now in state one once more.

In the language of ACP_τ the behaviour of the cell C_i described above can be expressed by the equations shown in table 9. The fourth equation defines a machine called TC which stands for *terminal cell.* Since TC never 'contains' any symbol (or always contains the empty string) it can always output a boolean value true and thus behaves like a palindrome-recognizer of size zero (note that the empty string is a palindrome).

$$C_i = s_{i+1}(\text{true})\cdot C_i + \Sigma_{x\in S}\, r_{i+1}(x)\cdot s_{i+1}(\text{true})\cdot C'_i(x)$$

$$C'_i(x) = [\{\Sigma_{y\in S}\, r_{i+1}(y)\} \| \{\Sigma_{v\in\{\text{true,false}\}}\, r_i(v)\}]\cdot C''_i(x,y,v)$$

$$C''_i(x,y,v) = [s_{i+1}(x{=}y\ \mathit{and}\ v) \| s_i(y)]\cdot C'_i(x)$$

$$TC = s_1(\text{true})\cdot TC$$

table 9. Formal definition of the behaviour of an individual cell.

Note that the second equation violates the scope rules of Σ since y and v are bounded variables in the first term. We will nevertheless use this notation as a shorthand for the correct but much more complex term

$$\Sigma_{y\in S}\, r_{i+1}(y)\cdot[\Sigma_{v\in\{\text{true,false}\}}\, r_i(v)\cdot C''_i(x,y,v)] + \Sigma_{v\in\{\text{true,false}\}}\, r_i(v)\cdot[\Sigma_{y\in S}\, r_{i+1}(y)\cdot C''_i(x,y,v)].$$

We prefer not to introduce a formal notion here.

From the cells described above we now construct a larger machine by putting the cells in a chain and defining communications between connected cells. Consider the configuration as pictured in figure 2 below. The cells C_{i-1} and C_i can communicate through channel **i** by the communication action $s_i(x) \mid r_i(x)$. Any separate action $s_i(x)$ or $r_i(x)$ will be encapsulated, except for $s_{k+1}(x)$ and $r_{k+1}(x)$, since there is no cell C_{k+1} to communicate with them. Hence these two actions can communicate with the outside world.

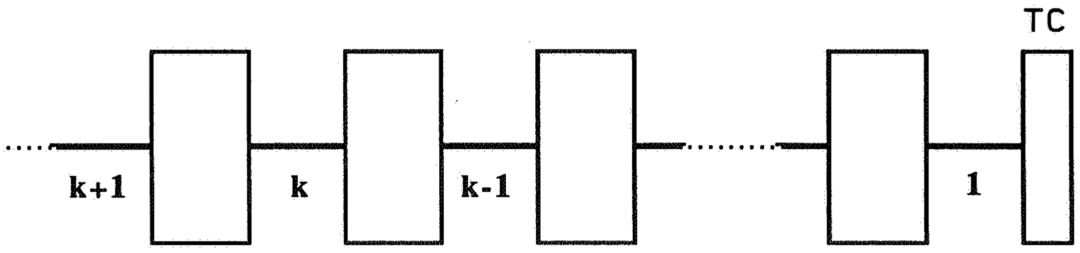

figure 2. A chain configuration of k cells.

From now we assume k to be fixed.

We have the following *communication function* defined on atomic actions:

$s_i(x) \mid r_i(x)$	$= c_i(x)$	for all $x \in S$ and $i < k+1$
$a \mid b$	$= \delta$	for all other pairs of actions $a,b \in A$.

The *encapsulation set* H_k of actions resulting in a deadlock is defined as

$$H_k = \{s_i(x), r_i(x) : x \in S \text{ and } i < k+1\}$$

The *abstraction set* I of internal communication actions is defined as

$$I = \{c_i(x) : x \in S \text{ and } i < \omega\}.$$

Note that none of the actions from I occur in the specification of table 8. One can look at them as actions that are *invisible* or *hidden*, and cannot be influenced from outside.

The machine pictured in figure 2 can algebraically be described as a communication merge M(k) of k individual cells i.e:

$$M(k) = \tau_I\, \partial_{H_k}(C_k \| \dots \| C_1 \| TC).$$

In the following we will formally prove that M(k) indeed is an implementation of the palindrome-recognizer given in table 8.

4. A FORMAL PROOF OF CORRECTNESS

Before turning to the formal proof itself let us first try an example to see how the machine works. Indeed this gives us some intuition about the practical behaviour of M(k) which will be helpful later in this paragraph. The specific example given below was found in [HEN].

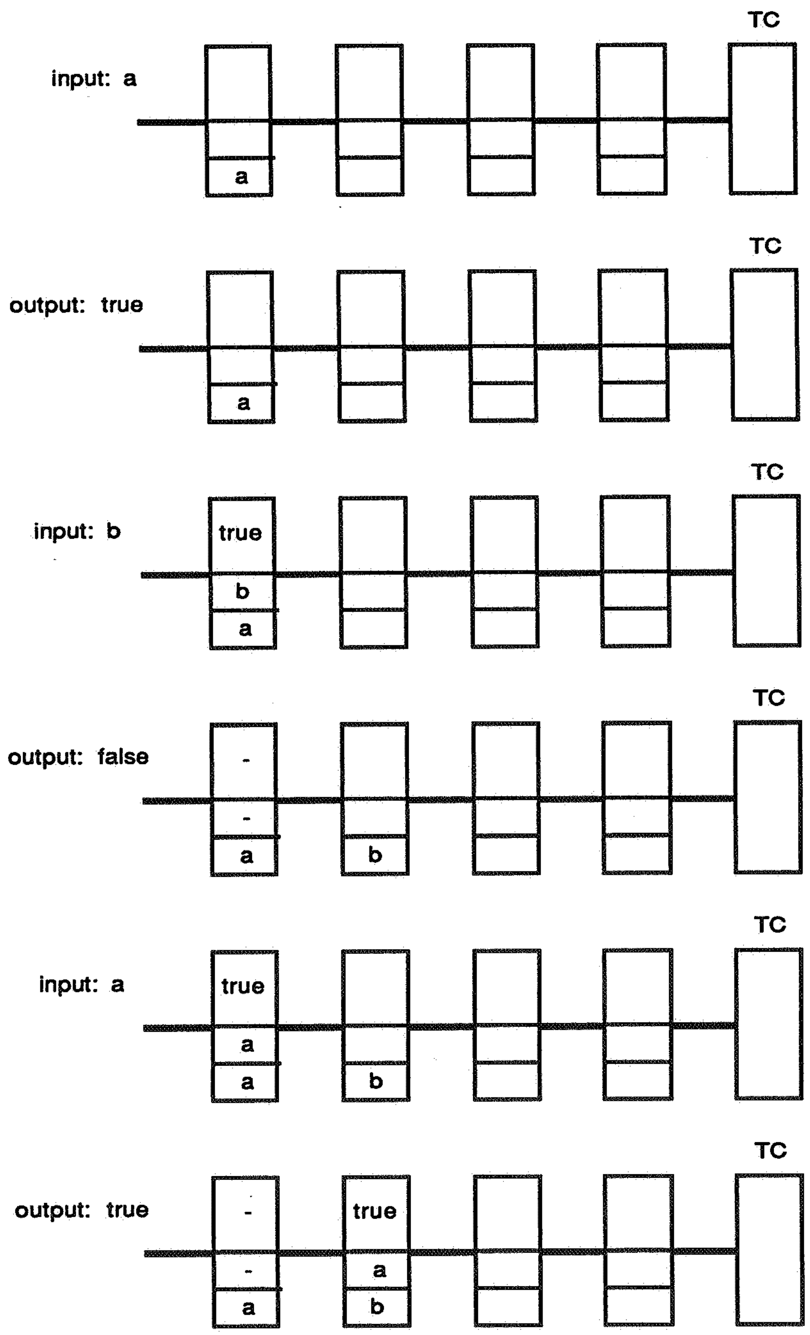

figure 3. An example of the machine M(4).

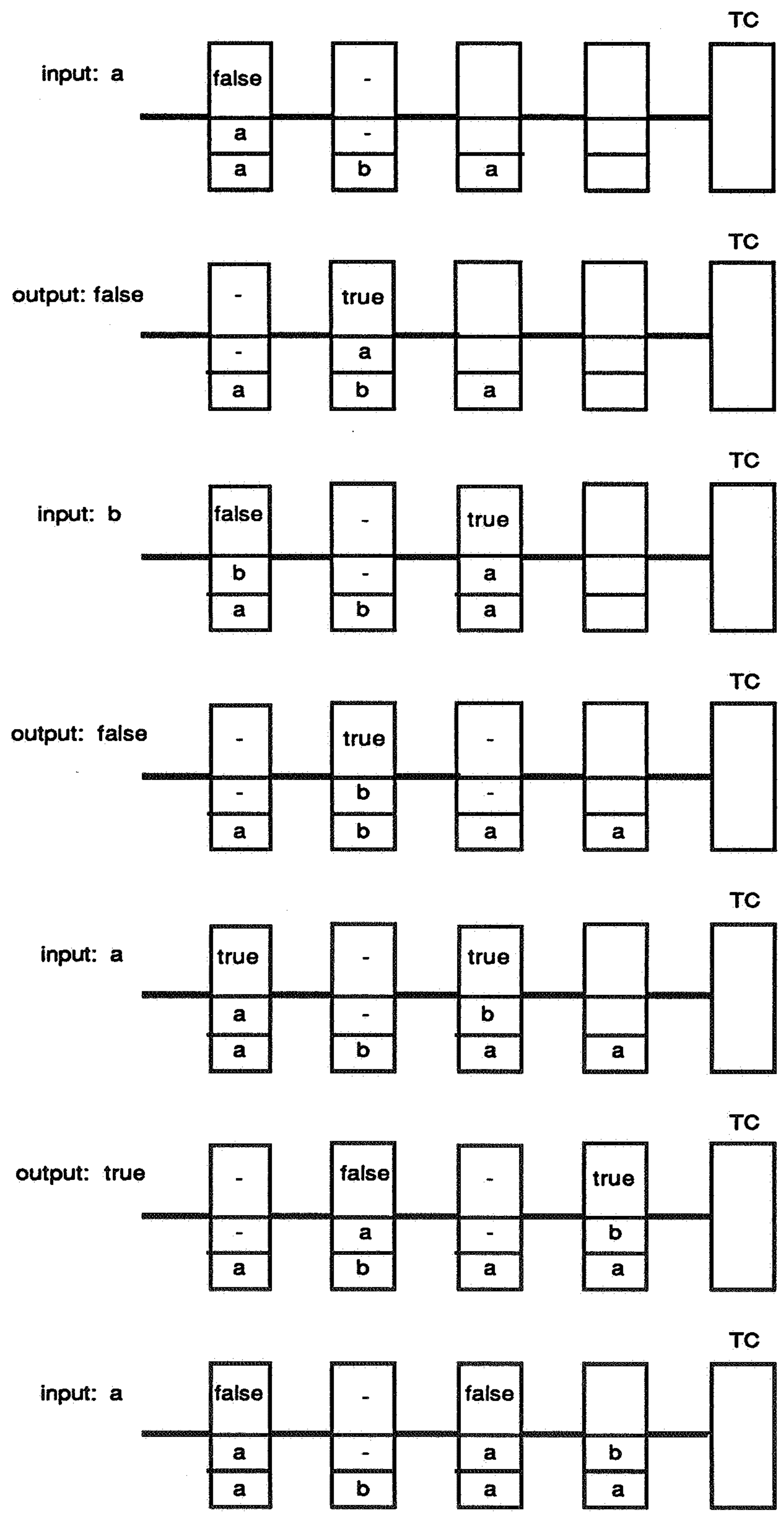

figure 3 (continued).

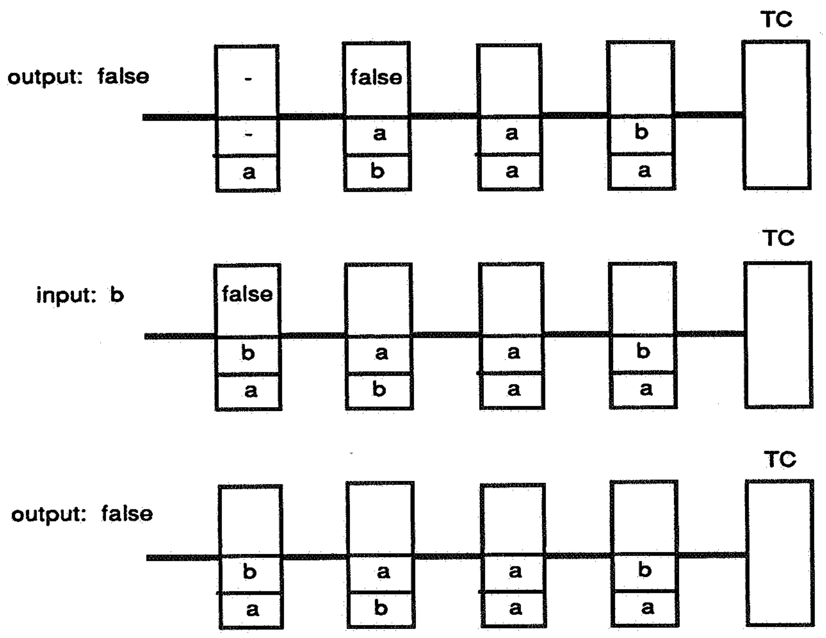

figure 3 (continued).

In figure 3 four connected cells are pictured and we can the machine respond at the inputstring *abaabaab*. As we see, immediately after receiving a new input symbol the machine returns a boolean value at the leftmost channel, stating whether or not the string input so far is a palindrome.

We will now get to the correctness theorem which will be proved by means of the equations of ACP_τ together with RSP, the Recursive Specification Principle which says that if two processes satisfy the same guarded recursive specification then they are equal.

theorem $M(k) = \tau_I \partial_{H_k}(C_k \| \ldots \| C_1 \| TC) = PAL_k(\varepsilon)$.

proof by induction on k.
k=0: $M(0) = \tau_I \partial_{H_0}(TC) = TC$, and using RSP we directly find $TC = PAL_0(\varepsilon)$.
k+1: we first prove $\tau_I \partial_{H_{k+1}}(C_{k+1} \| PAL_k(\varepsilon)) = PAL_{k+1}(\varepsilon)$. It is easily checked that the following two equations hold:

(1) $\tau_I \partial_{H_{k+1}}(C_{k+1} \| PAL_k(\varepsilon)) = s_{k+2}(true) \cdot \tau_I \partial_{H_{k+1}}(C_{k+1} \| PAL_k(\varepsilon)) +$
$+ \sum_{x \in S} r_{k+2}(x) \cdot s_{k+2}(true) \cdot \tau_I \partial_{H_{k+1}}(C'_{k+1}(x) \| PAL_k(\varepsilon))$.

(2) $\tau_I \partial_{H_{k+1}}(C'_{k+1}(x) \| PAL_k(\varepsilon)) = \tau \cdot \sum_{y \in S} r_{k+2}(y) \cdot \tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,true) \| PAL_k(\varepsilon))$.

As a lemma, we formulate what is in fact the crucial induction hypothesis:

lemma

for all symbols $x,y \in S$ and strings $v \in S^*$ with $|v| \leq 2k$, we have

(i) $\tau_I \partial_{H_{k+1}}(C'_{k+1}(x) \| \{s_{k+1}(ispal(v)) \cdot PAL_k(v)\}) = \tau \cdot PAL_{k+1}(v \cdot x)$

(ii) $\tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,ispal(v)) \| PAL_k(v)) = \tau \cdot s_{k+1}(ispal(y \cdot v \cdot x))) \cdot PAL_{k+1}(y \cdot v \cdot x)$.

proof:

Define: $Q(v,x) = \tau_I \partial_{H_{k+1}}(C'_{k+1}(x) \| \{s_{k+1}(ispal(v)) \cdot PAL_k(v)\})$ if $|v| \leq 2k$

$Q(v,x) = \delta$ if $|v| > 2k$.

Now we prove that for all $|v| \leq 2k$ (or equivalently: $|vx| < 2(k+1)$), we have

$$Q(v,x) = \tau \cdot \Sigma_{y \in S}\, r_{k+2}(y) \cdot s_{k+2}(ispal(y \cdot v \cdot x))) \cdot Q(y \cdot v,x)$$

and hence, by RSP and the specification of PAL_k in table 8:

$$Q(v,x) = \tau \cdot PAL_{k+1}(v \cdot x).$$

For all $|v| \leq 2k$ we have:

$Q(v,x) =$

$= \tau_I \partial_{H_{k+1}}(\{[\Sigma_{y \in S}\, r_{k+2}(y) \| \Sigma_{b \in \{true,false\}}\, r_{k+1}(b)] \cdot C''_{k+1}(x,y,b)\} \| \{s_{k+1}(ispal(v)) \cdot PAL_k(v)\}))$

$= \Sigma_{y \in S}\, r_{k+2}(y) \cdot \tau_I \partial_{H_{k+1}}(\{\Sigma_{b \in \{true,false\}}\, r_{k+1}(b) \cdot C''_{k+1}(x,y,b)\} \| \{s_{k+1}(ispal(v)) \cdot PAL_k(v)\})$

$+ \tau \cdot \tau_I \partial_{H_{k+1}}(\{\Sigma_{y \in S}\, r_{k+2}(y) \cdot C''_{k+1}(x,y,ispal(v))\} \| PAL_k(v))$

$= \tau \cdot \Sigma_{y \in S}\, r_{k+2}(y) \cdot \tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,ispal(v)) \| PAL_k(v))$ (using axiom T2).

(i) Suppose $|v| < 2k$, then we find

$\tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,ispal(v)) \| PAL_k(v))$

$= s_{k+2}(x{=}y \text{ } \mathit{and} \text{ } ispal(v)) \cdot$

$\tau_I \partial_{H_{k+1}}(\{s_{k+1}(y) \cdot C'_{k+1}(x)\} \| \{\Sigma_{z \in S}\, r_{k+1}(z) \cdot s_{k+1}(ispal(z \cdot v))) \cdot PAL_k(z \cdot v)\})$

$+ \tau \cdot \tau_I \partial_{H_{k+1}}(\{s_{k+1}(x{=}y \text{ } \mathit{and} \text{ } ispal(v)) \cdot C'_{k+1}(x)\} \| \{s_{k+1}(ispal(y \cdot v))) \cdot PAL_k(y \cdot v)\})$

$= \tau \cdot s_{k+2}(x{=}y \text{ } \mathit{and} \text{ } ispal(v)) \cdot \tau_I \partial_{H_{k+1}}(C'_{k+1}(x) \| \{s_{k+1}(ispal(y \cdot v))) \cdot PAL_k(y \cdot v)\})$

$= \tau \cdot s_{k+2}(x{=}y \text{ } \mathit{and} \text{ } ispal(v)) \cdot Q(y \cdot v,x)$,

since $|y \cdot v| \leq 2k$.

(ii) Suppose $|v| = 2k$, then we have

$$\tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,ispal(v)) \| PAL_k(v)) = \tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,ispal(v)) \| \delta)$$

$$= \tau \cdot s_{k+2}(x{=}y \textit{ and } ispal(v)) \cdot \delta$$

$$= \tau \cdot s_{k+2}(x{=}y \textit{ and } ispal(v)) \cdot Q(y \cdot v, x),$$

since $|y \cdot v| > 2k$.

Since $x{=}y$ *and* $ispal(v) \Leftrightarrow ispal(y \cdot v \cdot x))$

we have for all $|v| \leq 2k$:

$$\tau_I \partial_{H_{k+1}}(C''_{k+1}(x,y,ispal(v)) \| PAL_k(v)) = \tau \cdot s_{k+1}(ispal(y \cdot v \cdot x))) \cdot Q(y \cdot v, x).$$

After substitution we find

$$Q(v,x) = \tau \cdot \Sigma_{y \in S}\, r_{k+1}(y) \cdot s_{k+1}(ispal(y \cdot v \cdot x))) \cdot Q(y \cdot v, x)$$

which is precisely what we wanted.
For $|v| > 2k$ we directly find

$$Q(v,x) = \delta = PAL_{k+1}(v \cdot x)$$

By RSP we have lemma (i). Note that we implicitly proved (ii). **end proof.**

Using the lemma, the proof of the theorem is easy:
With lemma (ii) and (2) we have

$$\tau_I \partial_{H_{k+1}}(C'_{k+1}(x) \| PAL_k(\varepsilon)) =$$
$$= \tau \cdot \Sigma_{y \in S}\, r_{k+2}(y) \cdot s_{k+2}(ispal(y \cdot x))) \cdot PAL_{k+1}(y \cdot x)$$
$$= \tau \cdot PAL_{k+1}(x).$$

Finally with (1) we have

$$\tau_I \partial_{H_{k+1}}(C_{k+1} \| PAL_k(\varepsilon)) =$$
$$= s_{k+2}(true) \cdot \tau_I \partial_{H_{k+1}}(C_{k+1} \| PAL_k(\varepsilon)) + \Sigma_{x \in S}\, r_{k+2}(x) \cdot s_{k+2}(true) \cdot \tau \cdot PAL_{k+1}(x)$$
$$= PAL_{k+1}(\varepsilon)$$

using RSP again.

So we have

$$\tau_I \partial_{H_{k+1}}(C_{k+1} \| \tau_I \partial_{H_k}(C_k \| \ldots \tau_I \partial_{H_1}(C_1 \| TC) \ldots)) = PAL_{k+1}(\varepsilon).$$

It is easy to prove by induction, however, that

$\alpha(C_{k+1}) \mid \{\alpha(C_k \| M(k-1)) \cap H_k\} = \emptyset$, and

$\alpha(C_{k+1}) \mid \{\alpha(C_k \| M(k-1)) \cap I\} = \emptyset.$

So because $H_{k+1} \supseteq H_k$, using the conditional axioms CA1, CA2 and CA5 we find:

$$\tau_I \partial_{H_{k+1}}(C_{k+1} \| \tau_I \partial_{H_k}(C_k \| \dots \tau_I \partial_{H_1}(C_1 \| TC)\dots)) =$$
$$\tau_I \partial_{H_{k+1}}(\tau_I \partial_{H_k}(\dots \tau_I \partial_{H_1}(C_{k+1} \| C_k \| \dots \| C_1 \| TC)\dots)).$$

Since

$$H \cap I = \emptyset$$

we have

$$\tau_I \partial_{H_{k+1}}(\tau_I \partial_{H_k}(\dots \tau_I \partial_{H_1}(C_{k+1} \| C_k \| \dots \| C_1 \| TC)\dots) =$$
$$\tau_I \dots \tau_I \partial_{H_{k+1}} \dots \partial_{H_1}(C_{k+1} \| C_k \| \dots \| C_1 \| TC)$$

by axiom CA7 and finally with axioms CA5 and CA6 we find

$$\tau_I \dots \tau_I \partial_{H_{k+1}} \dots \partial_{H_1}(C_{k+1} \| C_k \| \dots \| C_1 \| TC) = \tau_I \partial_{H_{k+1}}(C_{k+1} \| C_k \| \dots \| C_1 \| TC)$$

which is exactly M(k+1).

Therefore, we have $M(k) = PAL_k(\varepsilon)$, for all k. *end proof.*

REFERENCES

[BBK1] J.C.M Baeten, J.A, Bergstra & J.W. Klop, *On the consistency of Koomen's Fair Abstraction Rule,* report CS-R8511, Centre of Mathematics and Computer Science, Amsterdam 1985, to appear in TCS.

[BBK2] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Conditional axioms and α/β-calculus in process algebra,* report FVI 86-17, Computer Science Department, University of Amsterdam1986, to appear in: Proc. IFIP Conf. on Formal Description of Progr. Concepts, (M. Wirsing, ed.), Gl. Avernæs 1986, North Holland.

[BE] J.A. Bergstra, *A process creation mechanism in process algebra,* LGPS no.2, Department of Philosophy, University of Utrecht, 1985.

[BK1] J.A. Bergstra & J.W. Klop, *Algebra of communicating processes,* Proceedings of the CWI Symp. Math. & Comp. Sci., eds, J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, North-Holland, pp. 89-138, Amsterdam 1986.

[BK2] J.A. Bergstra & J.W. Klop, *Algebra of communicating processes with abstraction,* Theor. Comp. Sci. 37, pp. 77-121, 1985.

[BHR] S.D. Brookes, C.A.R. Hoare & A.W. Roscoe, *A theory of communicating sequential processes,* J.ACM 31, pp. 560-599, 1984.

[EB] J.C. Ebergen, *A technique to design delay-insensitive VLSI circuits,* report CS-R8622, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[HEN] M. Hennessy, *Proving Systolic Systems Correct,* TOPLAS 8 (3), pp. 344-387, 1986.

[KU] K.T. Kung, *Let's design algorithms for VLSI systems,* Proceedings of the Conference on VLSI: Architecture, Design, Fabrication (California Institute of Technology, January 1979).

[KW1] Kossen,L. & Weijland,W.P., *Correctness proofs for systolic algorithms: palindromes and sorting*, report FVI 87-04, Department of Computer Science, University of Amsterdam, 1987.

[KW2] Kossen,L. & Weijland,W.P., *Verification of a systolic algorithm for string comparison*, to appear as CWI-report, Centre for Mathematics and Computer Science, Amsterdam 1987.

[MC] C.A. Mead and L.A Conway, *Introduction to VLSI-systems,* Addison-Wesley Publ. Comp., Reading, Massachusetts, 1980.

[ME] G.J. Milne, *CIRCAL: a calculus for circuit description,* Integration 1, pp. 121-160, 1983.

[MI] R. Milner, *A calculus of communicating systems,* Springer LNCS 92, 1980.

[MW] Mulder,J.C. & Weijland,W.P., *Log-time sorting by square comparison*, report CS-R8729, Centre for Mathematics and Computer Science, Amsterdam 1987.

[RE] M. Rem, *partially ordered computations with applications to VLSI-design,* Proc. Found. of Comp. Sci. IV.2 (J. de Bakker & J. van Leeuwen eds.), MC tract 159, pp. 1-44,MC,1983.

[WE] Weijland, W.P., *A systolic algorithm for matrix-vector multiplication*, report FVI 87-08, Department of Computer Science, University of Amsterdam, 1987.