Science of Computer Programming 2 (1982) 155–173
North-Holland

DECOMPOSITION OF DISTRIBUTED PROGRAMS INTO COMMUNICATION-CLOSED LAYERS

Tzilla ELRAD*
Illinois Institute of Technology, Chicago, IL 60616, U.S.A.

Nissim FRANCEZ**
Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

Communicated by M. Sintzoff
Received December 1982
Revised February 1983

Abstract. The safe decomposition of a distributed program into communication closed layers is suggested as a superstructure of its decomposition into a collection of communicating processes. This decomposition may simplify the analysis of a distributed program, as is exemplified by examples of program verification. A programming language construct to enforce safety of a decomposition is introduced. The application to systematic construction of distributed programs is also shown.

1. Introduction

The traditional decomposition of a distributed program is into a collection of communicating processes (or tasks), where processes are either sequential or contain nested concurrency. This decomposition prevails in the existing languages for concurrent and distributed programming. It induces a natural two-level definition method of its semantics: an a priori semantics is given to whole processes, independently of each other, and then the separate process meanings are bound to a joint meaning. In a denotational semantics context this can be seen, e.g., in [14], or [6, 8]. In the axiomatic context it can be observed in [15], [2], [12], or [3], and in a temporal context in [9]. An alternative approach defines the meaning of the whole program in the same level. This can be seen most often in the operational semantics, given by a centralized, non-deterministic interpreter, e.g., in [1], or in [4] in the context of weakest precondition semantics.

In this paper we present a different approach, which suggests an alternative decomposition as a super structure over a decomposition into processes.

* Work was partially supported by NSF Grant MCS-80-17577 at Syracuse University.
** World Trade Visiting Scientist at IBM T.J. Watson Research Center, on sabbatical leave from Technion, Haifa, Israel.

According to this decomposition, parts of processes that interact with each other (and are not interacting with other parts) are grouped together into layers. We introduce sufficient conditions under which the meaning of the program is such that it is equivalent to synchronizing all the processes in a given program at layer boundaries.

Such a decomposition may simplify the analysis of distributed programs, as shown by examples considering formal program verification. The method is useful in many other contexts of program analysis, as well as for systematic construction of distributed programs, as also shown by an example. Another attempt for the simplification of the analysis of concurrent programs (using shared variables) can be found in [13]. There, sufficient conditions are introduced to the equivalence of a concurrent program and its 'reduction', causing a program section to be executed atomically. The approach here is completely different, and is also used for (distributed) program construction.

We shall use the notation and terminology of CSP [11] for the concreteness of the discussion. However, the approach is more general in its nature.

The paper is organized as follows: In Section 2, we define the concepts related to the safe decomposition method and consider simple decompositions, which do not cross loop boundaries. In Section 3, we consider more complicated decompositions, in which loops are also decomposed. In Section 4, we introduce a programming language construct for the enforcement of safety of a layer. Section 5 contains an example of using the concepts developed for a systematic construction of a distributed program.

2. Decomposition into simple communication-closed layers

In this section, an alternative decomposition of a distributed program $P = [P_1 \| \ldots \| P_n]$ is suggested. We consider here decompositions called *simple*, which will not cross the boundaries of iterations. In Section 3, the decomposition is extended to loops also.

We first define an equivalence relation which the new decomposition will be shown to preserve. We assume that a *communication graph* G_P underlies P, determining potential communication capabilities among the processes of P, located at the nodes of G_P. The edges of G_P represent communication channels. For CSP, an edge connects P_i and P_j iff both contain matching i/o commands addressing each other. For simplicity, we assume all messages are of the same type. For Ada, an edge would connect P_i to P_j if one of them can call an entry in the other. Similar graphs can be constructed for any other language in which processes may communicate.

Two distributed programs P and P' are said to be *compatible* if $G_P = G_{P'}$. The computations of a distributed program P can be characterized by their externally observable behavior, consisting of two elements: a (global) *state transformation* *relation* R_P (capturing also non-termination), and a set of *communication histories* H_P, describing all the possible communications arising during the computation of P. The exact description of H depends on the communication primitives used. We shall take a communication history to be a sequence of triples $\langle i, j, m\rangle$, where i, j are node indices in G_P, and m is a message transferred from node i to node j.

Definition. Two compatible distributed programs P and P' are equivalent, $P \equiv P'$, iff $R_P = R_{P'}$. Thus, equivalent programs display identical state transformations, but may differ in their communication histories.

2.1. Simple communication-closed layers

In order to decompose a program P, we first represent each process P_i as

$$P_i :: S_1^i; \ldots; S_d^i, \quad i = 1, \ldots, n.$$

Here, the S_j^i's are any program segments, including the *empty* segment, denoted by Λ. The introduction of Λ components allows for d to be uniform over all $i = 1, \ldots, n$. We call d the *depth* of the decomposition. This decomposition of a process is *simple*, since it does not cross the borders of compound statements, e.g., iterations, or even selections.

Definition. (1) For $1 \leq j \leq d$, the jth simple layer of P, denoted by L_j, consists of $L_j :: [S_j^1 \| \ldots \| S_j^n]$.

(2) The decomposition of P into simple layers (of depth d) is $P' :: L_1; \ldots; L_d$. Note that the communication graph of P' is the union of the graphs of the layers.

In the rest of this section, we shall use 'layer' to mean 'simple layer'; more general layers are discussed in Section 3.

Obviously, there exist many decompositions of a distributed program into layers. We shall be interested in such decompositions that preserve the state transformation behavior of the program, i.e., are equivalent to it. First, we note the following trivial fact, following immediately from the definitions above.

Lemma. *Every program P is compatible with all its decompositions into layers.*

Next, we define a sufficient condition for equivalence of such a decomposition to the original program.

Definition. A layer L_j of a distributed program P is *communication-closed* iff under *no* execution of P a communication command in some S_j^i will communicate with a communication command in some $P_{i'}$, not belonging to $S_{j}^{i'}$.

In other words, in any communication in which one of the involved parties belongs to the considered layer, so does the second party. For readers familiar with the terminology used in the axiomatic definition of CSP in [2], we mention an alternative equivalent formulation of the definition: *no* two syntactically matching commands not both belonging to a given layer L_j are semantically matching.

Definition. A decomposition P' of a distributed program P into layers is *safe* iff all the layers are communication-closed.

Using these notions, we are now able to state a sufficient condition for the equivalence of a program to a decomposition of it into layers.

Theorem. *A distributed program P is equivalent to any of its safe decompositions into layers.*

Proof. By induction on d, the depth of the decomposition.

The basic argument involves a detailed case analysis to establish the commutativity of communications belonging to two different layers with disjoint source and target processes. Due to this commutativity, one reduces a safe decomposition of P of depth d, $P' :: L_1; \ldots ; L_d$ to an equivalent safe decomposition of P of depth $d-1$, given by

$$P'' :: L_1; \ldots ; \tilde{L}_{d-1}, \quad \text{where } \tilde{L}_{d-1} :: [S^1_{d-1}; S^1_d \| \ldots \| S^n_{d-1}; S^n_d],$$

and this claim follows by the induction hypothesis. □

As an example to the commutativity, consider the simple CSP distributed program

$$P :: [P_1 :: P_2?x; P_2?x \| P_2 :: P_1!0; P_1!1 \| P_3 :: P_4?y \| P_4 :: P_3!2]$$

and suppose the decomposition P' is such that

$$L_1 :: [P_2?x; P_2?x \| P_1!0; P_1!1 \| A \| A] \quad \text{and} \quad L_2 :: [A \| A \| P_4?y \| P_3!2].$$

Then, the history

$$h = \langle 2, 1, 0\rangle\langle 4, 3, 2\rangle\langle 2, 1, 1\rangle$$

would belong to $H_{\tilde{L}_1}$, but not to $H_{L_1;L_2}$, since the communication between P_3 and P_4, belonging to the second layer of P', was interleaved between the two communications between P_1 and P_2, both belonging to the first layer. However, $H_{L_1;L_2}$ contains an equivalent history

$$h' = \langle 2, 1, 0\rangle\langle 2, 1, 1\rangle\langle 4, 3, 2\rangle$$

obtained from h by commuting the last two communications, which are independent. Other cases are treated similarly. Note that the equivalence implies also that P is deadlock-free iff all layers of P' are deadlock-free.

Our claim is that using such decompositions one could simplify various analysis methods of distributed programs, such as verification or testing, since less computations have to be considered. We exemplify such a simplification in the context of program verification in Section 2.2.

2.2. Safe decomposition and program verification

In this subsection, we show an application of safe decompositions to program verification. Several verification techniques for distributed programs expressed in CSP were suggested, among which [2] and [12] are state-assertional proof methods, using the usual $\{p\}P\{q\}$ notation for partial correctness assertions.

Assume any proof-system H in which partial correctness assertions can be derived. Let $P :: [P_1\| \dots \|P_n]$ be a distributed program, and let $P' :: L_1; \dots ; L_d$ be a safe decomposition of P into layers. Then, we observe that by the theorem above the following rule is a sound enhancement of H:

$$\text{(SD)}\quad \frac{\{q_0\}L_1\{q_1\}, \dots, \{q_{d-1}\}L_d\{q_d\}}{\{q_0\}P\{q_d\}},$$

where each assertion $\{q_j\}L_{j+1}\{q_{j+1}\}$, $0 \leq j < d$, is provable in H.

In this way the layer boundary points (corresponding to the sequential composition of the layers) serve as natural synchronization points where an assertion must hold, even though in the 'real' computation it need not be the case that all the n controls reside *simultaneously* at a layer boundary. It is as if we are able to 'freeze' a local state at the boundary point and let an assertion hold only after *all* local states are 'frozen'.

We shall exemplify the use of the SD rule by verifying a variant of a distributed program to compute the minimum of n natural numbers $a_1, \dots, a_n$, taken from [12]. The underlying communication graph is the full graph over n vertices. Each element a_i is located at the process $M[i]$. The overall structure of the program is $MIN :: [M[1]\| \dots \|M[n]\|R]$, where R is a receiver process whose job is to accept $m = \min_{1\leq i\leq n}(a_i)$. For brevity we omit all declarations in the following text for the program of $M[i]$. For clarity we subscript all variables with their process index, and introduce mnemonic labels to be able to refer to program sections by their name.

```
$M[i]$ ::
  $init_i$: $my\_min_i := a_i$; $c_i := 1$; $sent_i := false$;
  $find_i$:* [ $\square_{j=1,n;\ j\neq i}$ $c_i < n$; $\sim sent_i$; $M[j]!(my\_min_i, c_i) \rightarrow sent_i := true$
              $\square$
              $\square_{j=1,n;\ j\neq i}$ $c_i < n$; $\sim sent_i$; $M[j]?(their\_min_i, cc_i)$
                                         $\rightarrow my\_min_i := min(my\_min_i,$
                                            $their\_min_i)$; $c_i := c_i + cc_i$
             ];
```

```
fin_i:   [sent_i → skip
          □
          ~sent_i → R!my_min_i
         ].
```

The details of R are also omitted, and we assume it has the corresponding $M?m$ command.

The 'big box' notation $\Box_{j=1,n,j\neq i}$ is an abbreviation for the expanded set of alternatives, where the case $j=i$ is exluded.

The intuitive explanation of the way the program acts is the following: Inductively, each process is responsible for maintaining the minimum of some non-empty subset of $\{a_1,\ldots,a_n\}$ of cardinality c_i, so that these subsets form a partition. At any round in its main loop, a process may send its local minimum away (together with the corresponding count), to any other process and exit the loop, or receive a local minimum and a count from some process and update its own local minimum and local count to represent the minimum of the union of the two subsets. At the end, only one process remains, representing the minimum of the whole set (with count equal to n), sending it to R. The difference from the program in [12] is in the introduction of counts, to avoid termination depending on CSP's distributed termination convention.

In the program text we appended labels to statements, anticipating the intended safe decomposition. We now define the following layers, using the mnemonic names derived from the corresponding labels.

$$INIT :: [init_1 \| \ldots \| init_n \| A],$$
$$FIND :: [find_1 \| \ldots \| find_n \| A],$$
$$FIN :: [fin_1 \| \ldots \| fin_n \| R].$$

Claim. *The decomposition MIN′ :: INIT; FIND; FIN is safe.*

Proof. Trivial (can be verified syntactically). □

In order to use the SD rule mentioned above, we annotate the layer boundaries with assertions as follows:

$$\{true\}$$
$$INIT;$$
$$\{\min_{i=1,n}(my\_min_i) = \min_{i=1,n}(a_i) \wedge \bigvee_{i=1,n} sent_i = false\}$$
$$FIND;$$
$$\{\exists 1 \leq i_0 \leq n. sent_{i_0} = false \wedge \forall 1 \leq i \leq n. i \neq i_0 \supset sent_i = true \wedge my\_min_{i_0} = \min_{i=1,n}(a_i)\}$$
$$FIN$$
$$\{m = \min_{i=1,n}(a_i)\}.$$

By applying the rule SD we can now deduce $\{true\}MIN\{m = min_{i=1,n}(a_i)\}$ which expresses the required correctness property of the program.

That each layer satisfies the corresponding pre-post assertions relationship can be derived using the proof systems of [2] or [12], and is simpler than verifying the whole program within any of the two systems. For example, the proof is relieved from considering situations in which some processes are still initializing, while others already left their loop. In the case of [2], these situations would be reflected in a complicated global invariant.

3. Decomposing loops

In this section we consider more general decompositions into layers. We allow more complicated layers whose boundaries may cross those of loops. Thus, different traversals of the same loop may be placed in different layers.

Let S be an iterative statement of the following form:

```
S :: *[$b_1$; $c_1 \rightarrow T_1$
       □
       •
       •
       •
       □
       $b_k$; $c_k \rightarrow T_k$
      ]
```

which we also abbreviate as $*[G_1 \square \ldots \square G_k]$.

Here $b_1, \ldots, b_k$ are the boolean parts of guards (ranging over variables local to the process in the case of CSP); $c_1, \ldots, c_k$ are the communication parts of guards (we take c_i as '*skip*' if not explicitly included); $T_1, \ldots, T_k$ are any program segments and G_i is an abbreviation for $b_i; c_i \rightarrow T_i$, $1 \leq i \leq k$. A guard $b; c$ is *passable* iff b is true and c matches a complementary i/o command in the process it is addressing. Also, denote by BS the selection constituting the body of the loop S.

There seems to be a rather elaborate theory of ordinary 'while-loops', but not much about non-deterministic loops as considered here. Hence, we start by identifying several properties of such loops which will be used to describe our intended decompositions. Since most known programming languages do not contain the appropriate constructs that would be necessary in order to treat the most general case of loop decompositions, we shall isolate some simpler special cases and deal with them.

Remark. In this section, we assume that the loops *always terminate.* Also, we disregard the CSP distributed termination convention, and assume that loops terminate *only* due to all boolean guards being false.

Definition. Let $A = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, k\}$. An A-slice of S is $SL_A :: G_{i_1} \square \ldots \square G_{i_m}$. The A-slice SL_A is *active* if it contains a passable guard.

In other words, an A-slice of a loop S is a fragment of the loop body, consisting of the collection of guards indexed by elements of A. Since A may be any subset, an A-slice is an arbitrary grouping of alternatives. We shall be interested in groupings which are not that arbitrary, and alternatives belonging to the same slice bear some logical relationship to each other.

Definition. For an A-slice of a loop S, the *A-induced loop* is $*[SL_A]$.

Thus, the A-induced loop repeats executing alternatives of the corresponding A-slice only.

Definition. Let $\mathcal{A} = (A_1, \ldots, A_d)$ be a partition of $\{1, \ldots, k\}$, the set of guard indices of a loop S. We say that $\mathcal{A}$ is a *faithful-slicing* of S iff at every traversal of the loop, at most one A_j-slice is active, $1 \leqslant j \leqslant d$, $A_j \in \mathcal{A}$.

Thus, in a faithful slicing, it is never the case that two guards belonging to different slices are passable. Obviously, this is a semantic property of the loop S, and usually cannot be syntactically determined given the partition $\mathcal{A}$. One well-known example of a faithful slicing occurs when the body of the loop is deterministic, i.e., the guards exclude each other, and $\mathcal{A}$ is the partition to singletons.

Note that the alternative operator '$\square$' is both associative and commutative, and the ordering of the alternatives is immaterial.

Definition. A loop S, faithfully sliced by $\mathcal{A} = (A_1, \ldots, A_d)$ is called *$\mathcal{A}$-flattenable* iff it is equivalent to the sequence of its A_i-induced loops, $1 \leqslant i \leqslant d$, i.e., to $S' :: *[SL_{A_1}]; \ldots ; *[SL_{A_d}]$.

Thus, in an $\mathcal{A}$-flattenable loop, each slice is executed repeatedly as long as it is active, and never attempted anymore thereafter.

We shall exemplify all these notions in the following example program, written in CSP. It describes a process D(istributor), that inputs from a process A a sequence of natural numbers, whose sum is accumulated in a local variable, s. The end of the sequence is sensed by inputing from A the special signal *eos*( ). The sum is distributed to all members of an array of processes $B[i]$, $i = 1, \ldots, N$, in any order. Local flags, $sent[i]$, $i = 1, \ldots, N$ are used to control the sending, avoiding no sending or double sending. The initial state satisfies:

$$s = 0,\ more = true,\ sent[i] = true,\ i = 1, \ldots, N.$$

The loop is given by

```
$S :: *[more; A?a \rightarrow s := s + a$
    $\square$
    $more; A?eos() \rightarrow more := false$; for $i := 1$ to $N$ do $sent[i] := false$
    $\square$
    $\square_{i=1,N} \; \neg sent[i]; B[i]!s \rightarrow sent[i] := true$
    $]$.
```

We assume guards are numbered in order of appearance. A natural faithful slicing of S is according to the partition $\mathscr{A} = (A_1, A_2)$, where $A_1 = (1, 2)$, $A_2 = (3, \ldots, N+2)$. It can be easily verified that S is equivalent to $S' :: *[SL_{A_1}]; *[SL_{A_2}]$. Initially, $\bigwedge_{i=1}^{N} sent[i]$ holds, hence all the guards of SL_{A_2} are non-passable. In order to become passable, the second alternative, receiving $eos()$ from A must be executed, which falsifies the flag *more*, and deactivates the A_1-slice. This could be formally proven using the proof-systems in [2] or [12] for CSP. Hence this program is (A_1, A_2)-flattenable.

Thus, in the local analysis of S, an assertion to the effect that

$$s = \sum_{j=1}^{n} a_j \wedge \neg more \wedge \bigwedge_{i=1,N} \neg sent[i]$$

holds after some iterations in the loop. Obviously, this is *not* a loop invariant.

We would like to capture a similar phenomenon in its full breadth, and find a global assertion to hold after every process executed some number of loop traversals. This, again, motivates the broader concept of a communication-free layer.

Consider a distributed program $P :: [P_1 \| \ldots \| P_n]$, where some of the processes P_j constitute flattenable loops for appropriate faithful slicings. By adding enough dummy statements, we can assume that all loops are flattened with the same number, d, of induced loops.

We can now apply again the definition of a *layer*, this time taking S_j^i, the ith component in the jth layer, as the jth-induced loop in the flattening of the ith loop, if P_i happens to be a loop. In case such a decomposition is communication-closed, Theorem 1 can be proved for the non-simple decompositions similarly to the simple case, given the fact that we deal with loops that terminate and are flattenable.

The intuitive interpretation of the boundary between successive layers is again a virtual synchronization point. The states of all the processes are 'frozen' at such a point, and may be shown to satisfy a given assertion. Since the layers are communication-closed, the 'real' execution, which does not synchronize at layer boundaries, is equivalent to the synchronized one. In particular, if a layer component is an induced loop, the original loop is 'frozen' after a number of traversals, though the 'real' execution might have proceeded to consecutive traversals.

As an example to the application of this way of analysis of a distributed program, we shall consider a simple example, where we show how to abstract from a pipelining effect. The interested reader is referred to [7] for examples of an attempt to verify directly a program containing pipelining effects, using (a generalization of) the cooperating-proof system of [2].

Example. *A remote adder.* Consider an array $P[i]$, $i = 1, \dots, n$ of processes. The process $P[1]$ is a source, emitting the elements of a local array $a[1, \dots, N]$. The processes $P[2], \dots, P[n-1]$ serve as elements of a pipeline, passing elements 'from left to right'. Process $P[n]$ is an adder, summing up all the elements received in a local variable, s. The ending is signalled by the special tagged message *eos*(), emitted by $P[1]$ upon completion of emitting the elements of the array a. Every process receiving the *eos*() signal sends it to its right neighbor and halts. When the adder receives the *eos*() signal it also halts. Upon termination, the assertion $s = \sum_{k=1}^{N} a[k]$ should hold.

The special signal *eos*() is introduced to bypass the distributed termination convention of CSP, which we excluded here. For brevity, we include a '*halt*' primitive statement, whose meaning is terminating the process activity. Following is the text of the full program:

```
            P :: [P[1]||...||P[n]]   where
              P[1] :: $j := 1$; *[$j \le N$; P[2]!a[j] $\to$ $j := j+1$
                                  □
                                  $j > N$; P[2]!eos() $\to$ halt
                                 ],
$(1<i<n)$     P[i] ::        f := true; done := false;
                             *[$\neg$done; f; P[i-1]?x $\to$ f := false
                               □
                               $\neg$done; f; P[i-1]?eos() $\to$ done := true
                               □
                               $\neg$done; $\neg$f; P[i+1]!x $\to$ f := true
                               □
                               done; P[i+1]!eos() $\to$ halt
                              ],
              P[n] :: s := 0;*[P[n-1]?x $\to$ s := s + x
                               □
                               P[n-1]?eos() $\to$ halt
                              ].
```

One natural way to reason intuitively about this program is to consider it in phases, where in the jth phase, the array element $a[j]$ 'travels' all the way from $P[1]$ to $P[n]$ and is added to s, and only then is the next element of the array a starting its 'travel', in another phase. Then, after adding $a[j]$ to s, the assertion $s = \sum_{k=1}^{j} a[k]$ would hold.

However, $s = \sum_{k=1}^{j} a[k]$ is *not* a (global) invariant (in the sense of [2]), since by the time $a[j_0]$ arrives to $P[n]$, more elements might have been emitted into the pipeline, and $j > j_0$ would hold, falsifying the above assertion.

We could capture this intuitive understanding of the remote adder program by considering a safe decomposition of it, behaving as explained, where the phases are the execution of layers in the decompsotion, with virtual synchronization between phrases.

The faithful slicing of P_i, $1<i<n$ would be by the partition $\mathcal{A}=(\{1,2\},\{3,4\})$, and we use *SEND*, *RECEIVE* as abbreviation for the slices. Thus

```
RECEIVE_i :: ¬done; f; P[i−1]?x → f := false
             □
             ¬done; f; P[i−1]?eos() → done := true

SEND_i ::    ¬done; ¬f; P[i+1]!x → f := true
             □
             done; P[i+1]!eos() → halt
```

Obviously, $P[1]$ has only a $SEND_1$ slice, including the increment of j, while $P[n]$ has only $RECEIVE_n$ slice, including the addition to s.

The faithfulness of this slicing can be easily verified. The following diagram shows the safe decomposition which can be easily seen to be equivalent to the original remote adder program. All the initializations are grouped into one layer, *INIT*. Thus, $P'::INIT:P''$, and

```
P'' :: L_1 :: [SEND_1||RECEIVE_2||A ...                   ||A]
                                                             ;
       L_2 :: [A||SEND_2||RECEIVE_3||A ...                ||A]
       •
       •                                                     ;
       •
       L_n :: ||A|| ...               ||A||SEND_{n−1}||RECEIVE_n].
```

In order to give a syntactic expression of this decomposition, we have to parametrize more than CSP allows.

Define

$$L[i]::[\underbrace{A\|\ldots\|A}_{i-1\text{ times}}\|SEND_i\|RECEIVE_{i+1}\|\underbrace{A\|\ldots\|A}_{n-1-i\text{ times}}].$$

Then our program could be expressed as

$$P''::\textbf{for } i:=1 \textbf{ to } n \textbf{ do } L[i].$$

In order to prove the safety of this decomposition, one could use the [2] cooperating proofs, and show that the dynamic matching of i/o guards is as claimed.

Thus, decompositions do not minimize the total effort of analysis, but modularize it by separation of concerns:

(1) Show the safety of a decomposition (thus guaranteeing its equivalence to the original program), and

(2) Reason about the virtual synchronization at layer boundaries, which is usually simpler than direct reasoning about the given program.

We applied this method successfully to some tree algorithms, with waves of communications (see, e.g., [10]), where the distinguished phases of up-tree and down-tree can be easily expressed as layers. The construction of one such simple program is shown in Section 5.

Obviously, one could extend these ideas to more elaborate decompostitions, not depending on flattenability, but most programming languages do not have the construct to express the resulting programs.

While passing, we note that suitable safe decomposition could be used to force program behavior according to the MAX-semantics for concurrency as given in [17].

4. A language construct for enforcing safe decomposability

In this section we suggest a programming language construct that could relieve the programmer from taking care of explicit synchronization to induce safety of an intended decomposition into layers. We start by an example which will motivate the need of that kind of a construct.

Let us return to the example in Section 2, of computing the minimum of a set $A=\{a_1,\ldots,a_n\}$ of elements. Suppose now that we are interested in computing repetitively the minimum of $k>1$ such sets $A_j=\{a_1^j,\ldots,a_n^j\}$, $1\leq j\leq k$.

An immediate solution would be to repeat the program *MIN* of Section 2 k times, each time with the elements of a new set A_j, by nesting *MIN* in **for**-loop:

for $j:=1$ **to** k **do** $MIN :: [M[1] \| \ldots \| M[n] \| R]$.

This certainly constitutes a correct solution, but an inefficient one in terms of processor utilization. The reason for the inefficiency is again the over-synchronization: a computation of $Min(A_{j+1})$ cannot start until that of $Min(A_j)$ is ended! However, one certaintly sees that a more loose synchronization would suffice, enabling a process that has finished its part in the computation of A_j to proceed to its part in the computation in A_{j+1}.

In the other extreme, one could consider embedding R and each $M[i]$ in a loop:

$M'[i]$ :: **for** $j:=1$ **to** k **do** $M[i]$, and then let
$MIN' :: [M'[1] \| \ldots \| M'[n] \| R']$.

The solution has no synchronization at all, but is incorrect since it does not guarantee that when $M'[i_1]$ and $M'[i_2]$ communicate, they refer to elements of the *same* A_j; this would be the case if these two corresponding communication requests were at the same jth iteration of the corresponding local loops.

Thus, the ultimate solution is obtained by inserting some synchronization that guarantees that whenever $M'[i_1]$ and $M'[i_2]$ communicate, they are synchronized to the same A_j, i.e., are both in their jth iteration of their local loop. We shall not bother here with a detailed programming of a specific synchronization method (a recent suggestion for such a method appears in [16]). Rather, we observe that its effect could be interpreted as rendering each activation of *MIN* as a communication closed layer!

By adding to the programming language a means for declaring this intention, the compiler could automatically generate the required synchronization code to enforce the safety of the layering.

Thus, one would like to describe the solution as

```
M_MIN :: [MM[1]|| ... ||MM[n]||MR], where
            MM[i] :: for j := 1 to k do
                         layer(j): begin ... code of M'[i] ... end layer(j)
                     od.
```

Therefore, the programmer is free to write down the unsynchronized version, and a **layer**(j) declaration that tells the compiler to generate the required synchronization code that enforces every two processes to be in the same iteration when communicating.

This solution achieves both the efficiency of letting processes 'run forward' whenever possible, and also the ease of expressibility, shifting the generation of synchronization overhead from a programmer to a compiler, who might do it well enough by using optimization techniques.

As far as program analysis is concerned, obviously the program can be analyzed in terms of its enforced decomposition and all the advantages mentioned above again apply.

Since, in general, the automatic deduction of a safe-layer structure of an arbitrary distributed program is impossible, we believe that a language construct indicating and enforcing such a structure may turn to be an important control structure, enhancing better structured concurrent programming.

5. Using communication-closed layers in constructing distributed programs

While in the previous sections we concentrated on the analytic approach, whereby distributed programs were analyzed in terms of their safe decompositions, we take a more synthetic approach in this section by considering communication-closed layers as a tool for the systematic construction of distributed programs.

We propose the following methodology:

(a) Starting with a specification of a whole distributed program, refine this specification into a sequence of layers specifications.

(b) Implement each layer separately by using any stepwise refinement method available for concurrent and distributed programming.

(c) Compose the layers into a whole program, preserving their communication-closeness.

Obviously, the outcome of applying this methodology is a program that is naturally amendable to safe decompositions. It gives the programmer the usual benefits of separation of concerns as it is a 'divide and conquer' method.

Since formal specifications of distributed programs and formal manipulation of such specifications are not advanced enough at the current state of art, we shall apply step (a) in an informal way. The suggested method may gain much in its effectiveness once such formal tools will be available. In the following example, we are not so much interested in the algorithmic consideration of a solution, e.g., its efficiency, as we are in the steps involved in its construction. Thus, a straightforward algorithm is chosen.

Example. *Constructing a distributed program to determine the size of a tree.*

Problem specification (informal): Suppose we are given a dynamic binary tree of processes, with a root process R, some intermediate node processes I_k, $k \in K$, and leaf processes L_q, $q \in Q$.

Here K and Q are two finite, non-empty index sets. We assume that each process 'knows' the identity of its parent and two descendants.

Furthermore, assume that we are guaranteed that the tree structure is fixed throughout the duration of the algorithm. Design a distributed *TCOUNT* program (initiated by the root), to compute the size (i.e., the number of nodes) of the tree.

We assume the following naming convention. The root process R and every intermediate node process I communicate with their offspring nodes process through channels l and r. Also, each leaf process L and intermediate node process I communicates with their parent node through a channel u.

We next derive the specifications of two layers, to be called *UP* and *DOWN*, each implementing a communication pattern known as a (unidirectional) wave [10].

A natural straightforward solution to a problem is obtained by an upgoing wave of communication, in which each process communicates to its parent the size of the subtree of which it is the root. However, since the root is the algorithm initiator, the leaves have to be notified when to start firing. Thus, the upward wave mentioned above is preceded by a downward wave in which each process notifies its descendants to initiate counting the subtree which they root.

As a consequence, we get the following two layers specifications:

DOWN – send a count signal in a wave from the root to leaves,
UP – accumulate sizes of subtrees from the leaves to the root.

Thus, the program *TCOUNT* will be safely decomposable as

TCOUNT :: *DOWN*; *UP*.

Had this been the 'real' structure, it would be over-synchronized, since no leaf would start its second, upward wave until *all* leaves finished the downward wave, which is not necessary. The final program will loosen this over-synchronization by taking care of behaving in an equivalent way.

We now proceed in further refining each layer.

In order to achieve the specification of *DOWN*, assuming the depth of the tree to be d, a sequence of d layers D_1: . . . ; D_d can be designed. In layer D_j, $1 < j < d$, every node of depth j receives a counting signal from its parent. As for D_1, it consists of R sending the count signal spontaneously to its two descendants; layer D_d consist of all leaves of level d receiving the counting signal.

A similar refinement applies to *UP*. To achieve its specification d layers $U_1; \ldots ; U_d$ can be designed. In layer U_j, $1 \leq j < d$, every node of depth $d+1-j$ sends the sizes of the subtree rooted by it to its parent. The size is 1 for a leaf and $1 + size(l) + size(r)$, where $size(l)$ and $size(r)$ are the sizes of the subtrees rooted by its descendants received in the previous layer. At layer U_d, the root, having received the sizes of the subtrees of its two descendants, terminates by determining the size of the whole tree.

At this stage, we obtained specifications that are directly implementable. Again, we choose CSP (with the extension to dynamic target determination [5]) as the programming language for expressing the constructed program. Up to this stage, the design was basically language-independent.

For a layer D_j, $i < j < d$, we have

$$D_j :: [\ldots \| SEND\_SG_k \| \ldots \| RCV\_SG_{k'} \| \ldots].$$

Here P_k is at depth j and is not a leaf (including the root), and $P_{k'}$ is at depth $j+1$. A leaf of depth j has Λ, the empty program, at that layer, as do all processes at a level different than j and $j+1$. Following is the CSP code for these program sections:

```
$SEND\_SG_k :: l\_send_k := true\,;\; r\_send_k := true\,;$
              $*[l\_send_k\,;\; P_{l(k)}!count(\,) \rightarrow l\_send_k := false$
                $\square$
                $r\_send_k\,;\; P_{r(k)}!count(\,) \rightarrow r\_send_k := false$
              $]$.
```

A nondeterministic loop is used to enable a process not to commit itself in advance to the order in which it communicates with its two descendants. This is a 'typical' CSP programming style. For clarity, we subscript variable names with the index of the process to which they belong.

```
$RCV\_SG_{k'} :: u\_received_{k'} := false\,;$
               $*[\neg u\_received_{k'};\; P_{u(k')}?count(\,) \rightarrow u\_received_{k'} := true]$.
```

The reason for having a loop here will become apparent after proceeding to the composition stage. Similarly, for a layer U_j, $1 < j \leq d$, we have

$$U_j :: [\ldots \| RCV\_CT_{k'} \| \ldots \| SEND\_CT_k \| \ldots]$$

where P_k is at depth $d+1-j$ and $P_{k'}$ is at depth $d-j$. The code sections are given by

```
$RCV\_CT_{k'} :: l\_receive_{k'} := true\,;\; r\_receive_{k'} := true\,;\; s_{k'} = 1\,;$
               $*[l\_receive_{k'};\; P_{l(k')}?sl_{k'} \rightarrow s_{k'} := s_{k'} + sl_{k'};\; l\_receive_{k'} := false$
                 $\square$
```

```
                    $r\_receive_{k'}; P_{r(k')}?sr_{k'} \rightarrow s_{k'} := s_{k'} + sr_{k'}; r\_receive_{k'} := false$
                 ].

    $SEND\_CT_k$ :: $u\_send_k := true$;
                 $*[u\_send_k; P_{u(k)}!s_k \rightarrow u\_send_k := false]$.
```

At layer U_1, for the root process R, the $SEND\_CT$ section is Λ, as are the sections in U_j for processes not at level $d+1-j$ or $d-j$.

Again, the reason for having a loop (executed once) is explained below.

Remarks. (1) We use the same variables of program sections in different layers. Had the layers been sequentially composed, this would not work, since the variables would become local to the concurrent command, i.e., the layer containing them. However, we already anticipate the next step of creating a process from the composition of the corresponding sections in each layer, thus variables become local to a process, and carry their value from one layer to the next.

(2) As the construction of this example is presented, it apparently depends on preassigning depths to the various processes. This, however, is not so, since the depth is reflected in the number of Λ sections 'padding' the real code. By correctly composing layer sections to processes, preserving the communication-closeness, all the Λ and the dependence on depth become implicit.

We now turn to the final stage of the suggested methodology, and compose processes out of layer sections, keeping in mind that we have to preserve communication closeness.

There are two natural ways of composing processes. The more straightforward one will sequentially compose corresponding sections. This approach would yield, after some further simplifications, the following program for an I-process

```
    $I$ :: $P_{u[I]}?count(\,)$; $[P_{l[I]}!count(\,) \| P_{r[I]}!count(\,)]$;
                   $[P_{l[I]}!sl \| P_{r[I]}?sr]$; $P_{u[I]}!sl + sr + 1$.
```

Although being simpler than the other solution we derive, it has the drawback of having *insistent* communication (i.e., having no possibility of an alternative in case the partner is not ready to communicate). We would rather choose a somewhat more interesting solution (though more complicated) having the same functionality, but having *indulgent* communications (i.e., having the ability to do something else if the partner is busy, coming back later to the communication). The reader interested in the difference between insistant and indulgent computation may consult [10]. Besides, the other approach exemplifies folding the whole structure into a loop, appropriately sliced. This is the reason for introducing loops, known to be traversed once only, in some sections above.

Actually, in this example the communication closeness is being taken care of almost automatically. Given that the root, which has to be known, starts the algorithm, though all other processes wait to receive to $count(\,)$ signal, only its two descendants will have communication request that will match the roots, by our naming convention. Thus, the wave will advance as if level by level.

By first folding the D_j section with the D_{j+1} section, we get, for P_k, an I node process.

```
$D_k$ :: $l\_send_k := true$; $r\_send_k := true$; $u\_received_k := false$;
    $*[\neg u\_received_k$; $P_{u(k)}?count() \rightarrow u\_received_k := true$
      □
      $u\_received_k$; $l\_send_k$; $P_{l(k)}!count() \rightarrow send_k := false$
      □
      $u\_received_k$; $r\_send_k$; $P_{r(k)}!count() \rightarrow r\_send_k := false$
    ].
```

It is easy to see that the partition {1}, {2, 3} is flattenable faithful slicing. Initially, the first guard is enabled while the two others are disabled. After one execution of the first alternative, it is now disabled, while the other two become enabled, and finally, after one execution of each of the alternatives {2, 3}, the whole loop terminates.

For P_k in L, i.e., a leaf, we obtain a simplified loop, without the two last alternatives.

One can see that a similar procedure yields, for the U-sections and P_k an I process,

```
$U_k$ :: $l\_receive_k := true$; $r\_receive_k := true$; $u\_send_k := true$; $s_k := 1$;
    $*[l\_receive_k$; $P_{l(k)}?sl_k \rightarrow s_k := s_k + sl_k$; $l\_receive_k := false$
      □
      $r\_receive_k$; $P_{r(k)}?sr_k \rightarrow s_k := s_k + sr_k$; $r\_receive_k := false$
      □
      $\neg l\_receive_k$; $\neg r\_receive_k$; $u\_send_k$; $P_{u(k)}!s_k \rightarrow u\_send_k := false$
    ].
```

Again, it is easy to verify that the natural slicing is faithful and flattenable.

Finally, by merging D_k; U_k into one loop, we obtain, for an intermediate node process the following program:

```
$I$ :: $l\_send := true$; $r\_send := true$; $u\_received := false$;
   $l\_receive := true$; $r\_receive := true$; $u\_send := true$; $s := 1$; $next := false$;
   $*[\neg u\_received$; $P_{u(1)}?count() \rightarrow u\_received := true$
     □
     $u\_received$; $l\_send$; $P_{l(I)}!count() \rightarrow l\_send := false$
     □
     $u\_received$; $r\_send$; $P_{r(I)}!count() \rightarrow r\_send := false$
     □
     $\neg next$; $\neg l\_send$; $\neg r\_send \rightarrow next := true$
     □
     $next$; $l\_receive$; $P_{l(I)}?sl \rightarrow s := s + sl$; $l\_receive := false$
     □
```

```
        $next; r\_receive; P_{r(I)}?sr \rightarrow s := s + sr; r\_receive := false$
        □
        $next; \neg l\_receive; \neg r\_receive; u\_send; P_{u(I)}!s \rightarrow u\_send := false$
        ].
```

Note the new boolean variable *next*, added for the sake of making the (U, D) slicing faithful. Some of the flag manipulation can be simplified, but we do not bother with it here.

By similar considerations, one gets the following programs for the root R and leaf processes L:

```
    $R$ :: $l\_send := true; r\_send := true;$
                    $l\_receive := true; r\_receive := true; s := 1; next := false;$
        *[$l\_send; P_{l(R)}!count() \rightarrow l\_send := false$
         □
         $r\_send; P_{r(R)}!count() \rightarrow r\_send := false$
         □
         $\neg next; \neg l\_send; \neg r\_send \rightarrow next := true$
         □
         $next; l\_receive; P_{l(R)}?sl \rightarrow s := s + sl; l\_receive := false$
         □
         $next; r\_receive; P_{r(R)}?sr \rightarrow s := s + sr; r\_receive := false$
         □
         $next; \neg l\_receive; \neg r\_receive \rightarrow HALT$
        ]
```

Actually, upon encountering *HALT*, the root R should send towards the leaves another communication wave, causing all of them to halt, to have a properly terminating program. We skip the details of this extra wave, which are similar to the *DOWN* wave.

Finally, the program for a leaf process is the following:

```
    $L$ :: $u\_received := false; u\_send := true; next := false;$
        *[$\neg u\_received; P_{u(L)}?count() \rightarrow u\_received := true$
         □
         $\neg next; u\_received \rightarrow next := true$
         □
         $next; u\_send; P_{u(L)}!1 \rightarrow u\_send := false$
        ].
```

Note again that in this program, two nodes in the tree can be busy in the upward wave, while in another part of the tree the downward wave is still ongoing. The design assures us of the correct synchronization.

Had we available the language construct described in the previous section, much of the work done in this construction could be done by a compiler.

References

[1] K.R. Apt, Formal justification of a proof system for communicating sequential processes, *J. ACM* **30**(1)(1983).

[2] K.R. Apt, N. Francez and W.P. de Roever, A proof system for communicating sequential processes, *ACM-TOPLAS* **2**(3) (1980).

[3] K.M. Chandy and J. Misra, Proofs of networks of processes, *IEEE Trans. Software Engrg.* **7**(4) (1981).

[4] Tz. Elrad and N. Francez, Weakest precondition semantics for communicating processes, *Proc. 5th International Symposium on Programming*, Torino, Lecture Notes in Computer Science **137** (Springer, Berlin, 1982); to appear in *Theoret. Comput. Sci.*

[5] N. Francez, Extended naming conventions for communicating processes, *Proc. 9th ACM-POPL Symposium*, Albuquerque (1982).

[6] N. Francez, C.A.R. Hoare, D.J. Lehmann and W.P. de Roever, Semantics of nondeterminism, concurrency and communication, *J. Comput. System Sci.* **19**(3) (1979).

[7] N. Francez and S. Katz, Distributed implementation and verification of abstract data types, IBM Israel Scientific Center Report No. 79 (1980), revised (1982).

[8] N. Francez, D.J. Lehmann and A. Pnueli, A linear history semantics for distributed languages, *Proc. FOCS Conference*, Syracuse (1980).

[9] N. Francez and A. Pnueli, A proof method for cyclic programs, *Acta Informat.* **9** (1978).

[10] N. Francez and M. Rodeh, Distributed termination without freezing, *IEEE Trans. Software Engrg.* SE-8, No. 2, May 1982.

[11] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21**(8) (1978).

[12] G. Levin and D. Gries, A proof technique for communicating sequential process, *Acta Informat.* **15** (1981).

[13] R.J. Lipton, Reduction: A method of proving properties of parallel programs, *Comm. ACM* **18**(12) (1975).

[14] R. Milner, *A Calculus for Communicating Processes*, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980).

[15] S.S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, *Acta Informat.* **6** (1976).

[16] F.B. Schneider, Synchronization in distributed programs, *ACM-TOPLAS* **4**(2) (1982).

[17] A. Salwicki and T. Mulder, On algorithmic properties of concurrent programs, manuscript, Mathematical Institute, Polish Academy of Sciences (1981).