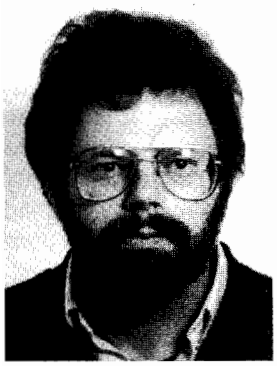
Distributed Computing (1989) 3:73–87

DISTRIBUTED COMPUTING

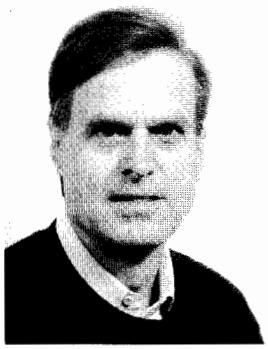
Decentralization of process nets with centralized control

R.-J.R. Back [1] **and R. Kurki-Suonio** [2]

[1] Department of Computer Science, Abo Akademi, Lemminkäisenk. 14, SF-20520 Turku, Finland
[2] Tampere University of Technology, Software Systems Laboratory, Tampere University of Technology, Box 527, SF-33101 Tampere, Finland



Ralph-Johan Back was born in Helsinki, Finland 1949. He received his Ph.D. degree in Computer Science in 1978 from the University of Helsinki. He worked as a research assistant at the University of Helsinki, Computing Centre from 1973 to 1979. He was a visiting scientist at the Mathematical Center in Amsterdam 1979 80 and a senior researcher at the Academy of Finland from 1981 to 1984. He has been professor of Computer Science at Abo Akademi since 1983. His research interests include program methodology, program verification, distributed and parallel programs, programming and specification language design and semantics.



Reino Kurki-Suonio received a Dr. Phil. degree from the University of Helsinki in 1964. At the University of Tampere he headed the first computer science department in Scandinavia since its creation in 1965. Currently he is a professor at the Software Systems Laboratory of Tampere University of Technology. He has held visiting positions at Carnegie-Mellon and Stanford Universities. During the years his research interests have shifted from formal grammars, parsing methods, and programming languages to formal specification and design of distributed systems.

Abstract. The behavior of a net of interconnected, communicating processes is described in terms of the joint actions in which the processes can participate. A distinction is made between centralized and decentralized action systems. In the former, a central agent with complete information about the state of the system controls the execution of the actions; in the latter no such agent is needed. Properties of joint action systems are expressed in temporal logic. Centralized action systems allow for simple description of system behavior. Decentralized (two-process) action systems again can be mechanically compiled into a collection of CSP processes. A method for transforming centralized action systems into decentralized ones is described. The correctness of this method is proved, and its use is illustrated by deriving a process net that distributedly sorts successive lists of integers.

Key words: Decentralized action systems – Process nets – Centralized control

Offprint requests to: R.-J. Back

1 Introduction

Distributed systems differ from centralized ones in that there is no agent which always has complete and up-to-date information about the state of the whole system. This makes it usually much harder to construct a distributed solution to a programming problem than to construct a centralized one. In this paper we shall show that the problem of incomplete information in distributed systems can, to a certain extent, be separated from the main task of constructing a (centralized) solution to the programming problem. We achieve this separation of concerns by first constructing a centralized solution and then transforming this, in a sequence of correctness preserving steps, into a distributed solution. The method by which this decentralization is achieved is the main topic of this paper.

A distributed system is here understood to be a process net, i.e., a collection of processes that

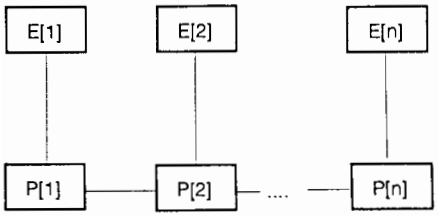


Fig. 1. Example process net

are connected into a network by bidirectional communication channels. The problem we consider is then simply the following: how should one construct the individual processes in this network so that the process net as a whole has the desired behavior?

To make things more concrete, let us consider the process net in Fig. 1. This net consists of the processes $E[1], \ldots, E[n]$, and $P[1], \ldots, P[n]$, with communication channels as shown by the connecting lines. The purpose of the system is to repeatedly sort lists of integers. Each list to be sorted is delivered by the environment processes $E[i]$ to the corresponding sorting processes $P[i]$. The latter will sort the list in parallel, by the neighboring processes $P[i]$ and $P[i+1]$ exchanging their numbers when these are in the wrong order. When process $P[i]$ finally has the ith integer in the sorted list, it returns this to the environment process $E[i]$. As soon as the environment has received all the integers, it generates a new list to be sorted, and so on.

Although this informal specification is simple, programming the individual processes to behave as described is not trivial. The problems are mainly connected with the limited information available to the individual processes. Each of them can only be sure about the values of its own local variables. They may have gathered information about the values in other processes, but this information may be outdated. For instance, how does an individual process know when it has the ith integer of the sorted list? How do the processes know when all integers of the sorted list have been received by the environment? How can two neighbors decide when to exchange their integers, as they do not know each other's integers beforehand? All these problems have to be handled if the process net is to function correctly.

The problems disappear if there is some central agent with immediate information about the local variables of all processes. This agent could use this information to control the computation in the process net. Postulating a central agent of this kind turns out to be a convenient device for specifying the intended behavior of the system. (Actually building such an agent into the system is not, however, such a good idea, as it would constitute a definite bottleneck in the system.) This idea will be elaborated in the following.

The behavior of the process net is described in tems of the *joint actions* in which the processes can participate. Each action involves some processes directly linked to each other by a communication channel. The effect of a joint action is to update the local variables of the processes involved. With each action we associate an *action guard* or *enabling condition*. The action may take place only when this enabling condition holds. This condition may depend on any variables in the system, not only on the local variables of the processes directly involved. The central agent is assumed to evaluate all enabling conditions and to decide whether an action should be executed or not.

To make things even simpler, we will assume that testing whether an action is enabled and its subsequent execution constitute an atomic action in the system. This means that the behavior of the system is the same as if it were executed in a strictly sequential manner: at each stage one of the enabled actions is nondeterministically chosen and executed to completion, before the next action is chosen.

In the example above we need three kinds of actions:

- Action $IN[i]$ in which process $E[i]$ delivers an integer to process $P[i]$. This action should be executed when the previous list has been sorted and all its elements delivered to the environment.
- Action $EX[i]$ in which $P[i]$ and $P[i+1]$ exchange their integers. This action should be executed when the integers in question are in the wrong order.
- Action $OUT[i]$ in which process $P[i]$ delivers an integer to $E[i]$. This action should be executed when $P[i]$ has the ith integer of the sorted list.

In all these actions, the processes involved cannot decide by themselves, by looking at their own local variables only, whether to participate in an action or not. This is true also of action $EX[i]$, since neither $P[i]$ nor $P[i+1]$ has direct access to both integers. In addition to these actions, the processes need to make some private initializations by which the systems is put into an appropriate initial state.

An important advantage of this approach is that it fits nicely in the temporal logic framework for reasoning about concurrent systems, as developed by Manna and Pnueli (1983). We thus have a ready-made formalism in which to express properties of the system, as well as for proving that the system does indeed have these properties. In

fact, our method of describing concurrent systems by actions is essentially their shared variable model. It should be pointed out, however, that while they usually assume that each action is carried out by a single process, we always have co-operation of two or more processes. This affects the fairness assumptions that are reasonable to make.

Our model has some similarity with their modeling of the communication events in CSP (Hoare 1978). A joint action is not, however, only intended to capture the synchronization and message passing between two processes, but should rather be understood as an event of greater extension in time: the two processes are locked into a transaction during which quite a lot of computation can take place. In this sense, joint actions are more like a symmetric version of the rendezvous concept in Ada (US Department of Defense 1980).

The main problem in the approach is that an action system cannot be implemented as such in a distributed fashion. What we would like to have is essentially a compiler that mechanically transforms a joint action system (with two-process actions) into a collection of CSP processes, one process for each node in the process net, such that the behavior is the same as that of the original action system. This compiler should, of course, produce a truly distributed system; introducing the central agent in one form or another in the CSP program would create a bottleneck, as already remarked.

We do not know how to make such a compiler for arbitrary action systems, but we do know how to make one for a restricted class of action systems that we will refer to as *decentralized* action systems. In these systems action guards are restricted in a way that allows each process to determine by itself, whether to participate in an action or not. An action can be executed when both processes are willing to participate, so the central agent is not needed in this case. Joint action systems where action guards do not satisfy these requirements are referred to as *centralized* action systems.

Hence, what we need is a method by which a centralized action system can be transformed into a decentralized one in a way that preserves the temporal properties of the original system. The description of such a decentralization method is the main topic of this paper. The basic idea is to use additional *control variables* in the processes to keep information about the global system state. We also add new actions, called *control actions*, by which this control information is kept up-to-date.

The method for constructing process nets that we propose is thus the following:

1. The intended behavior of the process net is first described as a centralized action system with two-process actions. The correctness of this system is expressed and proved in temporal logic.
2. The centralized action system is transformed into a decentralized one by adding control variables and control actions and modifying the original actions to make use of this additional information and to keep it up-to-date.
3. The resulting decentralized action system is compiled into an equivalent collection of CSP processes, each process controlling one node in the process net.

The last step can be automated, but decentralization requires some invention. The right kind of control information must be found, and the right kind of control actions must be designed. Also, the fact that the resulting system is equivalent to the original one must be proved (in temporal logic).

The advantages of this approach, besides the separation of concerns it induces, is the uniform approach it provides for the construction of distributed programs. There are essentially only two tools used: action systems and temporal logic. Only in the last stage, when the final CSP program is generated, is there any involvement with concurrently operating, communicating processes, and this stage can be automated.

The method of using a control computation to collect information about the global state of the system is not new in itself, see e.g., Chandy and Misra (1982) and Francez (1980). The formalization of this method as a transition from a centralized to a decentralized action system is, however, new, and provides a systematization of this approach, while at the same time a stringent criterion for the correctness of the decentralization is given within temporal logic. The use of decentralized action systems to describe the communication behavior of a collection of processes, as well as the method for translating such systems into CSP, is believed to be new. A method for doing the opposite translation, CSP to a kind of Petri nets, is described in Gergely and Ury (1982), while de Cindio et al. (1982) describes a variant of Petri nets that is quite close to our decentralized action systems, although their nets are not directly translatable into CSP. The connection with Manna and Pnueli (1983) was already mentioned above.

The rest of the paper is organized as follows. Joint action systems are described more precisely in the next section. Section 3 describes decentralized action systems as well as the way they can be transformed into CSP programs. The decentrali-

zation method is given and proved correct in Sect. 4. In Sect. 5 the method is illustrated by distributing a centralized action system for the sorting network outlined above. In the concluding remarks, we will give a more detailed account on related work.

2 Joint action systems

Let I be an index set, interpreted as the set of process names. A joint action system over I is a triple $S=(y, \mathscr{A}, \mathscr{F})$, where $\mathscr{A}$ is a set of *actions*, $\mathscr{F}$ is a family of subsets of $\mathscr{A}$, called the *fairness family*, and y is a set of variables indexed by I, i.e., $y=\{y_i \mid i \in I\}$, which we refer to as the *system state*. Variable y_i is interpreted as the *local state* of process i, consisting of its local variables and the location counter. In an *initialized* system the state y has a given initial value y_0 called the *initial state*. In non-initialized systems the initial state is arbitrary.

Each action $A \in \mathscr{A}$ is of the form

$$A:\ g_A(x_A) \longrightarrow y_A := f_A(y_A).$$

Here x_A and y_A are two non-empty subsets of y. The action *guard* g_A is a condition that the *enabling variables* x_A must satisfy for A to be enabled. The effect of the action is to assign new values $f_A(y_A)$ to the *update variables* y_A. (In the following we sometimes use the whole state y as an argument for g_A and f_A, even if only a subset of it is involved.) The topology of the process net determines which processes can be engaged in a joint action. More precisely, if the update variables y_A of action A are y_i and y_j, then processes i and j must be linked by a communication channel in the net.

We have above assumed that each action involves only two processes. This restriction is, however, only needed when implementing a decentralized action system in CSP. The definitions and results will therefore be given for the general case where any non-empty collection of processes may be involved in a joint action.

A *computation* in an action system S, starting from an initial state y_0, consists of applying in succession one enabled action after the other, as long as some action is enabled. More precisely, a computation is a (finite or infinite) sequence t of the form

$$t:\ y^0 \xrightarrow{A_1} y^1 \xrightarrow{A_2} y^2 \xrightarrow{A_3} \ldots$$

where each A_i is an action in $\mathscr{A}$, and each y_i is a possible state of the system, and which satisfies the following three conditions.

Firstly, the sequence must be *admissible*, i.e.,

$$g_{A_i}(y^{i-1}) = \text{true},$$
$$y^i = f_{A_i}(y^{i-1})$$

for each A_i in the sequence.

Secondly, if a computation t is finite, it must be *properly ending* in the final state y^n:

$$\forall A \in \mathscr{A}:\ \neg g_A(y^n).$$

Thirdly, a computation t must be *fair* with respect to each set $\mathscr{A}_i \in \mathscr{F}$. By this we mean, in accordance with [8], that one of the following conditions holds:

(i) t is finite, or
(ii) t is infinite and, from a certain point on, no action in $\mathscr{A}_i$ is enabled, or
(iii) an infinite number of actions in $\mathscr{A}_i$ occur in t.

Notice that each computation is necessarily fair with respect of the set of all acions $\mathscr{A}$. By default, we assume that $\mathscr{A}$ always belongs to $\mathscr{F}$.

Since each action is assumed to be deterministic, a sequence of actions $A_1, \ldots, A_i$ and the initial state determine the state y^i uniquely. The meaning of a initialized action system S can therefore be taken as the set of all finite or infinite sequences that correspond to computations. Such action sequences will be referred as a execution *traces*. The set of all possible trace t in S will be denoted by $T(S)$.

We shall illustrate the concept of joint action systems by the example outlined in the previous section. It turns out to be convenient to use labeled locations in the processes. Semantically this adds nothing to the model, as a location counter can be understood as another component in the local variable of the process, as is done in [MaPn82]. Both $E[i]$ and $P[i]$ will have two locations:

- at give, $E[i]$ is ready to give an integer to $P[i]$,
- at take, $E[i]$ is ready to receive an integer from $P[i]$,
- at in, $P[i]$ is ready to receive an integer from $E[i]$,
- at comp, $P[i]$ is ready to exchange integers with its neighbors, or to give its integer to $E[i]$.

Read as a statement, **at** l states that the location counter is set to l. In a condition, the same expression says that the location counter has the value l.

A natural form for expressing such sequencing constraints is offered by Petri nets as shown in Fig. 2. This shows, in fact, two subnets of the complete net. Joint actions are modeled here as events,

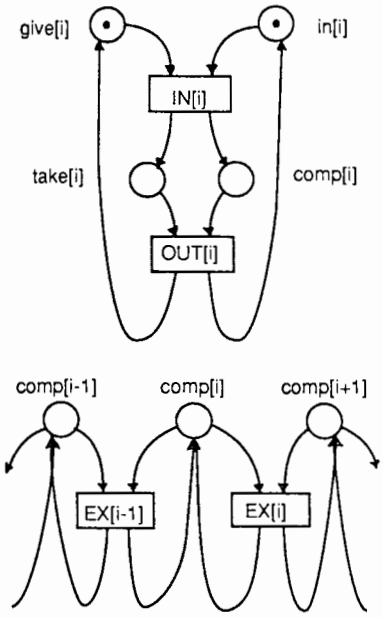


Fig. 2. Sequencing constraints of the sorting net

and the processes are represented by tokens (considered to contain the local variables). Each place node is associated with a location in some process, and the process token may only occupy places associated with this process.

The local variables of the processes and their initialization (including process labels) can be given as follows:

process $E[i: 1..n]$; **var** x, z: integer;
label give, take;
begin
$x := i^{th}$ integer of the first list; **at** give
end E;

process $P[i: 1..n]$;
var y: integer; s, t: boolean; **label** in,comp;
begin
$s, t :=$ true, true; **at** in
end P;

The three kinds of joint actions in which the processes may be engaged, are as follows:

action $IN[i: 1..n]$ **by** $E[i]$ **at** give, $P[i]$ **at** in;
when $\forall j: s[j] = s[1]$
begin
$y := x$; $t := \neg t$; **at** take, comp
end IN;

action $EX[i: 1..n-1]$
by $P[i]$ **at** comp, $P[i+1]$ **at** comp;
when $y[i] > y[i+1]$
begin
$y[i], y[i+1] := y[i+1], y[i]$;
at comp, comp
end EX;

action $OUT[i: 1..n]$ **by** $E[i]$ **at** take, $P[i]$ **at** comp;
when $\forall j: t[j] = t[1] \wedge$
$\forall j, k: (j \le i \le k) \Rightarrow (y[j] \le y[i] \le y[k])$
begin
$z := y$; $x := i^{th}$ integer of the next list;
$s := \neg s$; **at** give, in
end OUT.

Action $OUT[i]$ e.g., requires the participation of processes $P[i]$ and $E[i]$, where $P[i]$ must be at label "take" and $E[i]$ at label "comp". The action is enabled when the "when"-condition is satisfied. The effect of the action is to perform the assignments given in the statement block. After these assignments, process $E[i]$ will be at label "give" and process $P[i]$ will be at label "in". Variables s and t are here used to achieve the necessary synchronization to prevent two successive lists from becoming mixed with each other.

To property that this system should have is the following: Whenever the environment processes generate a new list to be sorted in variables x, they will eventually receive the sorted list in variables z and generate the next list in x. The formulation of this property in temporal logic, as well as proving that the above system has it, is left to Sect. 5.

3 Decentralized action systems

Consider an action A in an action system $S = (y, \mathscr{A}, \mathscr{F})$,

$$A: g_A(x_A) \longrightarrow y_A := f_A(y_A).$$

The guard of A (and A itself) is said to be local, if x_A is a subset of y_A. In other words, the enabling of a local action depends only on the local variables of the processes involved in the action.

The guard of A (and A itself) is said to be *separable*, if it is a boolean expression with unary predicates only. That is, g_A is built with boolean connectives out of atomic formulas of the form $p(y_i)$, where y_i is the local state of process i, $y_i \in x_A$. This forbids atomic formulas relating local states of several processes with each other. (The local state y_i may, of course, have an internal structure, e.g., that of a record, and $p(y_i)$ may relate components of y_i with each other.)

The guard of A (and A itself) is said to be *distributed*, if it is both local and separable. A joint action system is said to be *decentralized*, if all its

actions are distributed. If this is not the case, the system is *centralized.*

Let A be a distributed action involving two processes P and Q with local states y_P and y_Q, respectively. Because of separability, the guard g_A has the form

$$g_A = p_1(y_P) \wedge q_1(y_Q) \vee \ldots \vee p_n(y_P) \wedge q_n(y_Q),$$

when written in disjunctive form. This allows us to implement action A in CSP as follows.

Assume first that there is only one disjunct in g_A, i.e., $n=1$. Action A then has the form

$$A:\ p(y_P) \wedge q(y_Q) \longrightarrow y_P, y_Q := f_P(y_P, y_Q), f_Q(y_P, y_Q).$$

The two conjuncts can now be evaluated separately in the two processes, and the following commands are equivalent to action A:

$$P:\ [p(y_P); Q!\, a(y_P) \longrightarrow Q?\, y_Q; y_P := f_P(y_P, y_Q)],$$

$$Q:\ [q(y_Q); P?\, a(y_P) \longrightarrow P!\, y_Q; y_Q := f_Q(y_P, y_Q)].$$

We have here added the local variables y_Q and y_P to P and Q respectively, to stand for the corresponding variables in the other process. According to the rules of CSP (Hoare 1978), the communication will take place only if $p(y_P) \wedge q(y_a)$ holds. The constructor a is dedicated to this communication event, and the pattern matching capability of CSP then prevents interference with other similar communication events. This constructor is only needed in the first communication associated with an action.

In case g_A consists of two or more disjuncts, i.e., $n>1$, the situation can be handled by associating n different constructors with the action. In this case A can be implemented by the following commands in P:

$$[p_1(y_P); Q!\, a_1(y_P) \longrightarrow Q?\, y_Q; y_P := f_P(y_P, y_Q)$$
$$\vdots$$
$$\Box\, p_n(y_P); Q!\, a_n(y_P) \longrightarrow Q?\, y_Q; y_P := f_P(y_P, y_Q)],$$

together with the following commands in Q:

$$[q_1(y_Q); P?\, a_1(y_P) \longrightarrow P!\, y_Q; y_Q := f_Q(y_P, y_Q)$$
$$\vdots$$
$$\Box\, q_n(y_Q); P?\, a_n(y_P) \longrightarrow P!\, y_Q; y_Q := f_Q(y_P, y_Q)].$$

Assume now that all actions in a centralized system have been split into communication commands for the processes involved. We can then collect all communication commands $b_i \longrightarrow S_i$, $i=1, \ldots, m$, in P into one big iteration statement, which together with the initialization S_0 gives the process for node P:

$$P::\ [S_0; *[b_1 \longrightarrow S_1 \Box \ldots \Box b_m \longrightarrow S_m]].$$

The above discussion shows how a decentralized action system is implemented in CSP. This implementation is completely mechanical. Hence, to construct a distributed system in the form of a process net, it is sufficient to construct a decentralized action system for the processes involved.

There is, however, still another matter to deal with in connection with this implementation. That is the question of fairness (and justice). The problem is that what one may consider as natural fairness assumptions for a centrally controlled action system, may not be actually realizable in the CSP implementation of an otherwise equivalent decentralized action system. This question is treated in detail in other papers (Back and Kurki-Suonio 1984a, 1985, 1987). It sufficies to say here that if no fairness assumptions are required of the decentralized system, then the above CSP implementation is always correct.

4 Decentralization method

The decentralization of a centralized action system proceeds stepwise, taking one action (or some collection of closely related actions) at a time, and replacing it with a distributed action (or actions) with the same effect on the original system variables. As described briefly in the introduction, this requires adding control variables to the processes for keeping relevant information about the global system state, and control actions to keep this information up-to-date. The computation done by these new actions is loosely referred to as the *control computation.*

The method to be described does not give an exact prescription of how the control actions should be designed in order to distribute an action. Control computation may be triggered by some of the main actions in the system, or we may have a continually running control computation that periodically updates certain control variables. The distinction is similar to that between ordinary garbage collection and on-the-fly garbage collection. We may also have a combination of both kinds of computations simultaneously going on in the system.

What we will do is to define a general form for the control computation used to distribute actions, together with conditions that guarantee that the resulting new action system is a correct implementation of the original system. The method we describe is actually more general than what we need for the derivation of CSP implementations, as it can be applied to replace any actions by local

actions with the same effect. Thus, the replacing actions need not be separable, and the method works for actions with more than two processes.

Let $S=(y, \mathscr{A}, \mathscr{F})$ be an initialized action system. Each action B in S has the form

$$B:\ g_B(x_B) \longrightarrow y_B := f_B(y_B),$$

where x_B and y_B are subsets of y. Let A be used to denote an action that we want to distribute in a distribution step.

The distribution of actions will yield another initialized action system $S'=((y, z), \mathscr{A}', \mathscr{F}')$, where z is a collection of *control variables*. The components of z are local to the processes, which should be modified to give them suitable initialization. The actions in the modified system S' are as follows:

(i) For each action $A\in\mathscr{A}$ to be distributed there is a corresponding action $A'\in\mathscr{A}'$:

$$A':\ l_A(y_A, z_A) \longrightarrow y_A, z_A := f_A(y_A), h_A(y_A, z_A).$$

Thus, an action A' differs from A in that its enabling condition is local and may depend on local control variables z_A, and that its execution may also update these control variables. The effect on the original state variables y_A is, however, unchanged, and the new values of y_A do not depend on the values of z_A.

(ii) For the other actions $B\in\mathscr{A}$ there are corresponding actions $B'\in\mathscr{A}'$:

$$B':\ g_B(x_B) \longrightarrow y_B, z_B := f_B(y_B), h_B(y_B, z_B).$$

An action B' thus differs from B only in that it may, as a side effect, update some control variables that are local to the processes involved. The enabling condition and the effect on the original state variables remain unchanged.

(iii) The new control actions $C\in\mathscr{A}'$ have the form

$$C:\ g_C(y_C, z_C) \longrightarrow z_C := h_C(y_C, z_C).$$

A control action does not affect the old state variables, although its enabling condition and the new values of the control variables may depend on (local) old state variables.

The actions introduced by (i) and (ii) will be called *main actions* in order to distinguish them from the *control actions* of (iii). If the fairness family of S is $\mathscr{F}=\{\mathscr{A}_i\}$, then the fairness family $\mathscr{F}'$ of S' is defined to contain the sets $\mathscr{A}_i'$ obtained from $\mathscr{A}_i$ by replacing each A and B in $\mathscr{A}_i$ by the corresponding A' or B'. According to our conventions, the set of all actions $\mathscr{A}'$ also belongs to $\mathscr{F}'$. Notice that, as $\mathscr{A}\in\mathscr{F}$, then the set of all main actions also belongs to $\mathscr{F}'$, i.e., S' mut be fair with respect to main actions.

In addition, we make the following assumptions for actions A:

$$\Box(g_A \Rightarrow \Diamond\, l_A) \quad \text{in } S', \tag{1}$$

$$\Box(l_A \Rightarrow g_A) \quad \text{in } S', \tag{2}$$

$$g_A \wedge I \Rightarrow \Diamond\, l_A \quad \text{in } CS', \tag{3}$$

where I is some invariant in S', i.e., $\Box I$ in S', and CS' is the non-initialized action system formed by the control actions in S'.

Assumption (1) states that whenever the global guard g_A for an action A becomes true in the modified system, the local guard l_A for A' will eventually become true. Assumption (2) states that the local condition l_A for A' never becomes true unless the global condition g_A for A holds. Finally, assumption (3) states that the control computation alone, without any help from main actions B', is capable of enabling A', if the global condition g_A for A holds. The invariant I is introduced to make it possible to weed out those initial states of CS' that cannot possibly occur in any computation in S', and hence need not be taken into account in (3).

Notice that the fairness family $\mathscr{F}'$ guarantees that control computations cannot monopolize execution in S', so that no main actions ever get done. One way to impose this property on S' is to construct the control actions so that no infinite control computation without intervening main actions is possible. This is the case if

$$I \Rightarrow \Diamond \bigwedge_C \neg\, g_C \tag{4}$$

holds in CS' for some I such that $\Box I$ in S'. This has the advantage that fairness requirements for the implementation are not accumulated in the stepwise decentralization of the original system. (Notice that the control actions of one step become main actions in the subsequent steps.)

Given a trace t' in S', let $a(t')$ denote the sequence which is obtained from it by removing all control actions, and by replacing each A' by A and B' by B. It is obvious from the construction of S' that, if repetitions due to intervening control actions are ignored, the sequence of y values generated by t' in S' is the same as that generated by $t=a(t')$ in S. Since only the y component is of interest in the first place, we say that t' *simulates* t. As t can only refer to variables in y, t' satisfies all the temporal properties that t satisfies, provided that the next state temporal operator is disallowed.

The correctness of the decentralization method is expressed by the following theorem:

Theorem 1. *For the sets of traces $T(S)$ and $T(S')$, we have $T(S)=a(T(S'))$.*

The theorem is proved by the following two lemmas:

Lemma 1. $T(S)\subseteq a(T(S'))$.

Proof. Let $t\in T(S)$ be an arbitrary execution trace in S. The construction of a corresponding trace $t'\in T(S')$ with $t=a(t')$ is done by induction.

Assume that we have constructed a simulation u' of u, $a(u')=u$, where u is a prefix to t. Initially, u is empty, and choosing u' also empty gives a simulation of u. If the next action in t is of type B then the corresponding B' must be enabled after u', so $u'B'$ is a simulation of uB. If the next action after u in t is of type A, then A' need not be enabled after u', but on account of (3), there is a control computation v' enabling A', so $u'v'A'$ is a simulation of uA.

This process allows us to construct an admissible sequence t' that has exactly the same effect on y as t. We still have to show that t' either is a trace is S' or can be completed to one by trailing control actions.

If t is finite, we have to check that t' can either be made to end properly or be extended with an infinite but fair control computation. Since t is a trace, no A or B is enabled in its final state. No B' and, because of (2), no A' can then be enabled in the end of t'. There may, however, be control actions which are enabled. Since these do not modify the state component y, they cannot enable any B', and their enabling of some A' would lead into contradiction with (2). Therefore, only control actions are possible after t', so any completion of t' to a trace simulates t.

If t is infinite, we have to check the fairness of t'. If a main action in some $\mathscr{A}_i'\in\mathscr{F}'$ is infinitely often enabled in t', then the corresponding action in $\mathscr{A}_i\in\mathscr{F}$ is infinitely often enabled in t. (Actions B and B' have the same guards; for actions of type A this follows from (2).) Therefore this action occurs infinitely often in t, and hence, the corresponding main action occurs infinitely often in t'. This completes he proof of Lemma 1. □

Lemma 2. $a(T(S'))\subseteq T(S)$.

Proof. Let $t'\in T(S')$ be an arbitrary trace in S'. We have to show that $t=a(t')$ is a trace in $T(S)$.

First we check that t is an admissible sequence of actions in S. This is true, since the guards of B' and B are the same, and (2) implies that A is always enabled when A' is.

For a finite t we have to show that it ends properly. In t' there can only occur control actions after the last main action. If an action A would be enabled in the end of t, then g_A would hold during the trailing control computation in S'. Because of (1), the guard l_A of A' would then eventually be turned on. If the trailing sequence was finite, A' would then be enabled in the end of t', as g_A would still hold, and t' would not itself be properly ending; if the trailing sequence was infinite, A' would be enabled infinitely often (but still not be executed), and t' would not itself be fair with respect to main actions. If an action B would be enabled in the end of t, then B' would be enabled in the end of t' and during the trailing control computation, which also leads to contradiction.

Finally, the fairness of an infinite t is guaranteed by (1) which shows that, if an action of type A is infinitely often enabled in t, then the corresponding main action is infinitely often enabled in t'. For actions of type B the same thing is obvious, since the guards are the same in both systems. This completes the proof of Lemma 2. □

5 Examples

In this section we shall apply the decentralization method to the sorting example given in Sect. 2. More detailed analysis of decentralized action systems for this problem is given in [1]. We start with analyzing the operation of the centralized solution.

As for notations, we shall use $\bar{x}$, $\bar{y}$, and $\bar{z}$ to denote the vectors $x[i]$, $y[i]$, and $z[i]$, $i=1,\ldots,n$, respectively. By $\bar{u}$ and $\bar{v}$ we denote the following vectors: $u[i]=v[i]=y[i]$ when process $P[i]$ is at label comp, but $u[i]=x[i]$ and $v[i]=z[i]$ when $P[i]$ is at label in. The notation $perm(\bar{a})$ will be used for the set of permutations of a vector $\bar{a}$, and $n(\bar{a})$ is the cardinality of the set $\{(i,j)|i<j\wedge a[i]>a[j]\}$, i.e., the number of exchanges of consecutive numbers required to sort $\bar{a}$.

The specification of the system will be given in terms of the processes $E[i]$, which represent its external behavior. In order to state the desired criteria, let GIVE stand for the predicate indicating that all $E[i]$ are ready to give a number in the next list to $P[i]$:

GIVE: $\forall j$: $E[j]$ **at** give,

and, for the i*th* list of numbers $\bar{a}_i$, let INIT_i, MID_i, and END_i describe the situation at the three moments when all $E[i]$ are ready to give this list, when some numbers of the list have been given, and when the sorted version of the list has just

been received, respectively:

INIT_i: $\bar{x}=\bar{a}_i \wedge \text{GIVE}$,
MID_i: $\bar{x}=\bar{a}_i \wedge \neg\text{GIVE}$,
END_i: $\bar{z}\in perm(\bar{a}_i) \wedge n(\bar{z})=0 \wedge \text{INIT}_{i+1}$

The system specification is now

INIT_1 and
$\Box[(\text{INIT}_i \Rightarrow \text{INIT}_i \text{ Until } \text{MID}_i) \wedge$
$(\text{MID}_i \Rightarrow \neg\text{GIVE Until END}_i)]$.

The first condition simply states that the system is initially ready to start with the first list $\bar{a}_1$. The second condition states that, when the manipulation of the *ith* list $\bar{a}_i$ has started, then its sorting will be completed when the system is ready for the next list, and that this will eventually happen.

5.1 Correctness of action system

The first condition is obviously true because of the initializations. Let us therefore concentrate on the other condition. Notice first how $s[i]$ and $t[i]$ indicate the current labels of $P[i]$ and $E[i]$:

INV: $(s[i]=t[i]) \equiv (P[i]$ **at** in $\wedge E[i]$ **at** give$) \wedge$
$(s[i]\neq t[i]) \equiv (P[i]$ **at** comp $\wedge E[i]$ **at** take$)$.

Since this is enforced by the initializations and is preserved by all actions, it is an invariant of the system, i.e., $\Box$INV holds.

To reflect the way variables s and t are used in the system we strengthen INIT_i and END_i in the proof to

INIT'_i: $\text{INIT}_i \wedge \forall j$: $s[j]=s[1]=t[j]$,
END'_i: $\text{END}_i \wedge \text{INIT}'_{i+1}$.

INIT'_1 is obviously established by the initializations. For each $\bar{a}_i$, INIT'_i starts the *input stage*, characterized by

INEX_i: $\bar{u}\in perm(\bar{a}_i) \wedge$
$\forall j$: $s[j]=s[1] \wedge \exists j$: $s[j]=t[j]$.

Only actions IN and EX can be executed in this stage:

$\Box[\text{INEX}_i \Rightarrow \forall j$: $\neg\text{Enabled}(OUT[j])]$.

The first action IN turns on the condition MID_i for the rest of the input stage. GIVE is in then false. With the execution of the last action IN, the system enters a state where all $P[i]$ are at label comp, and all $t[i]$ have been complemented. This starts the *output stage*, characterized by

EXOUT_i: $\bar{v}\in perm(\bar{a}_i) \wedge$
$\forall j$: $t[j]=t[1] \wedge \exists j$: $s[j]\neq t[j]$,

which obviously implies $\neg$GIVE. Only actions EX and OUT are here possible:

$\Box[\text{EXOUT}_i \Rightarrow \forall j$: $\neg\text{Enabled}(IN[j])]$.

Only a finite number of these actions can be executed (each EX decrements $n(\bar{v})$ by one). With the execution of the last action OUT, condition GIVE becomes true, and END'_i then holds, which means that END_i also holds. This completes the proof.

Notice that no fairness assumptions were needed, as each EX$[i]$ decreases $n(\bar{u})$ in the input stage and $n(\bar{v})$ in the output stage. The decentralization of the system will be such that all control computations satisfy (4), and no fairness assumptions will therefore be introduced into the decentralized system either.

5.2 Decentralization of IN

It follows from the above proof that $s[i]$ and $s[j]$ always have the same value when both $P[i]$ and $P[j]$ are at label in. Hence, the enabling conditions of all $IN[i]$ are true when all $P[i]$ are at label in. Action $IN[i]$ remains enabled until executed. Therefore, decentralized control can be achieved with control actions that communicate information about $P[i]$ being at label in.

In order to implement this, we provide each $P[i]$ with two control variables, *leftok*$[i]$ and *rightok*$[i]$, which turn true when all $P[j]$ for $j\leq i$, resp. $j\geq i$, have entered label in. The required invariants for these variables are the following:

$P[1]$ **at** in $\equiv$ *leftok*$[1]$, $P[n]$ **at** in $\equiv$ *rightok*$[n]$,
leftok$[i]$ $\Rightarrow (P[i]$ **at** in $\wedge \forall j\leq i$: $s[j]=s[i])$,
rightok$[i] \Rightarrow (P[i]$ **at** in $\wedge \forall j\geq i$: $s[j]=s[i])$.

These invariants are enforced by initializing *leftok*$[i]$ and *rightok*$[i]$ to true, and letting $IN[i]$ turn them false. The only OUT actions that can safely change them are $OUT[1]$ and $OUT[n]$, which can set *leftok*$[1]$, resp. *rightok*$[n]$ to true.

However, in order to prevent $IN[i]$ from turning *leftok*$[i]$ and *rightok*$[i]$ false prematurely, i.e., before also *leftok*$[i+1]$ and *rightok*$[i-1]$ have been turned on, another pair of control variables, *lready*$[i]$ and *rready*$[i]$, is introduced with the invariants

lready$[n]$ = *leftok*$[n]$, *rready*$[1]$ = *rightok*$[1]$,

$i<n \Rightarrow$ *lready*$[i]$ = *leftok*$[i+1]$,

$i>1 \Rightarrow$ *rready*$[i]$ = *rightok*$[i-1]$.

These are initialized to *true*, and turned false together with *leftok* and *rightok* by IN.

The following control actions can then spread "ready signals" to all other $P[i]$:

action $LEFTOK[i: 1..n-1]$
by $P[i]$ **at** in, $P[i+1]$ **at** in;
when $leftok[i] \wedge \neg leftok[i+1]$
begin
$lready[i], leftok[i+1] := \text{true}, \text{true}$;
$lready[i+1] := (i = n-1)$
at in, in
end LEFTOK,

action $RIGHTOK[i: 1..n-1]$
by $P[i]$ **at** in, $P[i+1]$ **at** in;
when $\neg rightok[i] \wedge rightok[i+1]$
begin
$rready[i+1], rightok[i] := \text{true}, \text{true}$;
$rready[i] := (i = 1)$;
at in, in
end RIGHTOK;

The enabling condition of $IN[i]$ can now be changed into the local condition

$$lready[i] \wedge rready[i]$$

which implies the original condition, i.e., condition (2) for the correctness of the decentralization holds. On the other hand, consider the situation when $s[1] = \ldots = s[n]$ and $P[i]$ is at label in. When the system is started, the local condition holds by the initializations. Otherwise the situation arises when the initial state is re-entered from the output stage. Since all $P[i]$ are then at label in, at least $leftok[1]$ and $rightok[n]$ are true. Therefore, if $lready[i]$ or $rready[i]$ is not true, there is a control action LEFTOK resp. RIGHTOK which is enabled. As only a finite number of executions of those control actions is possible before the IN actions, $lready[i]$ and $rready[i]$ must eventually turn true, i.e. correctness condition (1) holds. Thus the decentralization is correct (conditions (3) and (4) are straightforward).

Notice that the variables $s[i]$ have turned into "ghost" variables that can be deleted without affecting the correctness of the system.

5.3 Decentralization of EX

The enabling condition for $EX[i]$ is local to the processes $P[i]$ and $P[i+1]$ involved. Its implementation is not possible, however, without nontrivial communication, since the relation $y[i] > y[i+1]$ cannot be evaluated by any of the two processes alone. We therefore introduce control actions by which the condition can be modified into separable form.

The condition could be evaluated in either $P[i]$ or $P[i+1]$, if they would possess up-to-date information about each other's numbers y. We therefore introduce control variables $ly[i]$ and $ry[i]$ for recording the latest avialable information about $y[i-1]$ and $y[i+1]$, respectively. Boolean control variables $cleft[i]$ and $cright[i]$ are used to trigger control actions by which this information is updated when required.

When an action $EX[i]$ is executed, the values of $y[i]$ and $y[i+1]$ are updated. Consequently, it is possible that the enabling conditions for $EX[i-1]$ or $EX[i+1]$, i.e., $y[i-1] > y[i]$ or $y[i+1] > y[i+2]$, become true. Let us consider the case when the former condition becomes true; the latter case is analogous. If the new $y[i]$ is smaller than $ly[i]$, $EX[i-1]$ can be enabled by a local condition of $P[i]$. However, as the value of $ly[i]$ may be outdated, we need the control variable $cleft[i]$ to indicate the need for updating $ly[i]$ by a control action $CHECK[i-1]$.

More precisely, $\bar{v} = \bar{y}$ will hold during the output stage, and, in addition, the control variables for EX are required to satisfy the invariants:

$$y[i] \geq ly[i+1], \quad y[i+1] \leq ry[i],$$
$$y[i] = ly[i+1] \vee cright[i],$$
$$y[i+1] = ry[i] \vee cleft[i+1],$$

for $i = 1, \ldots, n-1$. This situation can be achieved by actions IN setting all $cleft[i]$ and $cright[i]$ to true, and all $ly[i]$ and $ry[i]$ to the smallest and largest possible integers, respectively. With the invariants it can be easily verified that

$$(y[i] > ry[i]) \vee (y[i+1] < ly[i+1]) \tag{5}$$

implies $y[i] > y[i+1]$, and that $y[i] > y[i+1]$ implies that either (5) holds or else

$$cright[i] \wedge cleft[i+1]. \tag{6}$$

This means that we can use (5) to guard $EX[i]$, and (6) to guard the control action $CHECK[i]$:

action $EX[i: 1..n-1]$
by $P[i]$ **at** comp, $P[i+1]$ **at** comp;
when $y[i] > ry[i] \vee y[i+1] < ly[i+1]$
begin
$y[i], y[i+1] := y[i+1], y[i]$;
$ry[i], ly[i+1] := y[i+1], y[i]$;
$cleft[i], cright[i+1] := \text{true}, \text{true}$;
$cright[i], cleft[i+1] := \text{false}, \text{false}$;
at comp, comp
end EX;

action $CHECK[i]$
by $P[i]$ **at** comp, $P[i+1]$ **at** comp;
when $cright[i] \wedge cleft[i+1]$
begin
$ry[i], ly[i+1] := y[i+1], y[i]$;
$cright[i], cleft[i+1] :=$ false, false;
at comp, comp
end CHECK;

5.4 Decentralization of OUT

In order to recognize the conditions $(j \leq i) \Rightarrow (y[j] \leq y[i])$ and $(j \geq i) \Rightarrow (y[j] \geq y[i])$ locally, we introduce control variables $lmax[i]$ and $rmin[i]$ to record the maximum $y[j]$ to the "left" of i, and the minimum $y[j]$ to the "right" of i, respectively. In addition, we need control variables $islmax[i]$ and $isrmin[i]$ to indicate that these values have been recorded. Finally, the decentralized control must prevent $OUT[i]$ from being executed before all those control actions have been performed that require $P[i]$ to stay at label comp. This leads to introducing further variables $lmaxgiven[i]$ and $rmingiven[i]$ with the same values as $islmax[i+1]$ and $isrmin[i-1]$, respectively.

More precisely, the new control variables are required to satisfy the following invariants:

$P[1]$ **at** comp $\Rightarrow$
$(islmax[1] \wedge (lmax[1] \leq y[1]) \wedge rmingiven[1])$,

$P[n]$ **at** comp $\Rightarrow$
$(isrmin[n] \wedge (rmin[n] \geq y[n]) \wedge lmaxgiven[n])$,

$\forall i > 1$: $P[i]$ **at** comp $\wedge\ islmax[i] \Rightarrow$
$(t[i] = t[i-1] \wedge islmax[i-1] \wedge lmax[i] = \max_{j<i} y[j] \wedge rmingiven[i] = isrmin[i-1])$,

$\forall i < n$: $P[i]$ **at** comp $\wedge\ isrmin[i] \Rightarrow$
$(t[i] = t[i+1] \wedge isrmin[i+1] \wedge rmin[i] = \min_{j>i} y[j] \wedge lmaxgiven[i] = islmax[i+1])$.

Actions IN can easily enforce these invariants by proper assignments to the control variables, and none of the previous actions will ever turn them false. The required control computations can then be performed by the following actions that operate without any special triggering:

action $GIVELMAX[i: 1..n-1]$
by $P[i]$ **at** comp, $P[i+1]$ **at** comp;
when $islmax[i] \wedge \neg islmax[i+1]$
begin
$lmax[i+1] := lmax[i] \max y[i]$;
$lmaxgiven[i], islmax[i+1] :=$ true, true;
at comp, comp
end GIVELMAX;

action $GIVERMIN[i: 1..n-1]$
by $P[i]$ **at** comp, $P[i+1]$ **at** comp;
when $\neg isrmin[i] \wedge isrmin[i+1]$
begin
$rmin[i] := rmin[i+1] \min y[i+1]$;
$isrmin[i], rmingiven[i+1] :=$ true, true;
at comp, comp
end GIVERMIN;

Actions $EX[i]$ also have to be modified in order to preserve the new invariants:

action $EX[i: 1..n-1]$...
when ...
begin
...
$rmin[i] := rmin[i+1] \min y[i+1]$;
$lmax[i+1] := lmax[i] \max y[i]$;
at comp, comp
end EX;

When the guard for $OUT[i]$ becomes true, the control actions will eventually turn true all the boolean control variables involved, so we can replace this guard by the stronger distributed guard

$(lmax[i] \leq y[i] \leq rmin[i]) \wedge islmax[i] \wedge isrmin[i] \wedge lmaxgiven[i] \wedge rmingiven[i]$.

5.5 Combined actions

In the solution derived above, processes $P[i]$ and $P[i+1]$ have several common actions at label comp. It seems, however, undesirable that $CHECK[i]$, for instance, only updates $ry[i]$ and $ly[i+1]$, but does not exchange $y[i]$ and $y[i+1]$ if they are in the wrong order. Actions $EX[i]$, $CHECK[i]$, $GIVELMAX[i]$, and $GIVERMIN[i]$ can, however, be easily combined, if each of them is first modified so that the guard is rechecked in the action body, and nothing is done if it does not hold. The combined action can then be given a guard that is a disjunction of the individual guards. With this technique we arrive at the following solution:

process $E[i: 1..n]$; **var** x, z: integer;
label give, take;
begin
$x := i^{th}$ integer for the first list; **at** give
end E;

process $P[i: 1..n]$; **var** $y, ly, ry, lmax, rmin$: integer;
$cleft, cright, islmax, isrmin, leftok,$
$lready, rready, rightok, lmaxgiven,$
$rmingiven$: boolean;
label in, comp;

```
    begin
        leftok, rightok, lready, rready := true,
          true, true, true;
        at in
    end P;

action IN[i: 1..n] by E[i] at give, P[i] at in;
    when lready ∧ rready
    begin
        y := x; ly, ry := small, large;
        cleft, cright := true, true;
        lmax, rmin := small, large;
        islmax, isrmin := (i = 1), (i = n);
        lmaxgiven, rmingiven := (i = n), (i = 1);
        lready, rready := false, false;
        leftok, rightok := false, false;
        at take, comp
    end IN;

action EXX[i: 1..n−1]
    by P[i] at comp, P[i+1] at comp;
    when y[i] > ry[i] ∨ y[i+1] < ly[i+1] ∨
        cright[i] ∧ cleft[i+1] ∨
        islmax[i] ∧ ¬islmax[i+1] ∨
        ¬isrmin[i] ∧ isrmin[i+1]
    begin
        if y[i] > y[i+1] then
        begin
            y[i], y[i+1] := y[i+1], y[i];
            cleft[i], cright[i+1] := true, true;
        end;
        ry[i], ly[i+1] := y[i+1], y[i];
        cright[i], cleft[i+1] := false, false;
        lmax[i+1] := lmax[i] max y[i];
        islmax[i+1], lmaxgiven[i] :=
          islmax[i], islmax[i];
        rmin[1] := rmin[i+1] min y[i+1];
        isrmin[i], rmingiven[i+1] :=
          isrmin[i+1], isrmin[i+1];
        at comp, comp
    end EXX;

action OUT[i: 1..n] by E[i] at take, P[i] at comp;
    when y ≥ lmax ∧ y ≤ rmin ∧ islmax ∧
        isrmin ∧ lmaxgiven ∧ rmingiven
    begin
        z := y; x := i^th integer of the next list;
        leftok, rightok := (i = 1), (i = n);
        at give, in
    end OUT;

action LEFTOK[i: 1..n−1]
    by P[i] at in, P[i+1] at in;
    when leftok[i] ∧ ¬leftok[i+1]
    begin
        lready[i], leftok[i+1] := true, true;
        lready[i+1] := (i = n−1);
        at in, in
    end LEFTOK;

action RIGHTOK[i: 1..n−1]
    by P[i] at in, P[i+1] at in;
    when ¬rightok[i] ∧ rightok[i+1]
    begin
        rready[i+1], rightok[i] := true, true;
        rready[i] := (i = 1);
        at in, in
    end RIGHTOK.
```

Variables s and t have been deleted, as they are no longer needed. Some other minor simplifications have also been made in deriving this action system.

5.6 *Implementation in CSP*

Finally we shall show how the resulting action system looks when implemented using the communication primitives of CSP. As already mentioned, this can be done mechanically, as all guards are local and separable. The constructors needed for proper matching of communication commands are a to j. (Same constructors are used for similar communication events between different pairs of processes.) Variables yy, m, is are used in P for storing the values received in the communication.

```
process E[i: 1..n]; var x, z: integer;
                    label give, take;
    begin
        x := i^th integer of the first list;
    give: P[i]! a(x);
    take: P[i]? b(z);
        x := i^th integer of the next list;
          at give
    end E;

process P[i: 1..n];
    var y, ly, ry, lmax, rmin, yy, m: integer;
        cleft, cright, islmax, isrmin, leftok,
        lready, rready, rightok, lmaxgiven,
        rmingiven, is: boolean;
    label in, comp;

procedure leftpart;
  begin
        P[i+1]? (yy, m, is);
        if y > yy ⟶ y, yy := yy, y; cleft := true
        or if y ≤ yy ⟶ skip
```

```
        fi;
        ry:=yy; cright:=false;
        rmin:=m min yy;
        isrmin:=is; lmaxgiven:=islmax
    end leftpart;

procedure rightpart;
    begin
        P[i-1]!(y, rmin, isrmin);
        if yy > y ⟶ yy, y:=y, yy; cright:=true;
        or if yy ≤ y ⟶ skip
        fi;
        ly:=yy; cleft:=false;
        lmax:=m max yy;
        islmax:=is; rmingiven:=isrmin
    end rightpart;

begin
    leftok, rightok, lready, rready:=true,
        true, true, true;

in:
    if lready ∧ rready; E[i]? a(y) ⟶
            ly, ry:=small, large;
            cleft, cright:=true, true;
            lmax, rmin:=small, large;
            leftok, rightok, lready, rready:=
                false, false, false, false;
            islmax, isrmin:=(i=1), (i=n);
            islmaxgiven, rmingiven:=(i=n), (i=1);
            at comp
    or if i<n ∧ leftok; P[i+1]! h( )
        ⟶ lready:=true; at in
    or if i>1 ∧ ¬leftok; P[i-1]? h( )
        ⟶ leftok, lready:=true, (i=n); at in
    or if i<n ∧ ¬rightok; P[i+1]! j( )
        ⟶ rightok, rready:=true, (i=1); at in
    or if i>1 ∧ rightok; P[i-1]? j( )
        ⟶ rready:=true; at in
    fi;
comp:
    if i<n ∧ y>ry; P[i+1]! c(y, lmax, islmax)
        ⟶ leftpart; at comp
    or if i>1; P[i-1]? c(yy, m, is)
        ⟶ rightpart; at comp
    or if i<n; P[i+1]! d(y, lmax, islmax)
        ⟶ leftpart; at comp
    or if i>1 ∧ y<ly; P[i-1]? d(yy, m, is)
        ⟶ rightpart; at comp
    or if i<n ∧ cright; P[i+1]! e(y, lmax, islmax)
        ⟶ leftpart; at comp
    or if i>1 ∧ cleft; P[i-1]? e(yy, m, is)
        ⟶ rightpart; at comp
    or if i<n ∧ islmax; P[i+1]! f(y, lmax, islmax)
        ⟶ leftpart; at comp
    or if i>1 ∧ ¬islmax; P[i-1]? f(yy, m, is)
        ⟶ rightpart; at comp
    or if i<n ∧ ¬isrmin; P[i+1]! g(y, lmax, islmax)
        ⟶ leftpart; at comp
    or if i>1 ∧ isrmin; P[i-1]? g(yy, m, is)
        ⟶ rightpart; at comp
    or if y ≥ lmax ∧ y ≤ rmin ∧ islmax
        ∧ isrmin ∧ lmaxgiven ∧ rmingiven
        ⟶ E[i]! b(y);
            leftok, rightok:=(i=1), (i=n); at in
    fi
end P.
```

6 Concluding remarks

Let us first briefly summarize the main new contributions of this paper.

1. The action system approach as a way to describe distributed systems and as a conceptually simple basis for stepwise refinement of distributed programs is presented. The formalism is motivated by the simplicity of the proof theory. The temporal logic approach of Manna and Pnueli (1983), Pnueli (1986) can be used as such to prove properties of action systems.
2. A method by which control in a distributed program can be refined in a stepwise manner, by adding control variables and control actions and adapting the existing actions to utilize the information gathered by these, is formalized and the conditions under which this method preserves temporal correctness are given. The method supports stepwise refinement of a distributed system, starting from a rather high level description, where the processes are assumed to have information about the global state of the system, and then gradually changing this to a system where processes only utilize the information that they have available in their own local variables.
3. A method by which a decentralized action system can be atomatically translated into a CSP program (with output guards) is described.

This paper was first presented at the 2nd ACM Conference of Principles in Distributed Computing in Montreal 1983 (Back and Kurki-Suonio 1983). Since then, a number of other researchers have also taken up the issues considered here (in some cases unaware of the results described above). We will

below briefly describe the main developments along these lines.

Chandy and Misra (Chandy 1985; Chandy and Misra 1986, 1988) have proposed a very similar method to describing distributed systems as our actions systems, which they call UNITY. They basically view a distributed system as a collection of (conditional) assignment statements. This is equivalent to our mathematical model for action systems. There are some differences in the way in which they understand the execution of an action system, which means that they will have a somewhat weaker fairness notion than the one we use. They have developed the assignment statement formalism so that it is also suitable to describe synchronized parallel algorithms, of the kind used in VLSI-design (a similar approach is also studied by Martin and Tucker (1987). They have also developed a variant of the temporal logic proof theory that is specially engineered to reasoning about the behaviour of their systems, and which seems well suited to reason about action systems also, with some minor modifications. They have applied this approach to a large number of classical programming problems in distributed systems, thus showing the power and versatility of the action based approach. A somewhat similar approach, based on the notion of events, is put forward by Shankar and Lam (1987) and applied to the description of distributed systems with real-time constraints. Other approaches similar to the action system approach are the Raddle approach (see Evengelist et al. 1987 for an overview) and work by Ramesh and Mehndiratta 1987. In the latter approaches, the emphasis is more on the actions as a means for coordinating an arbitrary number of processes in a joint communication. This view of action systems is also taken in Back et al. (1985).

The method for stepwise refinement of control that we have analyzed here has previously been used in different case studies of constructing distributed algorithms in connection with problems such as termination detection (Dijkstra 1980; Francez 1980), algorithms for detecting deadlock (Chandy et al. 1983) and for taking global snapshots (Chandy and Lamport 1985). Chandy and Misra (1986) also use it explicitly as a method for stepwise refinement of distributed programs. They have a somewhat different approach to stepwise refinement, as they concentrate on refining the specification of the distributed system rather than the program description itself. A similar approach to stepwise refinement is also described in Back and Kurki-Suonio (1984a). The technique used in McCurley and Schneider (1986) is also along these lines. The guards used at one step are not implementable because they make use of global state information. Control computations are therefore added to gather the necessary information into the local variables of the processes and the original guards are replaced by new guards that only use this local information.

The approach described in this paper for stepwise refinement has recently gained popularity as a modularization method for distributed programs, referred to as *superposition*. The program is seen as being constructed in layers, where each higher layer consists of control computations that do not interfere with the more basic computations of the lower layers. This modularization technique has been described, with somewhat differing approaches, by Katz (1987), Bouge and Francez (1988) and Chandy and Misra (1988). The last two approaches do not permit control computations to assign to variables in the base algorithm. Katz (1988) takes a different view of superimposition, by also allowing the variables in the original algorithm to be modified by the control computation, although in a controlled way.

The implementation of action systems has also received attention. The main shortcoming of the method proposed here is that it assumes CSP with output guards. The presence of output guards makes this language difficult to implement in a distributed fashion (see e.g., Buckley and Silberschatz 1983). The main problem is how the process should reach agreement about which actions to be executed when two or more conflicting actions are enabled. In Back and Kurki-Suonio (1984b), Back et al. (1985) and Back and Kurki-Suonio (1987), we show how to solve this problem for actions with an arbitrary number of processes on broadcasting networks using broadcasting protocols such as CSMA/CD. This shows that there are efficient implementations of action systems. The problem of implementing action systems on point-to-point networks has been considered by other authors (Eklund 1985; Ramesh 1987; Bagrodia 1987). It is also discussed by Chandy and Misra (1988), who refer to it as the committee coordination problem.

Another interesting aspect of implementing action systems is how to guarantee the fairness assumptions needed to prove the liveness properties of the systems. The problem of guaranteeing fairness by distributed implementations was first considered in Back and Kurki-Suonio (1984b) and is further studied in Back and Kurki-Suonio (1985, 1987). Essentially the same problem, in the context of CSP implementations, is studied in Grumberg

et al. (1984) and is also discussed in Francez (1986). Another recent paper on this topics is Apt et al. (1987).

7 References

Apt K, Francez N, Katz S (1987) Appraising fairness in languages for distributed programming. Proc 14th ACM POPL Conf, Munich 1987, pp 189–198

Back RJR, Hartikainen E, Kurki-Suonio R (1985) Multi-process handshaking on broadcasting networks. Rep Comput Sci 42, Abo Akademi

Back RJR, Kurki-Suonio R (1983) Decentralization of process nets with centralized control. 2nd Annu ACM Symp on PoDC, Montreal, Canada, 1983, pp 131–142

Back RJR, Kurki-Suonio R (1984) A case study in constructing distributed algorithms: distributed exchange sort. Proc Winter School on Theoretical Computer Science, Lammi 1984, Finnish Society for Information Processing, pp 1–33

Back RJR, Kurki-Suonio R (1984) Co-operation in distributed systems using symmetric multi process handshaking. Rep Comput Sci 34, Abo Akademi

Back RJR, Kurki-Suonio R (1985) Serializability in distributed systems with handshaking. Res Rep CMU-CS-85-109, Carnegie-Mellon University, Pittsburgh

Back RJR, Kurki-Suonio R (1987) Distributed co-operation with action systems. Rep Comput Sci Mathematics 56, Abo Akademi

Bagrodia R (1987) A distributed algorithm to implement n-party rendezvous. University of Texas, Department of Computer Science technical report, June 1987

Bouge L, Francez N (1988) A compositional approach to superimposition. 15th ACM Conf on Principles of Programming Languages. San Diego, Calif, pp 240–249

Buckley GN, Silberschatz A (1983) An effective implementation for the generalizated input-output construct of CSP. ACM TOPLAS vol 5, 2 (April 1983), pp 223–235

Chandy M (1985) Concurrent programming for the masses. Proc 4th Annu ACM Symp PODC

Chandy KM, Lamport L (1985) Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3(1):63–75

Chandy M, Misra J (1982) A distributed algorithm for detecting resource deadlocks in distributed systems. ACM SIGACT-SIGOPS Symp on Principles of Distributed Computing 1982, pp 157–164

Chandy M, Misra J (1986) An example of stepwise refinement of distributed programs: quiescence detection. ACM TOPLAS, vol 8, no 3 (July 1866), pp 326–343

Chandy M, Misra J (1988) A foundation of parallel program design. Addison-Wesley (forthcoming)

Chandy KM, Misra J, Haas L (1983) Distributed deadlock detection. ACM Trans Comput Syst 1(2):144–156

de Cindio FG, de Michelis D, Pomello L, Simone C (1982) Superposed automata nets. Second European Workshop on Application and Theory of Petri Nets. In: Girault C, Reisig W (eds), Application and Theory of Petri Nets. Inf Fachber 52, Springer, Berlin Heidelberg New York Tokyo

Dijkstra EW, Scholten CS (1980) Termination detection for diffusing computations. Inf Process Lett, 11(1):1–4

US Department of Defense (1980) Reference manual for the Ada programming language. Proposed Standard Document, US Department of Defense, July 1980

Eklund P (1985) Synchronizing multiple processes in common handshakes. Rep Comput Sci 39, Abo Akademi

Evengelist M, Shen VY, Forman I, Graf M (1987) Using raddle to design distributed systems. MCC Tech Rep STP-285-87, September 1987

Francez N (1980) Distributed termination. ACM Trans Program Lang Syst 2(1):42–55

Francez N (1986) Fairness. Springer, Berlin Heidelberg New York Tokyo

Gergely L, Ury L (1982) Representation and verification of communicating sequential processes. Comp Ling Comput Lang 15:157–174

Grumberg O, Francez N, Katz S (1984) Fair termination of communicating processes. 3rd ACM SIGACT-SIGOPS Symp on Principles of Distributed Computing, Vancouver 1984

Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677

Katz S (1987) A superimposition control construct for distributed systems. MCC Tech Rep STP 268–87

Manna Z, Pnueli A (1983) How to cook a temporal proof system for your pet language. 10th ACM Conf on Principles of Programming Languages, Austin, Texas, pp 141–154

Martin AR, Tucker JV (1987) The concurrent assignment representation of synchronous systems. PARLE, Parallel Architectures and Languages Europe, Proc vol II. (Lect Notes Comput Sci 256) Springer, Berlin Heidelberg New York Tokyo, pp 369–386

McCurley R, Schneider FB (1986) Derivation of a distributed algorithm for finding paths in directed networks. Sci Comput Program 6:1–9

Pnueli A (1986) Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. Current trends in concurrency (Lect Notes Comput Sci 224) Springer, Berlin Heidelberg New York Tokyo, pp 510–584

Ramesh S (1987) A new and efficient implementation of multiprocess synchronization. PARLE, Parallel Architectures and Languages Europe, Proc vol II. (Lect Notes Comput Sci 256) Springer, Berlin Heidelberg New York Tokyo, pp 387–401

Ramesh S, Mehndiratta SL (1987) A methodology for developing distributed algorithms. IEEE Trans Software Engineering SE-13(8):967–976

Shankar A, Lam S (1987) Time-dependent distributed systems: proving safety, liveness and real-time properties. Distrib Comput 2:61–79