Automatic Construction of CSP Programs from Sequential Non-Deterministic Programs

Abha Moitra

TR 84-597
March 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

Automatic Construction of CSP Programs from Sequential Non-Deterministic Programs

Abha Moitra

Computer Science Department
Cornell University
Ithaca, New York 14853

ABSTRACT

In this paper we describe a systematic method for transforming a sequential program, written in a guarded command language, into a distributed program, written in CSP. The variables of the sequential program are first partitioned into n disjoint sets, and then the program is transformed into a CSP program of n communicating processes. The two versions of the program are shown to be *strongly equivalent,* in the sense that they exhibit the properties of reaching the same final states and of either aborting, terminating, or running forever. We also discuss the conditions under which, when compared to the execution of the original sequential program, a speed-up in the execution of the resulting distributed program can be achieved.

1. INTRODUCTION

An interesting question that can be posed is whether it is possible to transform sequential programs into distributed programs. In this paper we will investigate this problem by considering the transformation of sequential programs written in the guarded command language (GC) [3] into CSP programs [6]. Though this may seem paradoxical, the synthesis of distributed programs from sequential programs appears to be simpler and more straightforward than the synthesis of sequential programs from input-output assertions (e.g. [11]). The reason for this is that it is more difficult to go from logical formulae (i.e., input-output assertions) to a program than from a sequential program to another program even when the latter is a distributed program.

The design of sequential programs itself was described as an art until formal techniques were provided for their construction (e.g., Dijkstra [3], Gries [5]). The design of distributed programs has so far remained an art. Our purpose here is to show that starting with a (correct) sequential program, a ***strongly equivalent*** distributed program can be synthesized in a mechanical way.

Given a sequential program P, and a partition of its variables into n disjoint sets, transformation of P results in a CSP program S :: $[P_1 \; || \; .. \; || \; P_n]$ and P and S are strongly equivalent. The transformation is constructive and, in principle, can be automated. The structure, and hence the simplicity and clarity of the CSP program, depends critically on both the structure of the original sequential program and the partitioning of the variables. Different n way partitions of the variables of the same sequential program P can produce CSP programs with greatly differing properties. Later on, we shall indicate some guidelines for effectively partitioning variables.

2. STRONG EQUIVALENCE

In this section we consider the conditions under which two programs P and Q are said to be strongly equivalent. In principle, P and Q could be written in different languages, though here we shall only consider these programs to be written in GC and CSP respectively.

Definition 1 : $VARS(X)$ is the set of variables used in the program X. □

Definition 2 : $VAL(V)$ denotes value assignment to the set of variables V. $VAL(V)$ is called the ***valuation***. □

$VAL(VARS(X))$ therefore assigns values to all the variables of program X. So a *state* of the program X can be represented as an appropriate $VAL(VARS(X))$.

Definition 3 : $\{VAL_i(V)\}\ S\ \{VAL_f(V)\}$ holds iff there exists a computation of program S starting in $VAL_i(V)$ and terminating in $VAL_f(V)$. □

Note that if S does not terminate then $\{VAL_i(VARS(S))\}\ S\ \{VAL_f(VARS(S))\}$ can be said to hold trivially for any $VAL_f(VARS(S))$.

Definition 4 : Programs P and Q are said to be ***weakly equivalent*** if

$$\{VAL_i(V)\}\ P\ \{VAL_f(V)\} \Leftrightarrow \{VAL_i(V)\}\ Q\ \{VAL_f(V)\}$$

where $V = VARS(P)$ and $V \subseteq VARS(Q)$

for all possible valuations $VAL_i(V)$. □

Definition 5 : Programs P and Q are said to be ***strongly-equivalent*** (denoted by $P \equiv Q$) if they are weakly equivalent and for any initial valuation VAL :

(i) P can abort from VAL iff Q can abort from VAL

(ii) P terminates from VAL iff Q terminates from VAL □

3. OVERVIEW OF THE TRANSFORMATION

Consider the class of all sequential programs that can be written in GC. We intend to transform any program P written in GC into a CSP program. We are given

(i) the sequential program P

and (ii) the partition $VARS(P) = W_1 \cup \cdots \cup W_n$

where for $1 \leq i, j \leq n$, $W_i \neq \emptyset$,

and $W_i \cap W_j = \emptyset$, whenever $i \neq j$

We shall transform P into a CSP program $S :: [P_1 \ || \ .. \ || \ P_n]$ where

$$V_i = VARS(P_i) = W_i \cup \{temp_{i,j} \mid 1 \le j \le u_i\}$$

$$\text{and } V = VARS(S) = V_1 \cup .. \cup V_n$$

and $temp_{i,j}$, $1 \le i \le n$, $1 \le j \le u_i$ are an arbitrary but finite number of new variables. (These new variables will be used as integers, booleans and we assume that, as required, they are typed appropriately.) Note that $VARS(P_i) \cap VARS(P_j) = \emptyset$ for $i \ne j$.

The transformation works in two steps. First, the original GC program P is ***translated*** to another GC program $T[P] \Rightarrow Q$. The program Q is over the variables $V = V_1 \cup .. \cup V_n$. Q is then ***restricted*** to V_i (denoted by $Q \mid V_i$) to obtain the process P_i. That is

Step 1 *Translation* : $T[P] \Rightarrow Q$

Step 2 *Restriction* : $Q \mid V_i \Rightarrow P_i$ for $1 \le i \le n$

Goal : $P \equiv S :: [\ T[P] \mid V_1 \ || \ \cdots \ || \ T[P] \mid V_n\ ]$

Translation and restriction are defined to preserve sequencing; i.e., $T[S_1;\ S_2] = T[S_1];\ T[S_2]$ and $(S_1;\ S_2) \mid V_i = S_1 \mid V_i\ ;\ S_2 \mid V_i$, where S_1 and S_2 are statements of GC and ';' denotes sequential composition. Hence we only need to consider the translation and restriction of individual statements.

4. NOTATION

We shall use the following notation :

1. $exp^{a;b}_{c,d}$ to mean that all free occurrences of a and b are to be replaced by c and d respectively in exp

2. $\{*\}[\, b_j \xrightarrow[j=1]{h} S_j] \equiv \{*\}[\, b_1 \rightarrow S_1 \;\square\; .. \;\square\; b_h \rightarrow S_h \,]$

   The optional $\{*\}$ indicates that the above notation holds for both alternative and iterative commands.

3. $\{*\}[\, b_j \xrightarrow[j=1,..,h \text{ and } Cond1]{} S_j \,]$

   represents an alternative/iterative command which consists of all guarded commands $b_j \rightarrow S_j$, $1 \leq j \leq h$ which also satisfy the condition $Cond1$.

4. $x_i \overset{n}{\underset{i=1}{:===}} y_i \equiv \; x_1 := y_1 \,; \;\; .. \; x_n := y_n$

5. $A \overset{Cond1}{===>} B$

   $\overset{Cond2}{===>} C$

   to mean that under $Cond1$ ($Cond2$) A can be rewritten as B (C).

Let $EXT[V_i,\; P]$ denote the set of variables from $V - V_i$ used either in assignments to variables in V_i in the sequential program P or in boolean guards in P. (Informally, since process P_i will be over the variables V_i, we are identifying all the 'external' variables that will be used by process P_i during its execution.)

Definition 6 : $EXT[V_i,\; P]$, $1 \leq i \leq n$ is defined as follows.

$$EXT[V_i,\; S_1\,;\; S_2] = EXT[V_i,\; S_1] \cup EXT[V_i,\; S_2] \quad \text{where } P \equiv S_1;\; S_2$$

$$EXT[V_i,\; \{*\}[\, b_j \xrightarrow[j=1]{m} S_j]\,] = EXT[V_i,\; [\, b_1 \rightarrow S_1]\,]$$

$$\cup .. \cup EXT[V_i,\; [\, b_m \rightarrow S_m]\,]$$

$$EXT[V_i,\; skip] = \emptyset$$

$$EXT[V_i,\ x := exp] = \text{if } x \in V_i \text{ then } \{y \mid y \text{ is in } exp \text{ and there is some } j,\ 1 \leq j \leq n \text{ such that } i \neq j \text{ and } y \in V_j\} \text{ else } \emptyset$$

$$EXT[V_i,\ [b \rightarrow S]\,] = EXT[V_i,\ temp_{i,1} := b] \cup EXT[V_i,\ S] \qquad \Box$$

5. TRANSFORMATION OF SKIP AND ASSIGNMENT STATEMENTS

We now define translation and restriction for the *skip* and assignment statements.

Translation

T1. Define $T[skip] \Rightarrow skip$

T2. if for all $1 \leq i \leq n$ $EXT[V_i,\ x := exp] = \emptyset$ then define

$$T[x := exp] \Rightarrow x := exp$$

T3. if for some i, $1 \leq i \leq n$ $EXT[V_i,\ x := exp] = \{y_1, \ldots, y_j\}$ and $j \geq 1$ then define

$$T[x := exp] \Rightarrow \quad temp_{i,m} \mathrel{\underset{m=1}{\overset{j}{:===}}} y_m;$$

$$x := exp^{y_m}_{temp_{i,m}}$$

where $y_m = y_1, \ldots, y_j$ and $temp_{i,m} = temp_{i,1}, \ldots, temp_{i,j}$

That is, the assignment $x := exp$ is rewritten to involve variables only from V_i. T3 ensures that every assignment statement in $T[P]$ uses at most one variable from $V - V_i$ for assignment to any variable in V_i, $1 \leq i \leq n$. Further, it also ensures that the only assignments to variables in V_i using variables from $V - V_i$, $1 \leq i \leq n$ are of the form $temp_{i,j} := t$.

In translation T3 we are making use of variables $temp_{i,1},..,temp_{i,j}$. These variables may have already been used in the translation of previous statements and so in effect we are reusing these variables. Instead of reusing these variables we can always use 'fresh' variables for the translation of any statement since we are assured of an arbitrary but finite supply of such variables. But for simplicity of presentation we shall feel free to reuse these variables in the translation rules.

Restriction

R1. For $1 \leq i \leq n$ we define $\quad skip \mid V_i \;\Rightarrow\; skip$

R2. When $EXT[V_i,\ x := exp] = \emptyset$ for $1 \leq i \leq n$ and $x \in V_k$

then for $1 \leq j \leq n$ define

$$x := exp \mid V_j \overset{j=k}{===>} x := exp$$

$$\overset{j \neq k}{===>} skip$$

R3. When $EXT[V_i,\ x := t] = \{t\}$ and $t \in V_j$, for some $1 \leq i, j \leq n$. Then for $1 \leq k \leq n$ we define

$$x := t \mid V_k \overset{k=i}{===>} P_j\ ?\ x$$

$$\overset{k=j}{===>} P_i\ !\ t$$

$$\overset{k \neq i,j}{===>} skip$$

The transformation of assignment statements allows for parallelism. For example, if two consecutive assignment statements in P involve variables from different sets, then these assignment statements will be assigned to different processes and hence can be executed in parallel. Communication between processes is required if an assignment statement involves variables from two sets. An obvious consideration to be applied in partitioning variables is to minimize the amount of inter-process communication that will be required. We will consider this issue in later sections.

The transformation will introduce a number of inessential *skip* statements and we assume that they can be easily removed from the final CSP program. In all the examples we shall present such *skip* statements will be removed.

6. MINIMIZING COMMUNICATION

The transformation of the assignment statement

$$x := exp \qquad \text{where } x \in V_i \text{ for some } 1 \leq i \leq n \tag{1}$$

always results in one or more assignment statements. Of these, each assignment statement of the form

$$temp_{i,k} := y \tag{2}$$

introduced by T3 results in a pair of input and output commands of CSP. To minimize such assignment statements, consider the case where (1) is followed by a sequence of assignment statements Q :

$$x := exp ;\ Q$$

Then in program Q we are justified in substituting $temp_{i,k}$ for y in any assignment to variables in V_i until the value of the variable y is altered in Q. This amounts to propagating the effect of the assignment (1) forward, with the result that the number of assignment statements introduced subsequently by T3 can be reduced. We do not formalize this propagation of values as a part of the translation T, but will indicate its effect on Q as $Subs(Q)$.

7. TRANSFORMATION OF ALTERNATIVE COMMANDS

In the execution of an alternative command, at most one of its guarded commands should be executed. Consider the alternative command

$$[b_j \xrightarrow[j=1]{h} S_j] \tag{3}$$

(A) If all the variables in (3) belong to a single set V_i for some $1 \leq i \leq n$ then (3) should be executed by process P_i and we get the following transformation.

Translation

T4. if for some $1 \leq i \leq n$ $VARS(\, [b_j \xrightarrow[j=1]{m} S_j])\subseteq V_i$ then define

$$T[\,[b_j \xrightarrow[j=1]{m} S_j]\,] \Rightarrow [b_j \xrightarrow[j=1]{m} S_j]$$

Restriction

R4. if for some $1 \leq i \leq n$ $VARS(\, [b_j \xrightarrow[j=1]{m} S_j])\subseteq V_i$ then for $1 \leq k \leq n$ define

$$[b_j \xrightarrow[j=1]{m} S_j] \mid V_k \overset{k=i}{===>} [b_j \xrightarrow[j=1]{m} S_j]$$

$$\overset{k \neq i}{===>} skip$$

(B) If (3) involves variables from two or more sets, then rewrite it in an equivalent form as

$temp_{i,1} := 0$;

$*[\ temp_{i,1} = 0 \wedge b_j \xrightarrow[j=1]{m} S_j;\ temp_{i,1} := 1\ ]$;

$[temp_{i,1} > 0 \rightarrow skip\ ]$

where $1 \leq i \leq n$. In this way, we can ensure that at most one guarded command of (3) is executed. This rewriting is helpful since the alternative command introduced above can be transformed by T4, R4 and the transformation of the iterative command will be taken up in the next section. The transformation follows :

Translation

T5. if there does not exist k, $1 \leq k \leq n$: $VARS(\ [b_j \xrightarrow[j=1]{m} S_j]\ ) \subseteq V_k$ then

define

$$T[\ [b_j \xrightarrow[j=1]{m} S_j\ ]\ ] \Rightarrow temp_{i,1} := 0 ;$$

$$\mathrm{T}[*[\ temp_{i,1} = 0 \wedge b_j \xrightarrow[j=1]{m} S_j;\ temp_{i,1} := 1]\ ] ;$$

$$[\ temp_{i,1} > 0 \rightarrow skip]$$

The execution of an alternative command has been 'centralized' by involving one process P_i (through variable $temp_{i,1}$) in each of its guarded commands. This is to ensure that at most one guarded command of the original alternative command is executed. Also, note that if the sequential program P aborts due to the failure of all the guards in an alternative command, then the CSP program will also abort.

8. TRANSFORMATION OF ITERATIVE COMMANDS

Consider the iterative command

$$*[\ b_j \xrightarrow[j=1]{h} S_j] \tag{4}$$

(A) If all the variables involved in each guarded command of (4) belong to one set then that entire guarded command must be made part of the corresponding process.

Translation

T6. if for all $1 \leq j \leq h$ there exists some m_j, $1 \leq m_j \leq n$ such that $VARS(\ [\ b_j \rightarrow S_j]\ ) \subseteq V_{m_j}$ then define

$$T[*[\ b_j \xrightarrow[j=1]{h} S_j\ ]\ ] \Rightarrow *[\ b_j \xrightarrow[j=1]{h} S_j]$$

Restriction

R5. if for all $1 \leq j \leq h$ there exists some m_j, $1 \leq m_j \leq n$ such that $VARS(\ [\ b_j \rightarrow S_j]\ ) \subseteq V_{m_j}$ then for $1 \leq k \leq n$ define

$$*[\ b_j \xrightarrow[j=1]{h} S_j]\ |V_k \xRightarrow{k\ \in\ \{m_1,..,m_h\}} *[\ b_j \xrightarrow[j=1,..,h \text{ and } k=m_j]{} S_j]$$

$$\xRightarrow{k\ \notin\ \{m_1,..,m_h\}} skip$$

That is, the iterative command (4) is such that each guarded command has variables from only one set, say V_p where $1 \leq p \leq n$. So when restricting it by the variable set V_p we retain only those guarded commands which are over variables only from V_p. This transformation results in an iterative command for each of the involved processes. Note that no synchronization is required between the processes in this case.

Remark : Any boolean expression b_j can be rewritten in conjunctive normal form as $b_j \equiv b_{j1} \wedge .. \wedge b_{jn} \wedge b_j'$, with the restriction that b_{ji}, $1 \leq i \leq n$ is a boolean expression over variables from V_i only. □

Whenever we rewrite a boolean expression we will assume that it is written in the above mentioned conjunctive normal form.

Definition 7 : A process P_j is said to be ***actively involved*** in a boolean expression b, where $b \equiv b_1 \wedge .. \wedge b_n \wedge b'$, if b_j is not identically true. □

(B) If there exists a process which is actively involved in all the boolean guards of an iterative command then this process is chosen as the ***leader***. (If there is more than one such process, one of them is arbitrarily chosen the leader.) The leader keeps track of which boolean guards cannot be met and can therefore exit from the iterative command. The reason for the leader process to be actively involved in all the boolean guards is that no alteration in the values of variables of other processes can be made without its participation. The leader process also has the responsibility of ensuring that all the other processes also exit the iterative command when it does so.

(C) If (A) and (B) are not applicable, then dummy variables can be added and boolean guards augmented so that condition (B) is met; for example (4) is rewritten as

$$temp_{i,1} := 0 \;;$$

$$*[\; temp_{i,1} = 0 \wedge b_j \xrightarrow[j=1]{h} S_j \;]$$

where $1 \leq i \leq n$. (We would normally choose i to be the index of the process actively involved in most of the boolean guards.) Now condition (B) is met and P_i is the leader process.

We can now indicate how an iterative statement of the form (4) which meets condition (B) is to be transformed. Without loss of generality, let P_1 be the leader process. For the leader process P_1, we demand the following structure.

$temp_{1,j} \overset{h}{\underset{j=1}{:===}} true;$ (* $temp_{1,m}$ = false $\Rightarrow$ b_m = false *)

$*[\ temp_{1,j} \wedge b_{j1} \xrightarrow[j=1]{h}$ inform all other processes that the $j'th$ guarded command

has been chosen for possible execution ;

obtain values so that b_j can be evaluated ;

inform all other processes about the value of b_j ;

$[b_j \rightarrow T[\ S_j\ ] \,|\, V_1;$ (* execute S_j *)

$temp_{1,k} \overset{h}{\underset{k=1}{:===}} true$

(* all b_m, $1 \leq m \leq h$ may be true *)

$\square$

$\neg\ b_j \rightarrow temp_{1,j} :=$ false

(* $j'th$ guard cannot be chosen *)

]

] ; inform all other processes about termination of iterative command by sending a number greater than h

For any process other than the leader process, P_k where $2 \leq k \leq n$, the following structure is required.

$temp_{k,1} := true$; (* $temp_{k,1} = true \Rightarrow$ execute iterative command *)

[$temp_{k,1} \rightarrow P_1$? $temp_{k,2}$; (if $temp_{k,2} = j$ and $j \leq h$ then execute $b_j \rightarrow S_j$ *)

[$temp_{k,2} = j \xrightarrow[j=1]{h}$ send values so that P_1 can evaluate b_j ;

P_1 ? $temp_{k,3}$; (* $temp_{k,3} = b_j$ *)

[$temp_{k,3} \rightarrow T[\ S_j\ ] | V_k$ (* execute S_j *)

□

$\neg\ temp_{k,3} \rightarrow skip$

]

□

$temp_{k,2} > h \rightarrow temp_{k,1} := false$ (* exit loop *)

]

]

We now give the transformation of iterative commands to achieve the desired result.

Translation

T7. If *[$b_j \xrightarrow[j=1]{h} S_j$] has a leader process then without loss of generality assume that P_1 is the leader process. The translation is then defined as follows :

$$T[\,*[\, b_j \xrightarrow[j=1]{h} S_j]\,]$$

$$\Rightarrow temp_{1,m} \underset{m=1}{\overset{h}{:===}} true; \qquad (*\ temp_{1,m} = false \ \Rightarrow\ b_m = false\ *)$$

$$*[\, b_{j1} \wedge temp_{1,j} \xrightarrow[j=1]{h} T[\, temp_{1,h+1} := b_j\,]; \qquad (*\ \text{evaluate guard}\ *)$$

$$[\, temp_{1,h+1} \rightarrow T[\, S_j]\,;$$

$$temp_{1,m} \underset{m=1}{\overset{h}{:===}} true$$

$$(*\ \text{all } b_m,\ 1 \le m \le h \text{ may be true } *)$$

$$\Box$$

$$\neg temp_{1,h+1} \rightarrow temp_{1,j} := false \qquad (*\ b_j \text{ is false } *)$$

$$]$$

$$]$$

where b_{j1} is part of the boolean expression b_j which is over variables only from V_1.

In the above translation note that the variables $temp_{1,m}$, $1 \le m \le h+1$ should not be used in $T[\, S_j\,]$. So, in the translation of nested statements, fresh $temp_{i,j}$ variables should be used.

T7 above has defined the translation of an iterative command by introducing an additional alternative command. The only restriction (R4) we have defined for an alternative command is for the case when all the variables in it are from one set say V_i, $1 \le i \le n$. Since the alternative command introduced by T7 may not satisfy this condition, the restriction defined to handle the code obtained by T7 should be able to handle the introduced alternative command. This is achieved by the following restriction.

Restriction

R6. Let process P_1 be the leader process for the iterative command

$*[\ b_j \xrightarrow[j=1]{h} S_j\ ]$, and if for $1 \leq j \leq h$

$$S_j \equiv Q_{j1}\ ;\ [\ temp_{1,h+1} \rightarrow Q_{j2} \square \neg temp_{1,h+1} \rightarrow Q_{j3}\ ]$$

then for $1 \leq i \leq n$ define $*[\ b_j \xrightarrow[j=1]{h} S_j\ ]\,|\,V_i$

$$\overset{i=1}{===>}\ *[\ b_j \xrightarrow[j=1]{h} P_2\,!\,j\ ;\ ..P_n\,!\,j\ ;$$

(* code for the leader process *)

$$Q_{j1}\,|\,V_i\ ;$$

$$P_2\,!\,temp_{1,h+1}\ ;\ ..\ P_n\,!\,temp_{1,h+1}\ ;$$

$$[\ temp_{1,h+1} \rightarrow Q_{j2}\,|\,V_i$$

$$\square$$

$$\neg temp_{1,h+1} \rightarrow Q_{j3}\,|\,V_i$$

$$]\ ;$$

$$]\ ;\ P_2\,!\,h+1\ ;\ ..\ P_n\,!\,h+1$$

$\overset{i \neq 1}{===>}$ $temp_{i,1} := true$; (* code for non-leader process *)

$*[\ temp_{i,1} \rightarrow P_1\ ?\ temp_{i,2}$;

(* if $temp_{i,2} = j$ and $j \leq h$ then execute $b_j \rightarrow S_j$ *)

$[\ temp_{i,2} = j \xrightarrow[j=1]{h} Q_{j1} \,|\, V_i$;

$P_1\ ?\ temp_{i,3}$;

(* $temp_{i,3} = temp_{1,h+1}$ *)

$[\ temp_{i,3} \rightarrow Q_{j2} \,|\, V_i$

$\square$

$\neg temp_{i,3} \rightarrow Q_{j3} \,|\, V_i$

]

$\square$

$temp_{i,2} > h \rightarrow temp_{i,1} := false$ (* exit loop *)

]

]

Note that if only some (but not all) processes $P_1,..,P_n$ are involved in a guarded command then the code can be simplified by removing unnecessary communication, evaluation and assignments. Instead we have given here a general transformation which is concise and results in processes P_i, $2 \leq i \leq n$ having similar code. The transformation for iterative commands (T7, R6) gives rise to 'centralized' code. In a later section we shall discuss this issue in greater detail.

This completes the transformation of any program written in a sequential guarded command language, using the translations T1-T7 and the restrictions R1-R6.

Recently, Elrad and Francez [4] have discussed CSP programs that are composed of 'communication-closed' layers. That is, $S :: [P_1 \,||\, .. \,||\, P_n]$ where P_i can be viewed as $Q_i^1 ; .. ; Q_i^k$, $1 \leq i \leq n$. The parallel programs T_j :: $[Q_1^j \,||\, .. \,||\, Q_n^j]$, $1 \leq j \leq n$ are called the *layers* of S. A layer is said to be communication-closed if for all possible computations of S, all input/output commands from that layer match with corresponding input/output commands from that layer only. It is interesting to note that the transformations given here generate a communication-closed layer for each statement of the original sequential program P.

Another feature of the transformations introduced so far is that no output guards are used in the resulting CSP programs, resulting in faster implementation of CSP [2]. In a later section we will encounter some special cases where the usage of output guards can simplify the CSP programs. It is relatively straightforward to introduce transformations to handle special cases and we consider some of them in a later section. But the transformations introduced so far are sufficient to transform any sequential program.

9. SET PARTITION EXAMPLE

In this section we illustrate the transformations introduced in the previous sections by considering the following example.

Example 1 : Given two disjoint and non-empty sets of integers S and T, $S \cup T$ has to be partitioned into two subsets A and B such that $|S| = |A|$, $|T| = |B|$ and every element of A is smaller than any element of B. A sequential program to accomplish this is given below.

$$
\begin{aligned}
P1 = \; & maxS := max(S) \, ; \\
& minT := min(T) \, ; \\
& *[\; maxS > minT \rightarrow S := S - \{maxS\} + \{minT\} \, ; \\
& \qquad\qquad\qquad\qquad T := T - \{minT\} + \{maxS\} \, ; \\
& \qquad\qquad\qquad\qquad maxS := max(S) \, ; \\
& \qquad\qquad\qquad\qquad minT := min(T) \\
&]
\end{aligned}
$$

The variables of $P1$ are partitioned into two disjoint sets as follows :

$$W_1 = \{maxS, \, S\} \; ; \; W_2 = \{minT, \, T\}$$

The translation of the sequential program $P1$ is then as follows.

$$
\begin{aligned}
T[P1] \Rightarrow \; & maxS := max(S) \, ; \\
& minT := min(T) \, ; \\
& temp_{1,1} := true \, ; \qquad\qquad \text{(* execute loop while } temp_{1,1} \text{ is true *)} \\
& *\, [temp_{1,1} \rightarrow temp_{1,2} := \text{minT} \, ; \qquad\qquad \text{(* get value *)} \\
& \qquad\qquad temp_{1,3} := (\; maxS > temp_{1,2} \,); \text{(* evaluate guard *)} \\
& \qquad\qquad [\; temp_{1,3} \rightarrow S := S - \{maxS\} + \{temp_{1,3}\} \, ; \\
& \qquad\qquad\qquad temp_{2,1} := maxS \, ; \\
& \qquad\qquad\qquad T := T - \{minT\} + \{temp_{2,1}\} \, ; \\
& \qquad\qquad\qquad maxS := max(S) \, ; \\
& \qquad\qquad\qquad minT := min(T) \, ; \\
& \qquad\qquad\qquad temp_{1,1} := true \\
& \qquad\qquad \square \\
& \qquad\qquad \neg\, temp_{1,3} \rightarrow temp_{1,1} := false \text{ (* exit loop *)} \\
& \qquad\qquad] \\
&]
\end{aligned}
$$

From the above translation of the sequential program $P1$ we obtain processes P_1, P_2 given below where the inessential *skip* statements have been removed.

```
P_1 :: maxS := max(S) ;
      temp_1,1 := true ;
      *[ temp_1,1 -> P_2 ! 1 ;                                  (* send index *)
                     P_2 ? temp_1,2 ;                           (* get value *)
                     temp_1,3 := ( maxS > temp_1,2 ) ;          (* evaluate guard *)
                     P_2 ! temp_1,3 ;                           (* send guard value *)
                     [temp_1,3 -> S := S - {maxS} + {temp_1,2} ;
                                  P_2 ! maxS ;
                                  maxS := max(S) ;
                                  temp_1,1 := true
                     □
                     ¬ temp_1,3 -> temp_1,1 := false            (* exit loop *)
                     ]
      ] ; P_2 ! 2                                   (* inform about loop termination *)


P_2 :: minT := min(T) ;
      temp_2,2 := true ;                           (* temp_2,1 already used in T[ S ] *)
      *[ temp_2,2 -> P_1 ? temp_2,3 ;                           (* index value *)
                     [temp_2,3 = 1 -> P_1 ! minT ;              (* send value *)
                                      P_1 ? temp_2,4 ;          (* obtain guard value *)
                                      [temp_2,4 -> P_1 ? temp_2,1 ;
                                                   T := T - {minT} + {temp_2,1} ;
                                                   minT := min(T)
                                      □
                                      ¬ temp_2,4 -> skip
                                      ]
                     □
                     temp_2,3 > 1 -> temp_2,2 := false          (* exit loop *)
                     ]
      ]
```

where $P1 \equiv [\ P_1\ ||\ P_2\ ]$

10. PROOF OF STRONG EQUIVALENCE

In this section we prove that the transformations introduced in the previous sections preserve strong equivalence of the original sequential program P. We first prove that the transformation of any individual statement preserves strong equivalence.

Theorem 1 : The transformation of the *skip* statement preserves strong equivalence.

Proof : For $1 \leq i \leq n$ $\quad T[skip] \,|\, V_i \Rightarrow skip$, hence the proof is trivial. □

Theorem 2 : The transformation of the assignment statement preserves strong equivalence.

Proof : Consider the assignment statement $x := exp$.

Case 1. $x \in V_j$ for some $1 \leq j \leq n$ and for $1 \leq i \leq n$ $\quad EXT[V_i,\ x := exp] = \emptyset$.

Then $T[x := exp] \,|\, V_j \Rightarrow x := exp$ and $T[x := exp] \,|\, V_i$, $i \neq j \Rightarrow skip$. Therefore, execution of $T[x := exp] \,|\, V_k$, for $1 \leq k \leq n$ terminates and preserves weak equivalence.

Case 2. if for some $1 \leq i \leq n$ $EXT[V_i,\ x := exp] = \{y_1,..,y_j\}$ and $1 \leq j$.

The translated code for this assignment statement is given by T3 as follows

$$T[x := exp] \Rightarrow temp_{i,k} \overset{j}{\underset{k=1}{:===}} y_k;$$

$$x := exp^{y_m}_{temp_{i,m}}$$

where $y_m = y_1, .., y_j$ and $temp_{i,m} = temp_{i,1}, .., temp_{i,j}$

If $y_1 \epsilon V_p$ for some $1 \leq p \leq n$ then for all $1 \leq h \leq n$ we have

$$T[temp_{i,1} := y_1] | V_h \overset{h=p}{===>} P_i \ ! \ y_1$$

$$\overset{h=i}{===>} P_p \ ? \ temp_{i,1}$$

$$\overset{h \neq p,i}{===>} skip$$

P_i and P_p will synchronize, (therefore no deadlock is possible) and hence for $1 \leq h \leq n$ $T[temp_{i,1} := y_1] | V_h$ terminates. This analysis can be repeated to cover all the introduced assignment statements. Finally, the last statement in the translated program is over variables from V_i only, and hence its termination is covered under Case 1. Weak equivalence is preserved since $temp_{i,m}$ is a copy of y_m, $1 \leq m \leq j$. □

Theorem 3 : The transformation of the iterative command preserves strong equivalence.

Proof : Consider the iterative command

$$*[\ b_j \xrightarrow[j=1]{h} S_j] \tag{5}$$

Case 0. $VARS([\ b_j \xrightarrow[j=1]{h} S_j]) \subseteq V_i$ for some i, $1 \leq i \leq n$.

Then for $1 \leq k \leq n$ T6, R5 give

$$T[*[\ b_j \xrightarrow[j=1]{h} S_j]] | V_k \overset{k=i}{===>} *[\ b_j \xrightarrow[j=1]{h} S_j]$$

$$\overset{k \neq i}{===>} skip$$

The proof of strong equivalence is trivial in this case.

Case 1. For $1 \leq j \leq h$, $VARS([\ b_j \rightarrow S_j]) \subseteq V_{m_j}$ where $1 \leq m_j \leq n$.

That is, each guarded command is over variables from only one set. Then (5) can be rewritten as at most n iterative commands as follows :

$$*[\; b_j \xrightarrow[j=1,..,h \text{ and } m_j=1]{} S_j] \qquad (A_1)$$

. . . .

:

$$*[\; b_j \xrightarrow[j=1,..,h \text{ and } m_j=n]{} S_j] \qquad (A_n)$$

that is, for $1 \leq k \leq n$ (A_k) retains only those guarded commands which have variables only from V_k. And now Case 0 holds for each of (A_1) to (A_n) and this code is exactly what is given by T6, R5.

Case 2. Let there be a process actively involved in all the boolean guards of (5) and without loss of generality let that process be P_1.

If P_1 chooses the $j'th$ (for some j, $1 \leq j \leq h$) guarded command for execution, the following fragments of code are executed.

P_1 :: ... (at the start of (5))

$P_2 \,!\, j \,;\; .. P_n \,!\, j \,;$ (* execute $b_j \rightarrow S_j$ *)

$T[temp_{1,h+1} := b_j]\,;$ (* evaluate b_j *)

$P_2 \,!\, temp_{1,h+1} \,;\; .. P_n \,!\, temp_{1,h+1} \,;$ (* inform about guard value *)

$[temp_{1,h+1} \rightarrow T[Subs(S_j)\,]\; |V_1\,;$

$$temp_{1,i} \overset{h}{\underset{i=1}{:===}} true$$

$\square$

$\neg\, temp_{1,h+1} \rightarrow temp_{1,j} := false$

] (and now control is again at the start of (5))

For $2 \leq k \leq n$

$P_k :: P_1 \ ? \ temp_{k,2}$; (* execute $b_j \rightarrow S_j$ where $j = temp_{k,2}$ *)

$T[\ temp_{1,h+1} := b_j]\ |V_k$; (* help in evaluation of b_j *)

$P_1 \ ? \ temp_{k,3}$; (* $temp_{k,3} = temp_{1,h+1}$ *)

$[\ temp_{k,3} \rightarrow T[Subs(S_j)\ ]\ |V_k$

$\square$

$\neg\ temp_{k,3} \rightarrow skip$

] (and now control is again at the start of (5))

The execution of this code amounts to the execution of the guarded command '$b_j \rightarrow S_j$' for $1 \leq j \leq h$ in the sequential program P. S_j, $1 \leq j \leq h$ will again be composed of *skip*, assignment statements; alternative and iterative commands and so we can argue inductively on the level of nesting of statements that the transformation of S_j, $1 \leq j \leq h$ will preserve strong equivalence. Weak equivalence of (5) is preserved since every process P_k, $1 \leq k \leq n$ executes $T[\ Subs(S_j)\ ]\ |\ V_k$. The transformed code is such that whenever a communication request is made by P_k on P_j then process P_j is ready to honor that communication request. Hence no deadlock can occur. Finally, to show termination, consider the case when none of the boolean guards in P_1 is true:

$$\neg\ temp_{1,1} \wedge .. \wedge \neg\ temp_{1,h} \equiv \neg\ b_1 \wedge .. \wedge \neg\ b_h$$

Then P_1 terminates the execution of (5) and executes

$P_1 :: ..\ P_2\ !\ h+1\ ;\ ..\ P_n\ !\ h+1$

while all the other processes ($2 \leq k \leq n$) execute

P_k :: .. P_1 ? $temp_{k,2}$;

$temp_{k,1} := false$ (and exits from (5))

Case 3. If Cases 0, 1, and 2 are not applicable then (5) can be rewritten as

$$temp_{i,1} := 0 ;$$

$$*[\ temp_{i,1} = 0 \wedge b_j \xrightarrow[j=1]{h} S_j] \tag{6}$$

for any one i, $1 \le i \le n$. (Note that $temp_{i,1}$ is not altered or used in S_j, $1 \le j \le h$.) Now (6) is covered by Case 2 and hence strong equivalence is preserved. □

Theorem 4 : The transformation of the alternative command preserves strong equivalence.

Proof : Consider the alternative command

$$[\ b_j \xrightarrow[j=1]{h} S_j] \tag{7}$$

Case 1. $VARS([\ b_j \xrightarrow[j=1]{h} S_j]) \subseteq V_i$ for some i, $1 \le i \le n$.

Then for $1 \le k \le n$ we have

$$T\ [\ [b_j \xrightarrow[j=1]{h} S_j\ ]\ ]\ |V_k \overset{k=i}{===>} [b_j \xrightarrow[j=1]{h} S_j]$$

$$\overset{k \ne i}{===>} skip$$

The execution of $T\ [\ [\ b_j \xrightarrow[j=1]{h} S_j]\ ]\ |V_i$, $1 \le i \le n$ is the same as the execution of (7) in P and hence strong equivalence is preserved.

Case 2. If all the variables in the alternative command do not belong to some set V_j, $1 \le j \le n$ then (7) is rewritten as

$$temp_{i,1} := 0 \;;$$
$$*[\ temp_{i,1} = 0 \wedge b_j \xrightarrow[j=1]{h} S_j;\ temp_{i,1} := 1\ ]\;;$$
$$[\ temp_{i,1} > 0 \rightarrow skip] \tag{8}$$

for any one i, $1 \leq i \leq n$ and (7), (8) are equivalent. Strong equivalence is preserved since Theorems 2 and 3 preserve the strong equivalence of the assignment statement and the iterative command. Finally, the alternative command in (8) is over variables from only one set (V_i) and hence is covered by Case 1. □

We have proved that the transformation of any *skip*, assignment, alternative and iterative statement preserves strong equivalence. We must also show that the transformation preserves strong equivalence of the entire original program.

Theorem 5 : Consider any sequential program P written in GC with its variables partitioned into n disjoint sets $W_1, .., W_n$. Then P and S :: [$T[\ P\ ]\,|V_1\ ||$ .. $||$ $T[\ P\ ]\,|V_n]$ are strongly equivalent where for $1 \leq i \leq n$ $V_i = W_i \cup \{temp_{i,j}\ |1 \leq j \leq u_i\}$.

Proof : The proof follows directly from the Theorems 1,2,3,4 and the fact that the transformation preserves sequential composition. □

11. REDUCING CENTRALIZATION

Here we examine cases for which no leader process is required when transforming iterative commands. There are two aspects that have to be considered :

1. how to ensure that each involved process chooses the same guard, and
2. how to ensure termination of the iterative command.

At times, the choice of a guard in one process will ensure that the right guard is chosen by all the other processes.

Example 2 : Consider the following sequential program.

$P2 = *[a > c \rightarrow S_1 \square \; b > d \rightarrow S_2]$

and let the variables be partitioned as follows

$$W_1 = \{a, b\} \; ; \; W_2 = \{c, d\}$$

Then the following structure for the two processes is adequate for preserving weak equivalence :

$P_1 :: *[\; P_2 \; ? \; temp_{1,1} \rightarrow P_2 \; ! \; a \; ;$
if first guard in $P2$ is true then execute from S_1

$\square$
$P_2 \; ! \; b \rightarrow P_2 \; ? \; temp_{1,2} \; ;$
if second guard in $P2$ is true then execute from S_2

]

$P_2 :: *[\; P_1 \; ! \; c \rightarrow P_1 \; ? \; temp_{2,1};$
if first guard in $P2$ is true then execute from S_1

$\square$
$P_1 \; ? \; temp_{2,2} \rightarrow P_1 \; ! \; d \; ;$
if second guard in $P2$ is true then execute from S_2

]

(Note we are now making use of output guards.) In this case it is evident that when the first (second) guard of $P2$ is chosen by P_1 then P_2 also chooses the first (second) guard of $P2$. Since there is no leader process, both processes evaluate the boolean guards and it follows that both the processes obtain the same result. Hence in this case, checking of indices is not required. (Note that in the above example, termination is not ensured). It is straightforward to see that the checking of indices can always be done away with if there is only one guard in an iterative command.

One way of ensuring termination of an iterative command without a leader process is as follows : An exchange of values must be done so that, at the start of the iterative command, each process can test all the original boolean guards in that iterative command; this corresponds to *factoring out* the exchange of values.

Example 3 : Consider the following sequential program and partitioning of variables.

$P3 = *[\ x > y \rightarrow S\ ] \quad W_1 = \{x\}\ ;\ W_2 = \{y\}$

The following structure for two processes is adequate for preserving strong equivalence

$$
\begin{array}{lll}
P_1 :: & P_2\ !\ x\ ; & \\
 & P_2\ ?\ temp_{1,1}\ ; & \\
 & *[\ x > temp_{1,1} \rightarrow & T[\ S\ ]\,|\,V_1\ ; \\
 & & P_2\ !\ x\ ; \\
 & & P_2\ ?\ temp_{1,1} \\
 & \quad] &
\end{array}
$$

$$
\begin{array}{lll}
P_2 :: & P_1\ ?\ temp_{2,1}\ ; & \\
 & P_1\ !\ y\ ; & \\
 & *[\ temp_{2,1} > y \rightarrow & T[\ S\ ]\,|\,V_2\ ; \\
 & & P_1\ ?\ temp_{2,1}\ ; \\
 & & P_1\ !\ y \\
 & \quad] &
\end{array}
$$

where $P3 \equiv [\ P_1\ ||\ P_2\ ]$.

Definition 7 : The ***distributed termination convention*** allows an iterative command in a process to terminate if for each guard either of the following conditions is met :

(i) the boolean component is false

or (ii) the source named in the guard has terminated □

Further, since in $P3$ the iterative command is the last statement involving participation between P_1 and P_2 , the distributed termination convention of CSP can be used. To use the distributed termination convention here, say for the second process, first of all the guard in the second process must include an input command. To satisfy this requirement, we allow only the first process to evaluate the guard and if the guard is true it communicates with the second process otherwise it terminates. Consequently, the second process waits for a communication from the first process and if the first process has terminated then the second process terminates by distributed termination convention. This leads to the following two processes.

$$
\begin{array}{l}
P_1' :: P_2 \ ?\ temp_{1,1} \ ; \\
\quad *[\ x > temp_{1,1} \rightarrow P_2 \,!\, x \ ; \\
\qquad\qquad\qquad\qquad T[\ S\ ]\,|V_1 \ ; \\
\qquad\qquad\qquad\qquad P_2 \ ?\ temp_{1,1} \\
\quad\ ]
\end{array}
$$

$$
\begin{array}{l}
P_2' :: P_1 \,!\, y \ ; \\
\quad *[\ P_1 \ ?\ temp_{2,1} \rightarrow T[\ S\ ]\,|V_2 \ ; \\
\qquad\qquad\qquad\qquad P_1 \,!\, y \\
\quad\ ]
\end{array}
$$

where $P2 \equiv [\ P_1' \ || \ P_2'\ ]$.

With the above discussion, we can now introduce transformations to handle some special cases.

Simplification 1 : If the iterative command has only one guarded command, the guard can be evaluated (obtained) just before the start of the iterative command and at the end of the guarded command list. This is achieved by the transformation T7-1 and R6-1 as given below. □

Transformation : Define transformation as follows.

T7-1. $T[\ *[\ b \rightarrow S\ ]\ ]$

$\Rightarrow T[\ temp_{j,1} \underset{j=1}{\overset{n}{:===}} b\ ];$ $\quad$ (* n copies of b *) $\quad$ (9)

$*[\ \underset{j=1}{\overset{n}{\wedge}} temp_{j,1} \rightarrow T\ [\ S\ ];$ $\quad$ (* n copies of b as guard *)

$T[\ temp_{j,1} \underset{j=1}{\overset{n}{:===}} b\ ]$

]

or

$T[\ *[\ b \rightarrow S\ ]\ ]$

$\Rightarrow T[\ temp_{1,1} := b\ ];$ $\quad$ (* 1 copy of b *) $\quad$ (10)

$temp_{j,1} \underset{j=2}{\overset{n}{:===}} temp_{1,1};$ $\quad$ (* $n-1$ copies of $temp_{1,1}$ *)

$*[\ \underset{h=1}{\overset{n}{\wedge}} temp_{h,1} \rightarrow T[S];$ $\quad$ (* n copies of b as guard *)

$T[\ temp_{1,1} := b\ ];$

$temp_{j,1} \underset{j=2}{\overset{n}{:===}} temp_{1,1}$

]

R6-1. if b can be rewritten as $b_1 \wedge .. \wedge b_n$ where for $1 \leq i \leq n$, b_i is a boolean expression over variables only from V_i then define

$$*[\ b \rightarrow S\ ]\,|\,V_i \;\Rightarrow\; *[\ b_i \rightarrow S\,|\,V_i]$$

Note that in the above restriction, of the original n copies of the guard b each process just retains its 'own' copy of the guard b.

If translation is done according to (9), notice that in the distributed program each process will individually evaluate the guard. In the worst case this may lead to a situation wherein each process communicates with every other process. If the translation is done according to (10), process P_1 will evaluate the guard and send the guard value to all the other processes. The worst case in (10) would amount to requiring that P_1 communicates with all the other processes.

(9) and (10) indicate two of the many different possible requirements that can be placed on process interconnection by communication channels for an iterative command. The choice between these possibilities should minimize the interconnection requirement for the entire program P.

The set partition program $P1$ introduced earlier meets the requirements for the usage of T7-1 and R6-1. The transformation of the set partitioning program can now proceed as follows. We first obtain the following translation of the sequential program $P1$.

$$
\begin{array}{lll}
T[P1] \Rightarrow & maxS := max(S) \ ; & \\
& minT := min(T) \ ; & \\
& temp_{1,1} := minT \ ; & \text{(* using T7-1, (10) *)} \\
& temp_{1,2} := (maxS > temp_{1,1}) \ ; & \text{(* evaluate guard *)} \\
& temp_{2,1} := temp_{1,2} \ ; & \text{(* set variable to guard value *)} \\
& *[\ temp_{1,2} \wedge temp_{2,1} \rightarrow S := S - \{maxS\} + \{temp_{1,1}\} \ ; & \\
& \qquad temp_{2,2} := maxS \ ; & \\
& \qquad T := T - \{minT\} + \{temp_{2,2}\} \ ; & \\
& \qquad maxS := max(S) \ ; & \\
& \qquad minT := min(T) \ ; & \\
& \qquad temp_{1,1} := minT \ ; & \\
& \qquad temp_{1,2} := (maxS > temp_{1,1}) \ ; & \text{(* evaluate guard *)} \\
& \qquad temp_{2,1} := temp_{1,2} & \text{(* set var. to guard value *)} \\
&] &
\end{array}
$$

The sequential program above can now be restricted to yield process P_1.

$$
\begin{array}{ll}
P_1 :: maxS := max(S) \ ; & \\
\quad P_2 \ ? \ temp_{1,1} \ ; & \\
\quad temp_{1,2} := (maxS > temp_{1,1}) \ ; & \text{(* evaluate guard *)} \\
\quad P_2 \ ! \ temp_{1,2} \ ; & \text{(* send guard value *)} \\
\quad *[\ temp_{1,2} \rightarrow S := S - \{maxS\} + \{temp_{1,1}\} \ ; & \\
\qquad P_2 \ ! \ maxS \ ; & \\
\qquad maxS := max(S) \ ; & \\
\qquad P_2 \ ? \ temp_{1,1} \ ; & \\
\qquad temp_{1,2} := (maxS > temp_{1,1}) \ ; & \text{(* evaluate guard *)} \\
\qquad P_2 \ ! \ temp_{1,2} & \text{(* send guard value *)} \\
\quad] &
\end{array}
$$

The process P_2 is as follows.

$P_2 :: minT := min(T)$;
$\quad P_1 \,!\, minT$;
$\quad P_1 \,?\, temp_{2,1}$; (* obtain guard value *)
$\quad *[\ temp_{2,1} \rightarrow P_1 \,?\, temp_{2,2}$;
$\qquad T := T - \{minT\} + \{temp_{2,2}\}$;
$\qquad minT := min(T)$;
$\qquad P_1 \,!\, minT$;
$\qquad P_1 \,?\, temp_{2,1}$ (* obtain guard value *)
$\quad]$

where $P1 \equiv [\ P_1 \ ||\ P_2\ ]$.

The evaluation (obtaining) of guard value can also be minimized with respect to the execution time. For example, if P_i, $1 \leq i \leq 4$ are involved in the iterative command $*[\ b \rightarrow S\ ]$ then the translation

$T[\ temp_{1,1} := b\ ]$;
$temp_{2,1} := temp_{1,1}$;
$temp_{3,1} := temp_{1,1}$;
$temp_{4,1} := temp_{1,1}$;

would require more execution time than

$T[\ temp_{1,1} := b\ ]$;
$temp_{2,1} := temp_{1,1}$;
$temp_{3,1} := temp_{2,1}$;
$temp_{4,1} := temp_{1,1}$;

Simplification 2 : If the sequential program $P \equiv Q_1;\ *[\ b \rightarrow S\ ];\ Q_2$ where for $1 \leq i \leq n$ $EXT[\ V_i,\ Q_2\ ] = \emptyset$ then we can make use of the distributed termination convention of CSP. □

Note that if for some j, $1 \leq j \leq n$ $EXT[\ V_j,\ Q_2\ ] \neq \emptyset$ then P_j will require some communication after the execution of the command $*[\ b \rightarrow S\ ]$ is completed, which could be in contradiction to using distributed termination convention for the iterative command.

Transformation

T7-1L. if for some $1 \leq k \leq n$ $EXT[\ V_k, temp_{k,1} := b\ ] = \{y_1, .., y_h\}$ and the sequential program is $P \equiv Q_1;\ *[\ b \rightarrow S\ ];Q_2$ where for $1 \leq i \leq n$ and $i \neq k$ $EXT[\ V_i, Q_2\ ] \cap V_k = \emptyset$ then define

$$
\begin{aligned}
T[Q_1;\ *[\ b \rightarrow S\ ]; Q_2\ ] \Rightarrow\ & T[\ Q_1\ ]\ ; \\
& temp_{k,j} \overset{h}{\underset{j=1}{:====}} y_j\ ; \\
& *[\ b^{y_j}_{temp_{k,j}} \rightarrow T[S]\ ; \\
& \qquad\qquad temp_{k,j} \overset{h}{\underset{j=1}{:====}} y_j \\
&];\ T[\ Q_2\ ]
\end{aligned}
$$

where $y_j = y_1 ,.., y_h$ and $temp_{k,j} = temp_{k,1} ,.., temp_{k,h}$

R6-1L. if the translated program $T[\ P\ ] \equiv Q_1;\ *[\ b \rightarrow S\ ];\ Q_2$ where for some j, $1 \leq j \leq n$ $VARS(temp_{j,1} := b) \subseteq V_j$ and for $1 \leq k \leq n$ $k \neq j$ $EXT[\ V_k, Q_2\ ] \cap V_j = \emptyset$ then for $1 \leq i \leq n$ define

$$
\begin{aligned}
(Q_1;\ *[\ b \rightarrow S\ ];\ Q_2)\ |V_i \overset{i=j}{====>}\ & Q_1 | V_i\ ; \\
& *[\ b \rightarrow P_1\ !\ \textit{`dummy variable'}\ ; \\
& \qquad \cdots \\
& \qquad P_{j\text{-}1}\ !\ \textit{`dummy variable'}\ ; \\
& \qquad P_{j+1}\ !\ \textit{`dummy variable'}\ ; \\
& \qquad \cdots \\
& \qquad P_n\ !\ \textit{`dummy variable'}\ ; \\
& \qquad T[\ S\ ] | V_i \\
&]\ ;\ Q_2 | V_i
\end{aligned}
$$

$$\overset{i \neq j}{===>} Q_1 | V_i ;$$
$$*[\ P_j\ ?\ \textit{'dummy variable'} \rightarrow T[\ S\ ]\ | V_i\ ] ;$$
$$Q_2 | V_i$$

where 'dummy variable' is any arbitrary message. In fact, as we shall see below, if in $T[\ S\ ]\ |V_i$ communication from P_1 to P_i is required then this dummy communication can be done away with. The use of T7-1L and R6-1L for an iterative command is therefore context dependent. The set partition program $P1$ meets the requirements of the transformation T7-1L and R6-1L and hence the transformation of $P1$ can now proceed as follows. We first apply the translation step.

$$
\begin{array}{l}
T[\ P1\ ] \Rightarrow maxS := max(S) ; \\
\qquad minT := min(T) ; \\
\qquad temp_{1,1} := minT ; \\
\qquad *[\ maxS > temp_{1,1} \rightarrow S := S - \{maxS\} + \{temp_{1,1}\} ; \\
\qquad \qquad temp_{2,1} := maxS ; \\
\qquad \qquad T := T - \{minT\} + \{temp_{2,1}\} ; \\
\qquad \qquad maxS := max(S) ; \\
\qquad \qquad minT := min(T) ; \\
\qquad \qquad temp_{1,1} := minT \\
\qquad]
\end{array}
$$

The restriction of the above sequential program leads to process P_1 given below.

$$
\begin{array}{l}
P_1 :: maxS := max(S) ; \\
\qquad P_2\ ?\ temp_{1,1} ; \\
\qquad *[\ maxS > temp_{1,1} \rightarrow P_2\ !\ \textit{'dummy variable'} ; \\
\qquad \qquad S := S - \{maxS\} + \{temp_{1,1}\} ; \\
\qquad \qquad P_2\ !\ maxS ; \\
\qquad \qquad maxS := max(S) ; \\
\qquad \qquad P_2\ ?\ temp_{1,1} \\
\qquad]
\end{array}
$$

Process P_2 is as follows.

$$
\begin{aligned}
P_2 :: \; & minT := min(T) ; \\
& P_1 \; ! \; minT \; ; \\
& *[\; P_1 \; ? \; \text{'dummy variable'} \rightarrow P_1 \; ? \; temp_{2,1} \; ; \\
& \qquad\qquad\qquad\qquad\qquad\quad T := T - \{minT\} + \{temp_{2,1}\} \; ; \\
& \qquad\qquad\qquad\qquad\qquad\quad minT := min(T) ; \\
& \qquad\qquad\qquad\qquad\qquad\quad P_1 \; ! \; minT \\
& \;\;]
\end{aligned}
$$

where $P1 \equiv [\; P_1 \; || \; P_2 \;]$

The exchange of dummy variables can be done away in P_1 and P_2 above to give the following two processes.

$$
\begin{aligned}
P_1' :: \; & maxS := max(S) ; \\
& P_2 \; ? \; temp_{1,1} \; ; \\
& *[\; maxS > temp_{1,1} \rightarrow S := S - \{maxS\} + \{temp_{1,1}\} \; ; \\
& \qquad\qquad\qquad\qquad\;\; P_2 \; ! \; maxS \; ; \\
& \qquad\qquad\qquad\qquad\;\; maxS := max(S) ; \\
& \qquad\qquad\qquad\qquad\;\; P_2 \; ? \; temp_{1,1} \\
& \;\;]
\end{aligned}
$$

$$
\begin{aligned}
P_2' :: \; & minT := min(T) ; \\
& P_1 \; ! \; minT \; ; \\
& *[\; P_1 \; ? \; temp_{2,1} \rightarrow T := T - \{minT\} + \{temp_{2,1}\} \; ; \\
& \qquad\qquad\qquad\quad\; minT := min(T) ; \\
& \qquad\qquad\qquad\quad\; P_1 \; ! \; minT \\
& \;\;]
\end{aligned}
$$

where $P1 \equiv [\; P_1' \; || \; P_2' \;]$

The code obtained above is elegant and concise. In this section we have sketched in some detail how T7 and R6 can be modified to deal with special cases of iterative commands. The treatment of the possible special cases is by no means complete but should serve as a guideline.

We have included this section to indicate how the general transformation for iterative commands should be simplified to cover different special cases.

12. NESTED CONCURRENCY

In this section we consider briefly how *nested concurrency* may be achieved by the transformations. As an illustration of nested concurrency consider the following sequential program.

$$P \equiv Q_1 ; \;\; *[\; b_j \xrightarrow[j=1]{h} S_j \;] ; \;\; Q_2 \tag{11}$$

and assume that the variables of P are partitioned into n disjoint sets W_i, $1 \le i \le n$ so that $VARS(\; *[\; b_j \xrightarrow[j=1]{h} S_j \;]\;) \subseteq W_1$. By the transformations introduced so far we will obtain a strongly equivalent system of n communicating processes of the following form.

$$\begin{aligned} S :: [\;\; & P_1 :: Q_{11} ; \;\; *[\; b_j \xrightarrow[j=1]{h} S_j \;] ; \;\; Q_{21} \\ & \| \; P_2 :: Q_{12} ; \; Q_{22} \\ & \vdots \\ & \| \; P_n :: Q_{1n} ; \;\; Q_{2n} \;] \end{aligned} \tag{12}$$

Note that in P_1 in (12) the fragment of code $*[\; b_j \xrightarrow[j=1]{h} S_j \;]$ does not involve any communication statements and hence can be viewed as sequential code. For that reason this fragment of code can be transformed to give an equivalent system of communicating processes. For instance, this fragment of code is over the variables W_1 and if this set of variables is partitioned into 2 disjoint sets W_{11}, $W_{1,2}$ then our existing transformation would yield

$$*[\ b_j \xrightarrow[j=1]{h} S_j\ ] \equiv S1 :: [\ P_{1,1} \mid\mid P_{1,2}\ ] \tag{13}$$

where $P_{1,1}$ and $P_{1,2}$ are two communicating processes. Now

$$\begin{aligned} P \equiv S' :: [\ \ & P_1 :: Q_{11};\ \ S1; Q_{21} \\ & \mid\mid P_2 :: Q_{12}; Q_{22} \\ & \vdots \\ & \mid\mid P_n :: Q_{1n};\ \ Q_{2n}\ ] \end{aligned} \tag{14}$$

This is an example of nested concurrency and can easily be achieved by suitably extending some of the existing transformations. In this particular case, the transformation would proceed as normal to obtain (12) and (13) and then a simple substitution is required to obtain (14).

In the above discussion we have indicated one of the many situations wherein nested concurrency can be generated. The set of transformations can be easily augmented to introduce nested concurrency appropriately but we shall not explicitly present such transformations here.

13. PARTITIONING OF VARIABLES AND SPEED-UP

In this section we show how the execution time of a distributed CSP program can be reduced. Since every communication requires synchronization, parallelism can be increased (and hence execution time can be reduced) by reducing communication. The number of communications required can be reduced by examining the interdependencies between the variables in the sequential program. If variables in a group are closely related, i.e. if a change in the value of one variable makes it necessary for the other variables in the group to be updated, then this group of variables should be kept in a singe set W_i. Another guideline for partitioning variables is to ensure that all the variables involved in an alternative command in some set W_i, for any $1 \leq i \leq n$. (This does not imply that variables from two alternative commands should belong to the same set W_i.)

We now consider an important way in which execution can be speeded up. Consider the iterative command

$$*[\ b_j \xrightarrow[j=1]{h} S_j\ ] \tag{15}$$

If for all $1 \leq j \leq h$ $VARS([b_j \rightarrow S_j]) \subseteq V_{m_j}$ where $1 \leq m_j \leq n$, (which is Case (A) of the transformation of an iterative command) then no synchronization is required in the execution of (15). Such a structure often leads to a speed up. For example, if in (15) $h = n$ and $VARS([b_j \rightarrow S_j]) \subseteq V_j$, $1 \leq j \leq n$, then we may be able to achieve a speed up by a factor n in execution.

One obvious guideline for partitioning of variables is to ensure that Case (A) holds for iterative commands. Sometimes, even the number of sets into which the original set of variables should be partitioned is indicated; for example, if each iterative command has h guarded commands then we may want to partition the variables into h sets. Virtually every statement in the original sequential program indicates a preferred partitioning of variables. If such a partitioning of variables is not practical or is in conflict with the requirement of the other statements of the sequential program then one may also consider allowing the partitioning to be ***dynamic***.

For example, if the sequential program P where

$$P \equiv Q_1 \,;\, R \;;\, Q_2$$

is to be transformed into a distributed program $S :: [\, P_1 \,||\, P_2 \,]$. Note that in this case, $P_i :: T[\, Q_1 \,] | V_i \,;\; T[\, R \,] | V_i \,;\; T[\, Q_2 \,] | V_i \,;\; i = 1, 2$. Then it is possible that a partitioning W_1, W_2 is desirable for the execution of the sequential programs Q_1 and Q_2, but a conflicting partitioning $W_1{}'$, $W_2{}'$ is desired for the execution of R. In particular, let $W_1 = \{a, b, c\}$, $W_2 = \{x, y, z\}$, $W_1{}' = \{a, b\}$ and $W_2{}' = \{c, x, y, z\}$ (this could arise if c is not used in $T[\, R \,] | V_1$ but is used in $T[\, R \,] | V_2$). In that case the sequential program P can be rewritten as

$$P \equiv Q_1 \,;\;\; temp_{2,1} := \; c \,;\, R^{c}_{temp_{2,1}} ;\;\; c \; := \;\; temp_{2,1} \,;\;\; Q_2$$

In effect, the variable c will be ***temporarily loaned*** to $T[\, R \,] | V_2$. Naturally, since there is some communication overhead involved in this strategy, it is not always advisable to have the best possible partitioning of variables for each individual statement in the sequential program P.

In the following example we consider the transformation of a program which results in execution speed up by a factor of n.

Example 4 : n pairs (x_i, y_i), $1 \leq i \leq n$ of positive, non-zero numbers are given and the greatest common divisor (GCD) of each pair is to be calculated. A sequential program for this can be written as follows :

$$
\begin{array}{l}
P4 = *[\ x_1 \neq y_1 \rightarrow [\ x_1 > y_1 \rightarrow x_1 := x_1 - y_1 \\
\qquad\qquad\qquad\qquad \square \\
\qquad\qquad\qquad\qquad y_1 > x_1 \rightarrow y_1 := y_1 - x_1 \\
\qquad\qquad\qquad\qquad] \\
\quad \dots \\
\quad x_n \neq y_n \rightarrow \dots \\
\quad]
\end{array}
$$

The variables can be partitioned so that $W_i = \{x_i,\ y_i\}$, $1 \leq i \leq n$. By T6 and R5 we obtain

For $1 \leq i \leq n$

$$
\begin{array}{l}
P_i :: *[\ x_i \neq y_i \rightarrow [\ x_i > y_i \rightarrow x_i := x_i - y_i \\
\qquad\qquad\qquad\qquad \square \\
\qquad\qquad\qquad\qquad y_i > x_i \rightarrow y_i := y_i - x_i \\
\qquad\qquad\qquad\qquad] \\
\qquad]
\end{array}
$$

where $P4 \equiv S :: [\ P_1 \ || \ .. \ || \ P_n\ ]$

In this example, the transformation of the sequential program $P4$ into a distributed program S results in a possible speed-up in execution time by a factor of n.

14. RELATED WORK

Sintzoff [15] considers the problem of transforming a class of programs into a logically equivalent but 'more distributed' programs. Keller [8] investigates how parallelism can be effectively introduced in a framework of a number of automata (each executing some operation) computing concurrently. The automata can interact with each other via a set of memory cells. Lengauer and Hehner [9] and Lengauer [10] start with a sequential program and identify 'semantic relations' between parts of a program that may allow relaxation in the sequencing of the program's operations. So in all the above [8, 9, 10, 15], parts of the program that can be concurrently executed are identified and exploited.

Jackson [7] advocates viewing a system as a model of reality as opposed to viewing it functionally. The basic form of the model used by him is a network of processes, with one process for each independently active entity in the real world. Since this typically leads to a large number of processes he also considers how a class of processes may be compacted into a single process.

Manna and Wolper [12] separate a concurrent program into two parts : a *synchronization* part and a *functional* part. The synchronization part enforces the necessary constraints on the relative timing of the execution of the different processes. The functional part manipulates the data and performs the computation required of the program. Under the constraints that only non-terminating processes are considered and that the only communications allowed are between a distinguished process and the other processes; they consider the synthesis of the synchronization part by using propositional temporal logic.

Shrira and Francez [14] develop a methodology for transforming sequential recursive algorithms into distributive algorithms in CSP like notation. This is done under the assumption that in the sequential algorithm the program segments between recursive calls have a distributive implementation.

In [13] we considered the transformation of a sequential program P into a distributed program S. The transformations in [13] were much simpler than those derived in this paper since in [13] only the weak equivalence of P and S was ensured.

15. FUTURE WORK

The transformation of a program is achieved by considering individual statements. Though the execution time of the individual transformed statements can be minimized, as indicated in the previous sections, a much more elaborate approach must be taken to minimize the execution time of the entire program. Similarly, the minimization of communication channels required must also be done at the program level.

A small number of translations and restrictions are sufficient for transforming any arbitrary sequential program. But, as we have seen, additional translations and restrictions must be introduced for special cases to achieve elegant as well as execution efficient programs. Therefore, for the automated transformation of sequential programs to become viable, a large repertoire of translations and restrictions must be created.

In practice, the transformation of an iterative command will normally result in 'centralized' program code. This should be avoided whenever possible and we are working on sufficient and necessary conditions to achieve this. This will entail expansion of the repertoire of translations and restrictions.

CONCLUSIONS

In this paper we have defined a constructive transformation which can be used to derive strongly equivalent distributed programs from sequential programs. This transformation works on any sequential program written in a sequential guarded command language to yield a system of communication processes written in CSP. We have also considered the conditions under which a speed-up can be achieved by these transformations. The transformation given here can be easily automated.

ACKNOWLEDGMENT

I would like to take this opportunity to thank K.R. Apt, N. Francez, C.A.R. Hoare, M. Joseph, K. Lodaya, W.P. de Roever and M. Sintzoff for detailed comments on earlier versions of this paper; and to the referees for their interest and helpful suggestions.

REFERENCES

1. Apt, K.R., Francez, N., and de Roever, W.P. A proof system for communicating sequential processes. *ACM TOPLAS 2* (1980), 359-385.

2. Bernstein, A.J. Output guards and nondeterminism in 'Communicating Sequential Processes'. *ACM TOPLAS 2* (1980), 234-238.

3. Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

4. Elrad, T., and Francez, N. Decomposition of distributed programs into communication-closed layers. to appear in *Sci. Comput. Programming.*

5. Gries, D. *The Science of Programming.* Springer-Verlag, New York, 1981.

6. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21* (1978) 666-677.

7. Jackson, A.M. Information systems : modelling, sequencing and transformations. *Third Int. Conf. on Software Eng.* 1978, 72-81.

8. Keller, R.M. Parallel program schemata and maximal parallelism I : fundamental results. *Journal ACM 20* 1973, 514-537.

9. Lengauer, C., and Hehner, E.C.R. A methodology for programming with concurrency : an informal presentation. *Sci. Comput. Programming 2* (1982), 1-18.

10. Lengauer, C. A methodology for programming with concurrency: the formalism. *Sci. Comput. Programming 2* (1982), 19-52.

11. Manna, Z., and Waldinger, R.J. Toward automatic program synthesis. *Commun. ACM 14* (1971), 151-165.

12. Manna, Z., and Wolper, P. Synthesis of communicating processes from temporal logic specifications. *ACM TOPLAS 6* (1984), 68-93.

13. Moitra, A. Synthesis of communicating processes. *Conf. on Principles of Distributed Computing,* Montreal, August 1983, 123-130.

14. Shrira, L., and Francez, N. Distributed k-selection : from a sequential to a distributed algorithm. *Conf. on Principles of Distributed Computing,* Mont-

real, August 1983, 143-153.

15. Sintzoff, M. Principles for distributed programs. In *Semantics of Concurrent Computation*, Lecture Notes in Computer Science 70, G. Kahn (Ed.), Springer-Verlag, New York, 1979, 337-347.