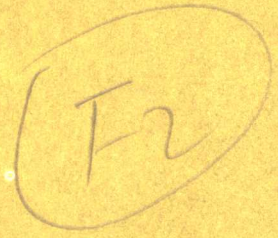
Logics of Programs

Edited by
Clarke Edmund and Dexter Kozen



Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

164

Logics of Programs

Workshop, Carnegie Mellon University
Pittsburgh, PA, June 6–8, 1983

Edited by Edmund Clarke and Dexter Kozen

Springer-Verlag
Berlin Heidelberg New York Tokyo 1984

Editors

Edmund Clarke
Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213 USA

Dexter Kozen
IBM Research
Box 218, Yorktown Heights, NY 10598 USA

AMS Subject Classifications (1980): 68C01, 68B10, 68B15, 03B45, 03D45

CR Subject Classifications (1982): F.3, F.4.1, B.2.2, B.6.3, D.2.4

ISBN 3-540-12896-4 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-12896-4 Springer-Verlag New York Heidelberg Berlin Tokyo

Printed in Germany

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

FOREWORD

Logics of Programs, as a field of study, touches on a wide variety of activities in computer science and mathematics. It draws on mathematical foundations of formal logic, semantics, and complexity theory, and finds practical application in the areas of program specification, verification, and programming language design. The Logics of Programs Workshop was conceived as a forum for the informal sharing of problems, results, techniques, and new applications in these areas, with special emphasis on bridging whatever abyss may exist between the theoreticians and the pragmatists.

The workshop was held on June 6-8, 1983 at Carnegie Mellon University. It was the fourth in an unofficial series, which started in 1979 with the workshop in Zürich organized by Erwin Engeler, and continued with the 1980 Poznan workshop organized by Andrzej Salwicki and the 1981 Yorktown Heights workshop organized by Dexter Kozen. Since the 1979 workshop, interest and participation has grown precipitiously: the CMU workshop drew 59 registered participants from 8 countries, as well as many unregistered participants. 38 technical papers were presented, representing the entire spectrum of activity in Logics of Programs from model theory to languages for the design of digital circuits. The contributions of the workshop participants appearing in this volume are unrefereed and are to be considered working papers.

The workshop was held in cooperation with the Association for Computing Machinery, and was made possible through the generous support of the National Science Foundation[1] and the Office of Naval Research[2]. We wish to thank all who helped with the organization of the workshop and preparation of the proceedings, especially John Cherniavsky, Robert Grafton, Magdalena Müller, and Nancy Perry.

Edmund Clarke
Dexter Kozen
Sept. 1, 1983

---

[1] grant MCS-8303082

[2] grant N00014-83-G-0079

CONTENTS

A STATIC ANALYSIS OF CSP PROGRAMS

Krzysztof R. APT

LITP, Université Paris 7
2, Place Jussieu
75251 PARIS
France

Abstract A static analysis is proposed as a method of reducing complexity of the correctness proofs of CSP programs. This analysis is based on considering all possible sequences of communications which can arise in computations during which the boolean guards are not interpreted. Several examples are provided which clarify its various aspects.

1. INTRODUCTION

Correctness proofs of concurrent and distributed programs are complicated because in general they are of the length proportional to the product of the lengths of the component programs. We claim in this paper that in the case of the CSP programs the length and the complexity of these proofs can be substantially reduced by carrying out first a preliminary static analysis of the programs. This analysis allows to reduce the number of cases which have to be considered at the level of interaction between the proofs of the component programs.

The analysis is quite straightforward and contains hardly any new ideas. It is based on considering all possible sequences of communications which can arise in computations during which the boolean guards are not interpreted. In this respect it bears a strong resemblance to the trace model for a version of CSP given in [H 1].

We apply this analysis to three types of problems. The first one consists of determining which pairs of input-output commands (i/o commands) may be synchronized during properly terminating computations. The second one consists of determining all possible configurations in which deadlock occurs. Finally we provide a sufficient condition for safety of a decomposition of CSP programs into communication-closed layers, a method of decomposition which has been recently proposed by Elrad and Francez [EF].

A similar analysis can be carried out for other programming languages which use rendez-vous as a sole means for communication and synchronization. In fact while writing this paper we encountered in the last issue of the Communications of ACM a paper by Taylor [T] in which such an analysis is carried out for ADA programs.

The only difference is in the presentation of this approach. R.N. Taylor presents an algorithm which computes all rendez-vous which may take place during execution of a program and all configurations in which deadlock may arise. His algorithm is also capable of determining which actions may occur in parallel. We on the other hand present the analysis in a formal language theory framework providing the rigorous definitions which can be used in the case of concrete examples. We also link this analysis with a subsequent stage being the task of proving correctness of the programs.

The paper is organized as follows. In the next section we introduce the basic definitions. In section 3 we provide three applications already mentioned above. Section 4 is devoted to a more refined analysis which takes into account the problem of termination of the repetitive commands. Finally in section 5 a number of conclusions is presented.

2. BASIC DEFINITIONS

We assume that the reader is familiar with the original article of Hoare [H]. Throughout the paper we consider programs written in a subset of CSP. We disallow nested parallel composition, assume that all variables are of the same type and consequently omit all the declarations. Additionally we allow output commands to be used as guards. For the reasons which will become clear later we label each occurence of an input or output command by a unique label.

By a parallel program we mean a program of the form $P_1 \| \ldots \| P_n$ where each P_i is a process. For simplicity we drop the process labels. So according to the notation of [H] each process is identified with the command representing its body. The name of the process can be uniquely determined from the position of the command within the parallel composition.

The analysis carried out here can be straightforwardly extended to the full CSP.

Throughout the paper we denote by S,T arbitrary (sequential) commands, by g guards, by b,c boolean expressions, by t expressions and by α,β i/o commands. Labels of the i/o commands are denoted by the letters k,l,m. Finally, we write $[\ \square_{i=1}^{m}\ g_i \to S_i\ ]$ instead of $[g_1 \to S_1 \square \ldots \square g_m \to S_m]$.

Consider now a parallel program $P_1 \| \ldots \| P_n$. We proceed in two stages.

1°) With each process P_i we associate a regular language $L(P_i)$ defined by structural induction. We put

$$L(x:=t) = L(skip) = \{\epsilon\},$$
$$L(l:P_j!t) = \{l : \langle i,j \rangle\},$$

$L(l{:}P_j?x) = \{l : \langle j,i\rangle\}$,

$L(S_1;S_2) = L(S_1)L(S_2)$,

$L(g \to S) = L(g)L(S)$,

$L(b) = \{\epsilon\}$, $L(b;l{:}\alpha) = L(b)L(l{:}\alpha)$ $(=L(l{:}\alpha))$,

$L([\ \overset{m}{\underset{i=1}{\square}}\ g_i \to S_i]) = \overset{m}{\underset{i=1}{\cup}}\ L(g_i \to S_i)$,

$L(*[\ \overset{m}{\underset{i=1}{\square}}\ g_i \to S_i]) = L([\ \overset{m}{\underset{i=1}{\square}}\ g_i \to S_i])^*$

Note that $L(P_i)$ is the set of all a priori possible communication sequences of P_i when the boolean guards are not interpreted. Each communication sequence consists of elements of the form $l{:}\langle i,j\rangle$ or $l{:}\langle j,i\rangle$ where l is a label of an i/o command uniquely identified and $\langle i,j\rangle$ ($\langle j,i\rangle$) records that fact that this i/o command stands for a communication from $P_i(P_j)$ to $P_j(P_i)$.

It is important that we associate with assignment and skip statements the set $\{\epsilon\}$ and not the empty language $\emptyset$. Otherwise not all communication sequences would be recorded in $L(P_i)$. The following example clarifies this issue.

Example 1

Let

$$P_1 \equiv [b_1 \to \text{skip} \ \square\ b_2 \to k{:}P_2!x]\ ;$$
$$*[l{:}P_2?y \to \ldots\ ;\ m{:}P_2!y]$$

where ... stands for a "private part" of P_1, i.e. a command not involving any i/o commands. Then

$$L(P_1) = \{(l{:}\langle 2,1\rangle)(m{:}\langle 1,2\rangle)\}^*$$
$$\cup\ \{k{:}\langle 1,2\rangle\}\{(l{:}\langle 2,1\rangle)(m{:}\langle 1,2\rangle)\}^*$$

If we associated with skip the empty language then the first part of $L(P_1)$ would not be present even though it represents possible communication sequences.

2°) We associate with $P_1\|\ldots\|P_n$ a regular language $L(P_1\|\ldots\|P_n)$. Its letters are of the form $k,l{:}\langle i,j\rangle$ standing for an instance of a communication between the output command of P_i labeled by k and the input command of P_j labeled by l.

First we define a projection function $[.]_i$ $(1\le i\le n)$ from the alphabet of $L(P_1\|\ldots\|P_n)$ into the alphabet of $L(P_i)$. We put

$$[k,l{:}\langle i,j\rangle]_i = k{:}\langle i,j\rangle$$
$$[k,l{:}\langle i,j\rangle]_j = l{:}\langle i,j\rangle$$
$$[k,l{:}\langle i,j\rangle]_h = \epsilon \quad \text{if} \quad h \neq i,j$$

and naturally extend it to a homomorphism from the set of words of $L(P_1\|\ldots\|P_n)$ into the set of words of $L(P_i)$.

We now define

$L(P_1 \| \ldots \| P_n) = \{h : [h]_i \in L(P_i),\ i = 1, \ldots, n\}$

Intuitively, $L(P_1 \| \ldots \| P_n)$ is the set of all possible communication sequences of $P_1 \| \ldots \| P_n$ which can arise in properly terminating computations during which the boolean expressions are not interpreted.

3. APPLICATIONS

1. Partial correctness

Given a parallel program $P_1 \| \ldots \| P_n$ we define

$STAT = \{(k:\alpha, l:\beta) : k:\alpha$ is from P_i, $l:\beta$ is from P_j,

$\&\ \exists h\, \exists a\, [h \in L(P_1 \| \ldots \| P_n),\ a$ is an element of h,

$L(k:\alpha) = \{[a]_i\}$ and $L(l:\beta) = \{[a]_j\}\}$.

Intuitively STAT (standing for static match) is the set of all pairs of i/o commands which can be synchronized during a properly terminating computation of $P_1 \| \ldots \| P_n$ which ignores the boolean guards.

The set STAT should be compared with two other sets of pairs of i/o commands :

$SYNT = \{(k:\alpha, l:\beta) : k:\alpha$ is from P_i, $l:\beta$ is from P_j and $k:\alpha$ and $l:\beta$ address each other (match)$\}$

$SEM = \{(k:\alpha, l:\beta) :$ in some "real" properly terminating computation of $P_1 \| \ldots \| P_n$ $k:\alpha$ and $l:\beta$ are synchronized$\}$.

In the proof systems of [AFR] and [LG] dealing with partial correctness of CSP programs the crucial proof rule is the one that deals with the parallel composition of the processes. First one introduces so called proof outlines for component processes. A proof outline of S is a special form of a proof of partial correctness of the program S in which each subprogram of S is preceded and succeded by an assertion. These assertions are supposed to hold at the moment when the control is at the point to which they are attached. As behaviour of each component process depends on the other processes we ensure the above property by comparing proof outlines of the component processes. Given a proof outline the only assertions which have to be justified using proof outlines ot other processes are those succeeding the i/o commands.

Thus one identifies all pairs of possibly matching i/o commands and checks that the assertions attached to them are indeed justified when the communication takes place. This part of verification of the proof outlines is called in [AFR] the cooperation test and in [LG] the satisfaction test.

If the proof outlines satisfy the test then one can pass to the conclusion stating partial correctness of the parallel program.

We now concentrate on the step consisting of identifying all pairs of possibly matching i/o commands. According to our definition this is the set SEM. But since SEM is in general not computable as a function of the program $P_1 \| \ldots \| P_n$, this set is replaced in [AFR] and [LG] by a larger set SYNT being obviously computable. We propose to replace in this analysis the set SEM by the set STAT.

Note that the following clearly holds.

Fact $SEM \subseteq STAT \subseteq SYNT$

Moreover, the set STAT is obviously computable. Using the set STAT instead of SYNT as an "approximation" for SEM is more economical as less checks in the cooperation (satisfaction) test phase are then needed. Also the proof outlines (and in the case of [AFR] - the global invariant) can be simplified.

As an illustration of the difference between the sets STAT and SYNT consider the following example :

Example 2

Let

$$P_1 \equiv k_1:P_2?x\ ;\ldots;k_2:P_2!z\ ;\ldots;\ *[b_1 \rightarrow \ldots;\ k_3:P_2?x\ ;\ldots;\ k_4:P_2!z\ ;\ldots],$$

$$P_2 \equiv l_1:P_1!y\ ;\ldots;\ l_2:P_1?u\ ;\ldots;\ *[b_2 \rightarrow \ldots;\ l_3:P_1!y\ ;\ldots;\ l_4:P_1?u\ ;\ldots]$$

Then

$$STAT = \{(k_i:\alpha_i,\ l_i:\beta_i) : 1 \le i \le 4\}$$

and

$$SYNT = \{(k_i:\alpha_i,\ l_j:\beta_j) : |i-j| \text{ is even}, 1 \le i,j \le 4\}.$$

Thus STAT has here 4 elements whereas SYNT has 8 elements.

The difference between STAT and SYNT becomes more evident for longer programs. For example if in the above programs both repetitive commands contained 2k instead of two alternating i/o commands in succession then STAT would contain $2(k+1)$ *elements whereas* SYNT *would contain* $2(k+1)^2$ *elements.*

It is important to note that the set STAT consists of pairs of i/o commands which can be synchronized during a properly terminating computation. The following two examples clarify this issue.

Example 3

Let

$$P_1 \equiv \ldots;\ k_1:P_1?x\ ;\ldots\ ,$$

$$P_2 \equiv \ldots;\ l_1:P_2!y\ ;\ldots;\ l_2:P_2!u\ ;\ldots$$

Then $L(P_1) = \{k_1:<2,1>\}$ and $L(P_2) = \{(l_1:<2,1>)(l_2:<2,1>)\}$ so $L(P_1 \| P_2) = \emptyset$. Thus STAT = $\emptyset$ even though the i/o commands labelled by k_1 and l_1, respectively can be synchronized. On the other hand, since $L(P_1 \| P_2) = \emptyset$, there does not exist a properly terminating computation of $P_1 \| P_2$. Indeed, for any properly terminating computation the sequence consisting of its consecutive communications belongs to $L(P_1 \| P_2)$.

Example 4

Let

$$P_1 \equiv [b_1 \rightarrow \ldots \square b_2 \rightarrow \ldots ; k_1:P_2?x;\ldots;k_2:P_2!x],$$

$$P_2 \equiv [c_1 \rightarrow \ldots \square c_2 \rightarrow \ldots ; l_1:P_1!y;\ldots;l_2:P_1!y].$$

Then $L(P_1) = \{\epsilon, (k_1:<2,1>)(k_2:<1,2>)\}$ and $L(P_2) = \{\epsilon, (l_1:<2,1>)(l_2:<2,1>)\}$ so $L(P_1 \| P_2) = \{\epsilon\}$. Thus STAT = $\emptyset$. The i/o commands labeled by k_1 and l_1, respectively can be synchronized but not during a properly terminating computation.

The situation when $L(P_1 \| \ldots \| P_n) = \emptyset$ should be compared with the situation when $L(P_1 \| \ldots \| P_n) = \{\epsilon\}$. In the first case no properly terminating computation of $P_1 \| \ldots \| P_n$ exists. In the latter case the properly terminating computations of $P_1 \| \ldots \| P_n$ can exist but in none of them a communication will take place.

In both cases STAT = $\emptyset$ so no cooperation (resp. satisfaction) test will take place in the proof rule dealing with parallel composition. This is in accordance with the fact that partial correctness of programs refers to properly terminating computations only. In both cases above no communication will take place in any such computation.

Finally we consider the following example :

Example 5

Let

$$P_1 \equiv *[k_1:P_2?x \rightarrow \ldots] ; k_2:P_2!u,$$

$$P_2 \equiv *[l_1:P_1!y \rightarrow \ldots] ; l_2:P_1?z.$$

Then STAT = SYNT = $\{(k_1:\alpha_i, l_1:\beta_i) : i=1,2\}$. Note however that the communication between the i/o commands with labels k_2 and l_2, respectively cannot take place as none of the repetitive commands can terminate. In particular no computation of $P_1 \| P_2$ terminates. The tools used so far do not allow us to deduce these facts formally. We shall return to this problem later.

2. Proofs of deadlock freedom

In the proof systemsof [AFR] and [LG] one proves deadlock freedom of the parallel programs by identifying first the set of blocked configurations, i.e. the vectors of control points corresponding to a deadlock. Then for each blocked configuration one shows that the conjunction of the assertions attached to the corresponding control points (and the global invariant in the case of [AFR]) is inconsistent. Thus the length of the proof of deadlock freedom is proportional to the number of blocked configurations.

We now suggest a more restricted definition of a blocked configuration which is sufficient for proofs of deadlock freedom and results in shorter proofs. The control points which are of interest here are those when the control resides in front of an i/o command or at the end of a process. With each control point of the first type we associate a set of i/o commands which can be at this point executed. With the control point of the second type we associate the set $\{\text{end } P_i\}$ corresponding to the situation when the control is at the end of the process P_i.

We define

$$C(k:\alpha) = \{\{k:\alpha\}\}$$

where $k:\alpha$ occurs in the process as an atomic command,

$$C([b_1 \to S_1 \,\square \ldots \square\, b_m \to S_m \,\square\, k_1:\alpha_1 \to S_{m+1} \square \ldots \square\, k_n:\alpha_n \to S_{m+n} \,\square b_{m+1}, k_{n+1}:\alpha_{n+1} \to$$

$$S_{m+n+1} \square \ldots \square b_{m+p}, k_{n+p}:\alpha_{n+p} \to S_{m+n+p}]) = \{A : A = \{k_i:\alpha_i : i=1,\ldots,n\} \cup B$$

where $B \subseteq \{k_{n+i}:\alpha_{n+i} : i=1,\ldots,p\}\}$, where $m \geq 0$ and $n+p \geq 1$,

$$C(*[\mathop{\square}_{i=1}^{m} g_i \to S_i]) = C([\mathop{\square}_{i=1}^{m} g_i \to S_i]).$$

For other type of commands S C(S) is not defined. Note that a typical set A considered above consists of all i/o guards which occur without the boolean guards together with a subset of those i/o guards which occur with a boolean guard.

Given now a process P_i we define $C(P_i)$ to be the union of all sets C(S) for S being a subprogram of P_i together with the element $\{\text{end } P_i\}$. Each element of $C(P_i)$ corresponds to a unique control point within P_i.

The identification of all blocked configurations depends on the fact whether so called distributed termination convention (d.t.c) of the repetitive commands is taken taken into account. According to this convention a repetitive command can be exited when all processes addressed in the guards with boolean part true have terminated. This convention corresponds to the following definition of a guard being failed : a guard fails if either its boolean part evaluates to false or the process addressed in its i/o part has terminated. A repetitive command is exited when all its guards fail. If in the definition of a failure of a guard we drop the second alternative we obtain the usual termination convention of the repetitive commands. In [H] the

distributed termination convention is adopted.

Consider first the simpler case when the usual termination convention is used.

A triple $\langle A_1,\ldots,A_n\rangle$ from $C(P_1)x\ldots xC(P_n)$ is called <u>blocked</u> if

i) $\exists\, i \;\; A_i \neq \{\text{end } P_i\}$

(not all processes have terminated)

ii) $\left(\bigcup_{i\neq j} A_i \times A_j\right) \cap SYNT = \emptyset$

(no communication can take place)

Alternatively ii) can be stated as : no pairs of elements from A_i and A_j $(i\neq j)$ match. The notion of a blocked tuple is from [AFR].

Let $Init(L)$ for a formal language L denote its <u>left factor</u> i.e. the set $\{u : \exists\, w(uw \in L)\}$. We now put

$$LP(P_1\|\ldots\|P_n) = \{h : [h]_i \in Init(L(P_i)),\ i=1,\ldots,n\}.$$

Intuitively, $LP(P_1\|\ldots\|P_n)$ is the set of all possible communication sequences of $P_1\|\ldots\|P_n$ which can arise in partial computations during which the boolean guards are not interpreted.

We now say that a tuple $\langle A_1,\ldots,A_n\rangle$ from $C(P_1)x\ldots xC(P_n)$ is <u>statically blocked</u> if

i) it is blocked

ii) $\exists\, h \in LP(P_1\|\ldots\|P_n) \;\; \forall\, i$

$$[A_i \neq \{\text{end } P_i\} \Rightarrow \forall\, \underline{d} \in A_i([h]_i a \in Init(L(P_i))) \text{ where } L(\underline{d}) = \{a\}$$

$$\wedge\; A_i = \{\text{end } P_i\} \Rightarrow [h]_i \in L(P_i)]$$

The second condition states that there exists a communication sequence which reaches the vector of the control points associated with $\langle A_1,\ldots,A_n\rangle$. Reachability is checked by considering the projections $[h]_i$ of the sequence h. If $A_i \neq \{\text{end } P_i\}$ then $[h]_i a$ for all $a \in \{L(\underline{d}) : \underline{d} \in A_i\}$ should be an initial part of a sequence from $L(P_i)$. If $A_i = \{\text{end } P_i\}$ then $[h]_i$ should be a sequence from $L(P_i)$.

If d.t.c. is used then we should add the following condition to the definition of a blocked triple

iii) For no $i_o, i_1,\ldots,i_k$ from $\{1,\ldots,n\}$, where $i_j \neq i_l$ for $j \neq l$: $A_{i_j} = \{\text{end } P_{i_j}\}$ and the processes addressed in the i/o commands of A_{i_o} are all among $\{P_{i_1},\ldots,P_{i_k}\}$.

This condition states that no exit can take place due to the distributed termination convention. Thus the set A_{i_o} should correspond to a repetitive command.

We denote the set of all statically blocked tuples by STATB and the set of all blocked tuples by SYNTB.

We now consider a couple of examples.

Example 6

Consider the processes P_1 and P_2 from the example 2. D.t.c. cannot be used here. It is easy to see that

$$\begin{aligned}\text{SYNTB} = {} & \{\langle\{k_i:\alpha_i\}, \{l_j:\beta_j\}\rangle : |i-j| \text{ is odd}, 1 \le i,j \le 4\} \\ & \cup \{\langle\{k_i:\alpha_i\}, \{\text{end } P_2\}\rangle : 1 \le i \le 4\} \\ & \cup \{\langle\{\text{end } P_1\}, \{l_j:\beta_j\}\rangle : 1 \le j \le 4\}\end{aligned}$$

whereas

$$\begin{aligned}\text{STATB} = {} & \{\langle\{k_3:\alpha_3\}, \{\text{end } P_2\}\rangle, \\ & \langle\{\text{end } P_1\}, \{l_3:\beta_3\}\rangle\}\end{aligned}$$

Thus SYNTB has 16 elements whereas STATB has only two elements.

Example 7

Let

$$\begin{aligned}P_1 \equiv {} & \ldots; k_1:P_2!x\ ;\ldots;\ k_2:P_2?z\ ;\ldots; \\ & *[b_1 \to k_3:P_2!x\ ;\ldots;\ k_4:P_2?z\ ;\ldots],\end{aligned}$$

$$\begin{aligned}P_2 \equiv {} & l_1:P_1?y\ ;\ldots;\ l_2:P_1!u\ ;\ldots; \\ & *[l_3:P_1?y \to \ldots\ l_4:P_1!u\ ;\ldots].\end{aligned}$$

This is a structure of the program partitioning a set studied in [D] and [AFR].

Consider first the case when the distributed termination convention is not used. Then SYNTB and STATB are the same as in the previous example.

Suppose now that d.t.c. is used. Then

$$\begin{aligned}\text{SYNTB} = {} & \{\langle\{k_i:\alpha_i\}, \{l_j:\beta_j\}\rangle : |i-j| \text{ is odd}, 1 \le i,j \le 4\} \\ & \cup \{\langle\{k_i:\alpha_i\}, \{\text{end } P_2\}\rangle : 1 \le i \le 4\} \\ & \cup \{\langle\{\text{end } P_1\}, \{l_j:\beta_j\}\rangle : j = 1,2,4\}\end{aligned}$$

and

$$\text{STATB} = \{\langle\{k_3:\alpha_3\}, \{\text{end } P_2\}\rangle\}.$$

Here SYNTB has 15 elements whereas STATB only one. Note that the only staticly blocked pair cannot arise in actual computations either. The only way P_2 can terminate is due to the termination of P_1. Thus if the control in P_2 is at its end then the same must hold for P_1. We note that our analysis is not precise enough in order to deal with this type of situations. The next example gives more evidence to this effect.

Example 8

Let for i=1,...,n

$P_i \equiv *[b_i \; ; \; P_{i-1}!x_i \rightarrow \ldots$

$\square \; c_i ; P_{i+1}!x_i \rightarrow \ldots$

$\square \; P_{i-1}?y_i \rightarrow \ldots$

$\square \; P_{i+1}?z_i \rightarrow \ldots$

where the addition and substraction is modulo n.

This is a structure of the distributed gcd program considered in [AFR]. The labels of i/o commands are omitted as they are not needed here.

We have in the case when d.t.c. is not used

$$\begin{aligned} SYNTB = \{&\langle A_1,\ldots,A_n\rangle : \exists\, i \; A_i \neq \{\text{end } P_i\} \\ &\wedge \forall\, i[P_{i-1}!x_i \in A_i \rightarrow A_{i-1} = \{\text{end } P_{i-1}\} \\ &\wedge P_{i+1}!x_i \in A_i \rightarrow A_{i+1} = \{\text{end } P_{i+1}\} \\ &\wedge A_i \neq \{\text{end } P_i\} \rightarrow \{P_{i-1}?y_i,\ P_{i+1}?z_i\} \subseteq A_i]\} \end{aligned}$$

and

STATB = SYNTB.

Suppose now that d.t.c. is used. Then

$$\begin{aligned} SYNTB = \{&\langle A_1,\ldots,A_n\rangle : \exists\, i \quad A_i \neq \{\text{end } P_i\} \\ &\wedge \forall\, i\ [P_{i-1}!x_i \in A_i \rightarrow A_{i-1} = \{\text{end } P_{i-1}\} \\ &\quad \wedge P_{i+1}!x_i \in A_i \rightarrow A_{i+1} = \{\text{end } P_{i+1}\} \\ &\quad \wedge A_i \neq \{\text{end } P_i\} \rightarrow (\{P_{i-1}?y_i,\ P_{i+1}?z_i\} \subseteq A_i \\ &\qquad\qquad \wedge (A_{i-1} \neq \{\text{end } P_{i-1}\} \vee A_{i+1} \neq \{\text{end } P_{i+1}\}))]\} \end{aligned}$$

and once again STATB = SYNTB.

We see that in this example all blocked tuples are statically possible. The reason for it is that recording sequences of communications does not suffice to distinguish between two control points : the beginning and the end of a repetitive command.

On the other hand a simple informal argument allows to reduce the number of blocked triples which can arise in actual computations to one. The argument runs as follows. Suppose that d.t.c. is not used. Then no process P_i can terminate. Assume now that this convention is used. If some process P_i has terminated then by d.t.c. his neighbours P_{i-1} and P_{i+1} must have terminated, as well. Thus no process can terminate as the first one. In other words no process can terminate.

Thus in both cases a blocked tuple $\langle A_1,\ldots,A_n\rangle$ with some $A_i = \{\text{end } P_i\}$ is not possible. This reduces the number of possible blocked tuples to one being $\langle A_1,\ldots,A_n\rangle$ where for $i = 1,\ldots,n$ $A_i = \{P_{i-1}?y_i, P_{i+1}?z_i\}$.

In the next section we propose a more refined analysis which leads to a more restricted notion of static match and staticly blocked configurations. These notions will allow to deal properly with the above examples.

3. Proofs of safety of a decomposition of programs into communication-closed layers

In a recent paper [EF] Elrad and Francez proposed a method of decomposition of CSP programs which simplifies their analysis and can be used for a systematic construction of CSP programs. It is defined as follows.

Suppose that we deal with a parallel program P of the form $P_1 \| \ldots \| P_n$ where for all $i=1,\ldots,n$ $P_i \equiv S_i^1 ;\ldots; S_i^k$. Some of the commands S_i^k can be empty. We call the parallel programs $T_j \equiv S_1^j \| \ldots \| S_n^j$ $(j=1,\ldots,k)$ the *layers* of P.

A layer T_j is called *communication-closed* if there does not exist a computation of P in which a communication takes place between two i/o commands from which one lies within T_j and the other outside T_j. A **decomposition** $T_1;\ldots;T_k$ of P is called *safe* iff all the layers T_j are communication-closed. In other words a decomposition $T_1;\ldots;T_k$ of P is safe if there does not exist a computation of P with a communication involving two i/o commands from different layers.

In [EF] also more general types of layers are considered whose boundaries may cross the repetitive commands. Our analysis does not extend to such decompositions. The interest in considering safe decompositions stems from the following observation.

Fact ([EF]) Suppose that $T_1;\ldots;T_k$ is a safe decomposition of the parallel program P. Then the programs $T_1;\ldots;T_k$ and P are input-output equivalent.

Proof (informal) Obviously every computation of $T_1;\ldots;T_k$ is also a computation of P. Consider now a properly terminating computation of P. Due to safety of the decomposition we can rearrange some steps of this computation so that it becomes a properly terminating computation of $T_1;\ldots;T_k$. Both computations terminate in the same final state.

Thus both programs generate the same pairs of input-output states. □

As an example of a safe decomposition consider the following program

$$P \equiv P_1?x \| P_2!y \| P_3?u \| P_4!z$$

Consider now the layers

$$T_1 \equiv P_1?x \| P_2 y \| \lambda \| \lambda$$