Distributed Cooperation with Action Systems

R. J. R. BACK
Abo Akademi
and
R. KURKI-SUONIO
Tampere University of Technology

Action systems provide a method to program distributed systems that emphasizes the overall behavior of the system. System behavior is described in terms of the possible interactions (actions) that the processes can engage in, rather than in terms of the sequential code that the processes execute. The actions provide a symmetric communication mechanism that permits an arbitrary number of processes to be synchronized by a common handshake. This is a generalization of the usual approach, employed in languages like CSP and Ada, in which communication is asymmetric and restricted to involve only two processes. Two different execution models are given for action systems: a sequential one and a concurrent one. The sequential model is easier to use for reasoning, and is essentially equivalent to the guarded iteration statement by Dijkstra. It is well suited for reasoning about system properties in temporal logic, but requires a stronger fairness notion than it is reasonable to assume a distributed implementation will support. The concurrent execution model reflects the true concurrency that is present in a distributed execution, and corresponds to the way in which the system is actually implemented. An efficient distributed implementation of action systems on a local area network is described. The fairness assumptions of the concurrent model can be guaranteed in this implementation. The relationship between the two execution models is studied in detail in the paper. For systems that will be called fairly serializable, the two models are shown to be equivalent. Proof methods are given for verifying this property of action systems. It is shown that for fairly serializable systems, properties that hold for any concurrent execution of the system can be established by temporal proofs that are conducted entirely within the simpler sequential execution model.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Broadcasting networks, distributed systems, fairness, guarded commands, handshaking mechanisms, models of concurrency, multiprocess communication, programming languages, program verification, scheduling, temporal logic, true concurrency

1. INTRODUCTION

The behavior of a distributed system is usually described in terms of a collection of communicating processes. Each process carries out a sequential piece of program and interacts with the other processes by sending and receiving

Authors' addresses: R. J. R. Back, Abo Akademi, Department of Computer Science, Lemminkäisenkatu 14, SF-20520 Abo, Finland; R. Kurki-Suonio, Tampere University of Technology, Computer Systems Laboratory, P.O. Box 527, SF-33101 Tampere, Finland.

messages. Message passing can be synchronized, as in CSP [25], Occam [27], or Ada [1], or it can be asynchronous, as is usually the case in computer networks. This approach can be implemented by standard techniques, and has proved useful in designing distributed systems.

A problem with the process-based approach is that it can be difficult to get a picture of the overall behavior of the system. Processes may interact in unexpected ways, and the number of potential interactions to be considered is usually quite large. This is reflected in proof rules for process-based programs, which tend to be cumbersome and intricate. In this paper we consider another approach to describing distributed systems, which is dual to the process approach: The behavior of the system is described in terms of the possible interactions that may occur between the processes during execution. We refer to these interactions as *actions*, and call a description of the system in terms of interactions an *action system*. By focusing on the interactions that may occur, it is easier to design the overall behavior of the system. On the other hand, it becomes harder to implement such a system in a distributed fashion; so there is a trade-off involved.

The action system approach was first introduced in [5], where it was applied to the stepwise refinement of distributed algorithms. A CSP implementation was given for the special case of two-process actions. The semantic and proof theoretic issues involved in action systems, with special emphasis on fairness issues, were studied in a subsequent paper [7]. The work reported here is an extension of that paper. A case study of applying the mechanism in the systematic derivation of a correct distributed algorithm is given in [6]. Efficient implementations of action systems are described in [7] and [4]. A more detailed study of fairness in action systems is given in [8].

1.1 Informal Presentation of Action Systems

An action system is similar to a production system such as OPS5 [19], i.e., it is a collection of productions that are executed repeatedly, as long as possible. Another similar construct is the guarded iteration statement [15]

do $guard_1 \rightarrow statement_1 \Box \cdots \Box guard_n \rightarrow statement_n$ **od**,

which has the same basic execution mechanism: guarded statements are executed repeatedly, as long as some guard is true. The main difference between action systems and these other language mechanisms is in the way they are executed: guarded commands and production systems are designed for sequential execution, whereas an action system is designed for concurrent execution by a collection of independent communicating processes.

An action system consists of a collection of *processes* and a collection of *actions*. A process is of the form

process p : **var** y_p; $statement_p$.

The variables y_p stand for the *local variables* of process p, while $statement_p$ assigns initial values to these variables. The local variables together constitute the *global state* y of the action system.

An action is of the form

action a **by** $processes_a$: $guard_a \rightarrow statement_a$.

This states that action a may be executed jointly by the processes in the set $processes_a$, provided that $guard_a$ is satisfied and the processes are not participating in other actions (the action is then said to be *enabled*). Executing the action changes the program state in the way described by $statement_a$. We refer to the statement as the *body* of the action. The guard and the body of an action may only refer to local variables of the processes participating in the action, i.e., to the set of variables $y_a = \cup\{y_p \mid p \in processes_a\}$.

In order to allow a distributed implementation of action systems, we will require that the guards in actions are *separable*. This means that the guard is a Boolean condition where each atomic predicate refers to local variables of only one process. The truth of each atomic predicate in the guard can then be determined by some process participating in the action by inspecting only its own local variables. As an example, the guard

$$(x \geqslant 0 \wedge y \leqslant z) \vee w \geqslant 0$$

contains the atomic predicates $x \geqslant 0$, $y \leqslant z$ and $w \geqslant 0$. It is separable if the local variables y and z belong to the same process, otherwise it is not.

The execution of an action system proceeds by repeatedly executing individual actions, as long as there are actions that are enabled. No process may participate in more than one action at a time. As long as this restriction is obeyed, any number of actions may be executed in parallel. An action that is executed by two or more processes together is referred to as a *joint action*, because the processes have to cooperate to achieve the effect of the action. An action that is executed by one process only is referred to as a *private action*, because it can be carried out by that process alone, without cooperation with other processes.

Example 1. Let us consider the Dining Philosophers problem as an example of an action system. A diagram of this system is given in Figure 1. The action system consists of n philosopher processes and n fork processes, $n \geq 2$, declared as follows:

```
process Philosopher[i:1 .. n];
  var hungry:boolean;
  hungry := false;
process Fork[i:1 .. n];
```

A set of processes may be declared by appropriate indexing, as shown by the example. Each philosopher process has a local variable *hungry*, initialized to *false*, to indicate whether he is hungry or not. The fork processes do not have any local variables at all, as they are used purely for synchronization. The global state thus consists of the variables $Philosopher[i].hungry$, $i = 1, \ldots, n$.

There are two kinds of actions in the system, thinking actions and eating actions. The thinking actions are

```
action Thinking[i:1 .. n] by Philosopher[i]:
¬Philosopher[i].hungry →
  Think;
  Philosopher[i].hungry := true;
```

Thinking[i] is a private action by philosopher $Philosopher[i]$. It is enabled if the philosopher is not hungry. Executing the action results in some unspecified

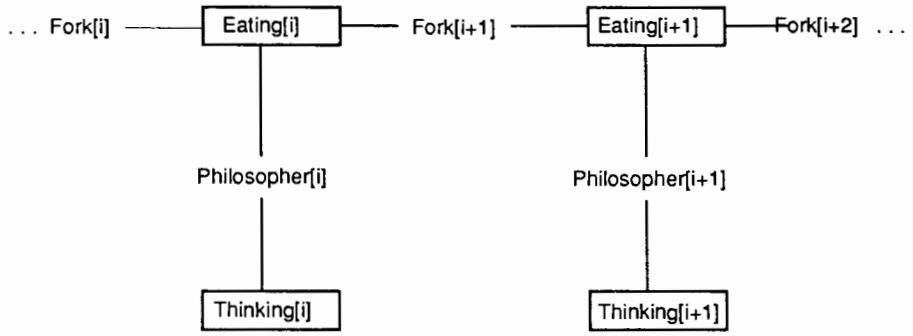

Fig. 1. Dining philosophers.

thinking being done, after which the philosopher becomes hungry again. Each philosopher will thus repeatedly become hungry, as the thinking actions always terminate.

The eating actions are

action *Eating*[*i*:1 . . *n*] **by** *Philosopher*[*i*], *Fork*[*i*], *Fork*[*i* + 1]:
Philosopher[*i*].*hungry* →
  Eat;
  Philosopher[*i*].*hungry* := *false*;

Eating[*i*] is a joint action by *Philosopher*[*i*] and his left and right forks, *Fork*[*i*] and *Fork*[*i* + 1]. The action is enabled when the philosopher is hungry. The action body consists of an unspecified eating part followed by an assignment that makes the philosopher nonhungry. We assume here and in the sequel that all index arithmetic is modulo *n*, so *i* + 1 above stands for *i* mod *n* + 1.

Mutual exclusion and freedom from deadlock are here guaranteed by the action mechanism: a fork can only be engaged in one action at a time, so an eating action can only take place when the philosopher is hungry and the two forks are simultaneously available. Individual starvation is, however, possible: a philosopher will starve if his two neighbors keep eating alternatingly.

1.2 Communication in Action Systems

In the above example the processes are synchronized by the eating actions. However, no transfer of information actually takes place here. To illustrate this, as well as more intricate synchronization between processes, let us add a token-passing mechanism by which individual starvation can be prevented.

Example 2. We add to each fork process a local Boolean variable *token*, which is true when the fork has the token. The fork process is now

process *Fork*[*i*:1 · · · *n*];
  var *token*: *boolean*;
  token := (*i* = 1);

Initially, the token is at *Fork*[1]. The eating action is changed to

```
action Eating[i: 1 ··· n] by Philosopher[i], Fork[i], Fork[i + 1]:
Philosopher[i].hungry ∧ ¬Fork[i + 1].token →
  Eat;
  Philosopher[i].hungry := false;
  if Fork[i].token
  then Fork[i].token, Fork[i + 1].token := false, true;
```

This mechanism will prevent the philosopher whose right fork has a token from eating, thus giving his right neighbor a chance to eat. The eating action will pass the token from the left to the right fork of the philosopher, if there is a token at the left fork. The synchronization of the eating action now involves conditions on local variables in both the philosopher process and in the right fork process.

The example illustrates how process communication is implicit in the guard and in the action bodies. The processes are *synchronized* by the guard: the eating action, for instance, is only executed if the philosopher is hungry, the forks are available, and the right fork does not have a token. An arbitrary number of processes may be synchronized by a joint action: the eating action synchronizes three processes. *Information transfer* is achieved with the action body: executing the eating action passes the token from the left fork to the right fork. No explicit synchronization or communication primitives are needed in the language.

Transfer of information is *symmetric* in the action system mechanism, in the sense that all processes participating in an action are treated in the same way. Each process can use the local variables of the other processes to update their own local variables. Usually processes in a communication event are treated asymmetrically. In CSP, for instance, information is only transferred from the sender to the receiver. In Ada, input information is transferred from caller to callee, while output information is transferred from callee to caller.

Another difference has to do with the possibility to choose between alternative communications. In the original CSP [25] and in Occam, this is restricted to the receiver (unless output guards are allowed), in Ada to the callee. (The restriction on output guards is removed in [26], in which CSP is treated from a proof and model theoretic point of view, and where implementation efficiency therefore is not a prime concern.) In action systems, each process can be involved in many different actions, any number of which can be simultaneously enabled. Each process involved in an action may thus be in a position to choose between alternative actions. The communication mechanism is thus symmetric also in this aspect. The action mechanism can therefore be characterized as a generalization of the communication mechanisms of CSP and Ada to *symmetric multiprocess handshaking*. This communication mechanism can be implemented in an efficient manner, as shown in Section 3 below.

1.3 Reasoning about Action Systems

The close resemblance of action systems to the guarded iteration statements and production systems suggests that one could reason about the behavior of an action system as if it were executed in a sequential manner. Actions would be

executed one at a time, such that at each step one of the enabled actions would be chosen nondeterministically and executed to completion before selecting the next action. This is the way in which the guarded iteration statement is executed. The advantages of being able to use this abstraction would be considerable. The sequential execution model is simpler to reason about, and it has a well-developed proof theory. For action systems that are intended to terminate one can use Hoare logic [24] or the weakest precondition technique of Dijkstra [15]. For continually running action systems (*reactive systems*, in the terminology of Pnueli [32]), temporal logic provides a suitable framework for reasoning, along the lines of Manna and Pnueli [31].

The main problem to be considered in this paper is to what extent such a simple sequential (and nondeterministic) execution model can be used in reasoning about actions systems, when the actions are in fact executed in parallel, in a distributed fashion. The two models of execution lead to different views of when an action can be considered enabled. In a sequential execution an action is enabled if its guard is true. In a concurrent execution this is not sufficient; we must also require that all processes needed for the action be free to participate in it (i.e., they are not engaged in any other concurrently executing action).

This difference shows up in the different notions of fairness that one might reasonably expect to hold for an execution. If actions are executed in a sequential manner, then fairness for actions means that if the guard of an action is infinitely often true, then the action is infinitely often executed. We refer to this notion of fair sequential execution as *action fairness*. As an example, action fairness for the eating actions in Example 1 is sufficient to guarantee absence of individual starvation: each philosopher that becomes hungry will sooner or later be able to eat.

If actions are executed in a concurrent and distributed manner, then the fairness notion becomes the following: If infinitely often the guard of an action is true *and* all processes needed for the action are free to participate in it, then the action is infinitely often executed. We refer to this notion of fair concurrent execution as *handshake fairness*. An even weaker notion of fairness requires only that processes are treated fairly: a process that infinitely often is able to participate in some action should infinitely often do that. This is referred to as *process fairness*. Sequential and concurrent executions thus have different notions of fairness, and hence properties that hold for sequential executions of action systems need not be valid for concurrent executions of the same action systems.

Handshake fairness for the eating action in Example 1 above would mean the following: If infinitely often the eating action is enabled (the philosopher is hungry *and* both his forks are free), then the eating action will take place infinitely often. Handshake fairness is not sufficient to prevent individual starvation. If the two neighbors of a philosopher eat alternately, such that at each moment one of the neighbors is eating, then there is never any moment when the philosopher in the middle has both forks simultaneously available. The computation is handshake fair, even if the philosopher never gets to eat.

Process fairness for the philosophers in Example 2 means that if infinitely often the philosopher is free to engage in some action (eating or thinking), then it will infinitely often participate in some action. In the example, process fairness

is sufficient to guarantee that the philosopher gets to eat infinitely often (assuming that eating and thinking actions are always finite).

1.4 Overview of Paper

The content of the rest of the paper is as follows. Action systems are defined formally in the next section. We define execution of an action system in a way that corresponds to the execution of guarded iteration statements. We refer to this execution model as the *sequential execution model.* We will show how to prove properties of action systems in the temporal logic framework. For liveness properties we assume action fairness for specific actions.

The implementation of action systems will be described in Section 3. We first present a *concurrent execution model.* This model corresponds to a parallel and distributed execution of actions in a system. We will show how this model can be implemented in a distributed fashion. As a concrete example we consider the implementation of this execution model on a local area network with reliable broadcasting. All processes are assumed to receive the broadcast messages in the same order. The unique ordering of messages makes it possible to build an efficient distributed execution mechanism for actions. We show that the implementation is correct with respect to the concurrent execution model.

Two different execution models for action systems are thus given, the sequential model in Section 2 and the concurrent model in Section 3. The sequential model is the one we want to reason about, while the concurrent model is the one that we are able to implement efficiently in a distributed fashion. The relationship between these two models is studied in Section 4. The main problem considered is the following: to what extent can the abstraction of sequential execution be kept up when execution is actually done according to the concurrent execution model? In other words, to what extent are properties that hold for a sequential execution of an action system valid also for a concurrent execution of the action system?

We show that as far as safety properties go, this abstraction can be preserved without problems. However, for liveness properties this is not the case, because the notion of fairness that we want to use in the sequential model (action fairness) is stronger than the fairness notions that are reasonable in the concurrent execution model (handshake fairness or process fairness). The latter notions are restricted by what can be guaranteed in a distributed system without introducing a centralized scheduling mechanism for actions.

This leads us to study the fairness notions in the two execution models in more detail. Computations in the concurrent execution model will be shown to simulate the computations in the sequential execution model. This allows us to relate the computations in the two models to each other, and to define action fairness for computations in the concurrent model. We will say that an action system is *fairly serializable,* if action fairness is guaranteed by handshake or process fairness in the concurrent execution model. If the action system is fairly serializable, then each fair computation in the concurrent model simulates an action fair computation in the sequential execution model.

We will define subclasses of action systems that are fairly serializable. These subclasses are defined both by static properties of action systems and by dynamic

properties of the execution of action systems. The former can be established by proving properties of the guards and bodies of individual actions, the latter by proving that the execution of the action system satisfies certain temporal logic formulas. In practice, proving that a certain action system is fairly serializable amounts to proving that certain forms of conflicts and conspiracy among processes cannot occur during execution of the system.

In Section 5 we then consider how to prove properties of concurrent executions of action systems. We want to establish these properties by reasoning only about sequential executions of action systems. For safety properties this does not present any problem, as each safety property that holds for sequential execution of an action system will also hold for concurrent execution of this system. Liveness properties are proved in two stages. One first proves that any sequential execution satisfies the required liveness property, assuming that the necessary assumptions about action fairness are valid. One then proves that these fairness assumptions hold in a concurrent execution, by proving that the action system is fairly serializable for these actions.

We will also show that in the latter case it is sufficient to consider only sequential executions of action systems. By introducing the notion of serializability we are thus able to establish both safety and liveness properties of distributed executions of an action system using only the sequential execution model in our reasoning. We may thus ignore that the action system is, in fact, executed in a distributed fashion.

The concept of a *global state* is well defined in the sequential and concurrent execution models, but it is not clear what it really corresponds to in an actual distributed execution of an action system, i.e., whether the global states of a concurrent computation can in fact be observed in a real distributed implementation of the action system. We will analyze this issue more closely and show that if a temporal property holds in the concurrent execution model of an action system, then the property can also be observed to hold in an actual distributed execution of the system.

2. ACTION SYSTEMS

We will here define more precisely what an action system is and how it is executed. We will use the temporal logic framework of Manna and Pnueli [31] to define action systems and their executions.

A *transition system* is a quadruple $(\Sigma, \Sigma^0, T, \mathcal{F})$ consisting of a set of *states* Σ, a set of *initial states* $\Sigma^0 \subseteq \Sigma$, a finite set of *transitions* T, and a *fairness family* $\mathcal{F} = \{F_1, \ldots, F_n\}$, $F_i \subseteq T$. Each transition $t \in T$ is a partial function $t: \Sigma \to \Sigma$. Transition t is said to be *enabled* in a state $\sigma \in \Sigma$, if $t(\sigma)$ is defined.

A *computation* in a transition system is a finite or infinite sequence c of the form

$$c: \quad \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots \sigma_{i-1} \xrightarrow{t_i} \sigma_i \cdots, \quad \sigma_i \in \Sigma, \quad t_i \in T,$$

which satisfies the following four conditions:

(1) The computation is *properly initialized*, i.e., $\sigma_0 \in \Sigma^0$.

(2) The computation is *admissible*, i.e., for each transition t_i in c, t_i is enabled in σ_{i-1}, and $t_i(\sigma_{i-1}) = \sigma_i$.

(3) If finite, the computation is *properly ending*, i.e., no transition in T is enabled in the final state σ_n.

(4) If infinite, the computation is *fair* with respect to each fairness set $F \in \mathcal{F}$. This means that the following condition holds: If c contains an infinite number of states σ_i in which a transition $t \in F$ is enabled, then c contains an infinite number of transitions in F.

One could also impose the weaker requirement of *justice* (or *weak fairness*) with respect to certain sets of transitions [31]. For simplicity we only deal with fairness in this paper.

2.1 Formal Definition of Action Systems

An *action system* is a tuple $S = (Proc, Act, \mathbf{Var}, \mathbf{Init}, \mathbf{Actions})$ where the components are as follows:

(1) *Proc* is a nonempty set of *process names*.

(2) *Act* is a nonempty set of *action names*.

(3) $\mathbf{Var} = \{y_p \mid p \in Proc\}$ is a partitioning of the *program variables* into *local variables* y_p of the processes $p \in Proc$. We write $y = \cup\mathbf{Var}$ for the set of all program variables in the system.

(4) **Init** is a collection of *process initialization* statements $S_p(y_p)$, $p \in Proc$, that assign initial values to the local variables y_p.

(5) **Actions** is a collection of *actions* of the form

$$\textbf{action } a \textbf{ by } Proc_a\colon g_a(y_a) \rightarrow S_a(y_a), \quad a \in Act.$$

Here $Proc_a \subseteq Proc$ is the set of processes *participating* in the action a, $y_a = \cup\{y_p \mid p \in Proc_a\}$ is the set of local variables of these processes, the Boolean condition $g_a(y_a)$ is the *action guard*, and the statement $S_a(y_a)$ is the *action body*.

We will assume that all initialization statements S_p and action bodies S_a are *deterministic*. Initialization statements are assumed to always terminate, while action bodies are assumed to terminate whenever the guard is satisfied. In the sequel we usually model the effect of an initialization statement S_p by an assignment $y_p := c_p$, where c_p is some list of constants that matches the list of variables y_p. Similarly, we will model the effect of an action body S_a by an assignment $y_a := f_a(y_a)$, where f_a is some (list of) function(s) that gives the effect of executing the action body.

An action system S may be constrained by an *initial condition* Q on the program variables, which must be established by the initialization of the action system. Processes without initialization statements may assign any initial values to their local variables that are consistent with the given initial condition. If no initial condition is indicated, then it is assumed to be *true*.

A guard of the form $g(y_p)$, $p \in Proc$, is called a *local guard* of p, as it only refers to the local variables of process p. A guard is said to be *separable* if it is a Boolean combination of local guards. An action system is said to be *separable* if each action guard in the system is separable. Separability makes it easier to implement action systems in a distributed fashion, as shown in Section 3. The semantic and

proof theoretic analysis does not depend on this assumption, so it will only be made in that section.

The *sequential execution model* for an action system S, assuming *action fairness* for the actions in $A \subseteq Act$, is the transition system $Seq_S(A) = (\Sigma_y, \Sigma_y^0, T, \mathcal{F}_A)$ where

(1) each state $\sigma \in \Sigma_y$ is an assignment of values to the program variables y, i.e., $\sigma: y \rightarrow D$ for some fixed domain of values D.
(2) Σ_y^0 is the set of all initial states σ_0 such that $\sigma_0(y_p) = c_p$ for each p for which an initialization statement $y_p := c_p$ has been provided in **Init**,
(3) $T = \{act_a \mid a \in Act\}$ is a set of *action transitions*, and
(4) $\mathcal{F}_A$ consists of all singleton sets $\{act_a\}$, $a \in A$.

An action transition $act_a: \Sigma_y \rightarrow \Sigma_y$ is enabled in a state $\sigma \in \Sigma_y$ if the guard $g_a(y_a)$ of action a is true in σ. It maps the values of the program variables y_a to $f_a(y_a)$, leaving the other program variables unchanged. If an initial condition Q is given, then the set of initial states is further restricted to those states σ_0 that satisfy Q.

A computation c in $Seq_S(A)$ is said to be *action fair* for action a, if c is fair with respect to $\{act_a\}$. By definition, each computation of $Seq_S(A)$ is action fair for every action in A.

Action fairness is a strong requirement. In our first example, action fairness for eating actions is sufficient to prevent a philosopher from starvation. If *Philosopher*[i] is hungry, then the eating action will be enabled, and it stays enabled until the philosopher has eaten. Action fairness requires that the philosopher will eat sooner or later.

No concurrency is involved in the sequential execution model. It is therefore the easiest way of thinking and reasoning about action systems. Actions are executed one at a time, starting from a given initial state, as long as there are enabled actions to execute. At each stage, one of the enabled actions is chosen nondeterministically for execution. The computation terminates when there are no enabled actions any more. This, in essence, is the way in which the guarded iteration statement of Dijkstra is executed. The difference is that we are also interested in infinite computations, in which case we also require action fairness of the computation, i.e., we require that the alternatives in the guarded iteration statement must be chosen in a fair manner [21].

2.2 Proving Properties of Sequential Executions

We will here show how to prove properties of action systems within the temporal logic framework. The actions are identified with the transitions in the sequential execution model. Hence one reasons about the behavior of an action system directly in temporal logic, by considering all possible sequences of actions, subject to the given action fairness constraints. The proof rules and proof methods of [31] carry over directly to action systems, and can be used as such in establishing their temporal properties. (Other expositions of the temporal logic approach are given in [28, 32].) This close correspondence with the temporal logic framework is one of the main advantages of and motivations for using the sequential execution model for action systems.

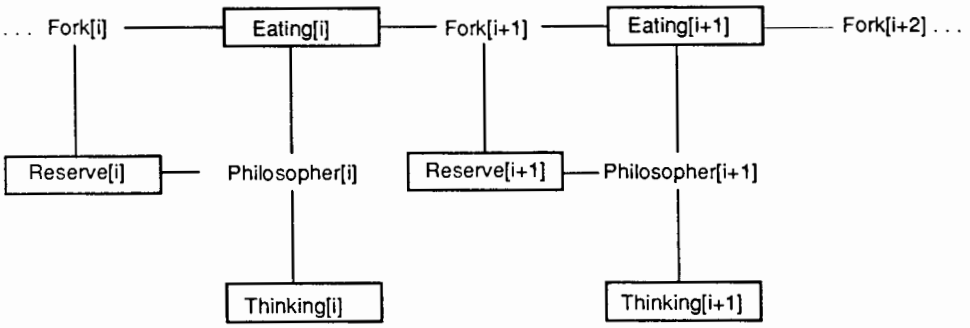

Fig. 2. Dining philosophers with reserve actions.

Example 3. We consider the solution with token passing in Example 2, but modify it in such a way that a philosopher has to give preference to his right neighbor only when the latter has made a reservation for his left fork by a special reserve action. When the token eventually arrives at this fork, the left neighbor is then forced to give the right neighbor a chance to eat, provided that the latter has not eaten since he reserved the left fork. The resulting action system is illustrated in Figure 2.

The philosopher processes are as before, while the fork processes are extended with local variables for the reservation. Initially the first fork has the token, and no fork is reserved.

process *Philosopher*[*i*:1 . . *n*];
  var *hungry*:*boolean*;
  hungry := *false*;

process *Fork*[*i*:1 . . *n*];
  var *token*, *reserved*:*boolean*;
  reserved := *false*;
  token := (*i* = 1);

The thinking action is unchanged, while the eating action must be adapted to take token passing and reservations into account. An additional action is introduced by which a philosopher reserves his left fork.

action *Thinking*[*i*:1 . . *n*] **by** *Philosopher*[*i*]:
¬*Philosopher*[*i*].*hungry* →
  Think;
  Philosopher[*i*].*hungry* := *true*;

action *Reserve*[*i*:1 . . *n*] **by** *Philosopher*[*i*], *Fork*[*i*]:
Philosopher[*i*].*hungry* ∧ ¬*Fork*[*i*].*reserved* →
  Fork[*i*].*reserved* := *true*;

action *Eating*[*i*:1 . . *n*] **by** *Philosopher*[*i*], *Fork*[*i*], *Fork*[*i* + 1]:
Philosopher[*i*].*hungry* ∧ ¬(*Fork*[*i* + 1].*token* ∧ *Fork*[*i* + 1].*reserved*) →
  Eat;
  Philosopher[*i*].*hungry* := *false*;
  if *Fork*[*i*].*token*
  then *Fork*[*i*].*token*, *Fork*[*i* + 1].*token* := *false*, *true*;
  Fork[*i*].*reserved* := *false*;

We prove properties of this action system by reasoning about the possible sequences of actions that can occur during a computation of the system, subject to the given action fairness assumptions. A safety property $\Box I$ states that condition I holds throughout the computation of the action system, i.e., I is an *invariant* of the system. This is established by proving that

(i) the initializations together establish the invariant, i.e.,

$$true\{S_{p_1}; \ldots; S_{p_n}\}I,$$

where $p_1, \ldots, p_n$ are the processes for which initializations have been provided, and

(ii) each action preserves the invariant, i.e.,

$$I \wedge g_a\{S_a\}I$$

holds for each action $a \in Act$.

Here $P\{S\}Q$ stands for the partial correctness of statement S with respect to precondition P and postcondition Q [24].

An example of an invariant property is that there is always exactly one fork with an associated token in the action system described above, i.e.,

$$\sum_{i=1}^{n} Fork[i].token = 1$$

holds throughout the execution of the action system. This is obviously established by the initializations of the local variables and it is preserved by each action execution.

Our definition of action systems requires that each initialization statement and action body terminates if executed. If we require total correctness rather than partial correctness in (i) and (ii), then this property is established at the same time as establishing the invariance of the given condition.

Liveness properties are also proved in the usual way, using permitted fairness assumptions when necessary. As an example, consider the property that each philosopher infinitely often gets hungry:

$$\Box\Diamond Philosopher[i].hungry,$$

for each $i = 1, \ldots, n$. This follows directly from the fact that whenever $Philosopher[i]$ is not hungry the thinking action is enabled. If action fairness is assumed for thinking actions, then this action will be eventually executed. The thinking action will then necessarily terminate, making the philosopher hungry. Liveness in this case is straightforward to establish.

Proving that the action system is starvation free is more complicated. Starvation freedom is expressed by the property

$$\Box(hungry_i \Rightarrow \Diamond \neg hungry_i),$$

for $i = 1, \ldots, n$. (For brevity, we write $hungry_i$ for $Philosopher[i].hungry$ and similarly for the other local variables in the system.) We prove this property by using the method of well-founded ranking [31]. In this method, the set of possible states is partitioned into a number of cases which forms a well-founded order in

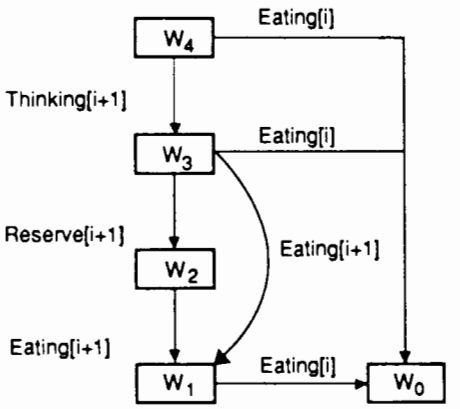

Fig. 3. Proof diagram for starvation freedom.

the sense that (i) no transition leads from a lower case to a higher case, and (ii) for each case for which there are lower cases, there is at least one transition that necessarily leads to a lower case. These transitions are called *helpful* transitions. Fairness for the helpful transitions guarantees that a case at the bottom in this order is eventually reached.

Assume that philosopher i is initially hungry. We then have two cases: either his right fork has a token or it does not. If it does have a token, there are again two cases: either the right fork is reserved or it is not. Finally, in the latter case we have two cases: either philosopher $i + 1$ is hungry or he is not. This gives us altogether five possible cases to consider, which we denote by W_i, $i = 0, 1, 2, 3, 4$ (W_0 is the case when philosopher i is not hungry):

$$W_0 = \neg hungry_i,$$
$$W_1 = hungry_i \wedge \neg token_{i+1},$$
$$W_2 = hungry_i \wedge token_{i+1} \wedge reserved_{i+1},$$
$$W_3 = hungry_i \wedge token_{i+1} \wedge \neg reserved_{i+1} \wedge hungry_{i+1},$$
$$W_4 = hungry_i \wedge token_{i+1} \wedge \neg reserved_{i+1} \wedge \neg hungry_{i+1}.$$

We already noted above that there is exactly one token in the system. Hence, $W_i \Rightarrow \neg token_{i+2}$, for $i = 4, 3, 2$. Similarly, one can prove that fork i is only reserved if philosopher i is hungry, so $reserved_{i+1} \Rightarrow hungry_{i+1}$.

Figure 3 shows the helpful actions in these situations. In W_3, for instance, action $Eating[i]$ will lead directly to W_0 whereas action $Reserve[i + 1]$ will lead to situation W_2 and $Eating[i + 1]$ will lead to W_1. None of the other actions can change this situation. This shows that fairness for eating actions is sufficient to guarantee that starvation is impossible. Note that we do not really need action fairness for thinking and reserve actions to guarantee starvation freedom.

3. IMPLEMENTATION OF ACTION SYSTEMS

In this section we show how to implement an action system in a distributed fashion. We first define a transition model for the concurrent execution of actions. After this we show how this model can be implemented in a distributed fashion. We describe an implementation of action systems on a local area network

with reliable broadcasting, and show that the implementation is correct in the sense that it simulates the concurrent execution model of action systems.

3.1 Concurrent Execution Model

In order to model a parallel and distributed implementation of action systems, we must describe the real concurrency that is present in these. This is done by the *concurrent execution model* of an action system. The model is similar to the sequential execution model, in that actions are still treated as the active components that are executed. However, they now have to compete for the processes, which are treated as passive resources in the system. Real concurrency is modeled by indicating the start and the end of process participation in an action. An action will be enabled only if its guard is true and all the processes needed for its execution are free.

The *concurrent execution model* for an action system $S = (Proc, Act, \mathbf{Var}, \mathbf{Init}, \mathbf{Actions})$, assuming *process fairness* for the processes $P \subseteq Proc$ and *handshake fairness* for the actions $A \subseteq Act$, is a transition system $Con_S(P, A) = (\Sigma_{y,free}, \Sigma^0_{y,free}, T, \mathcal{F}_{P,A})$ where the components are defined as follows:

(1) The state of the transition system is determined by the variables y, together with a collection of Boolean variables $free = \{free_p \mid p \in Proc\}$, indicating for each process p whether it is free or currently involved in some action or initialization.

(2) $\Sigma^0_{y,free}$ is the set of all initial states σ_0 such that (i) σ_0 assigns the values c_p to the variables y_p for each p for which an initialization statement has been given, and (ii) $free_p = false$ for each $p \in Proc$.

(3) For each action $a \in Act$ there is a *handshake transition* hs_a in T, and for each process $p \in Proc_a$ there is a *release transition* rel_p in T.

(4) The fairness sets in $\mathcal{F}_{P,A}$ include all singleton sets $\{rel_p\}$, $p \in Proc$. In addition, for each $p \in P$ there is a process fairness set F_p and for each $a \in A$ a handshake fairness set F_a, as described below.

The handshake transition hs_a is enabled if $g_a(y_a)$ holds and $free_p = true$ for each $p \in Proc_a$. It maps the values of y_a to $f_a(y_a)$ and the values of $free_p$ to false, for each $p \in Proc_a$, leaving all other state components unchanged. A release transition rel_p is enabled if $free_p = false$. It maps the value of $free_p$ to true, leaving the other state components unchanged.

If an initial condition Q is given for the action system, then the set of initial states is further restricted to those states σ_0 that satisfy Q.

A handshake transition hs_a corresponds to a synchronizing handshake. It can be executed only if all processes needed for the action are free, $free_p = true$ for each $p \in Proc_a$, and willing to participate in the action a, $g_a(y_a) = true$. The effect of the action is considered to take place immediately, i.e., all variables receive their updated values by transition hs_a. The release transition rel_p releases process p from the action in which it is engaged. The initialization of the variables $free_p$ implies that each process starts by a release transition. Intuitively, this means that we do not assume that all processes are started up at the same time.

By making the start and the finish of process participation in the action into separate transitions, we are able to model the fact that executing a transition

takes time, and that at any specific moment in the execution of an action system some processes may be in the middle of executing an action, while other processes are free and waiting to be engaged in some new action.

We define two different notions of fairness in the concurrent model. Both of these can be achieved by a distributed implementation of an action system, as we will show below. The first notion is *process fairness*. This means that if a process is infinitely often able to make progress it is also infinitely often allowed to do that. Formally, process fairness for a process $p \in P$ means fairness with respect to the set of transitions

$$F_p = \{hs_a \mid a \in Act_p\}.$$

Here $Act_p = \{a \in Act \mid p \in Proc_a\}$ is the set of all actions in which process p can participate.

The second notion is *handshake fairness*. By this we mean that if the guard of an action is infinitely often true and all the required processes are free, then the action is also infinitely often executed. Formally, handshake fairness with respect to an action a means fairness with respect to the singleton set

$$F_a = \{hs_a\}.$$

As stated above, a computation is always assumed to be fair with respect to release transitions rel_p. This reflects our assumption that action bodies always terminate.

3.2 Distributed Implementation of Action Systems

There are essentially three problems involved in implementing an action system: (i) it must be detected when actions become enabled; (ii) the enabled actions must be scheduled for execution; and (iii) the action bodies must be executed. All this must be done in a distributed fashion.

We will assume that all action guards are separable. A separable guard $g_a(y_a)$ can be written in disjunctive normal form $g_a^1(y_a) \vee \cdots \vee g_a^k(y_a)$ where each disjunct is of the form

$$g_a^i(y_a) = \bigwedge_{p \in Proc_a} g_{a,p}^i(y_p),$$

i.e., it is a conjunction of local guards $g_{a,p}^i(y_p)$ of the processes p involved in the action. A conjunction of this form is called a *simple guard*. An action a,

action a **by** $Proc_a$: $g_a(y_a) \rightarrow S_a(y_a)$

can be replaced by actions $a^1, \ldots, a^k$, where a^i is

action a^i **by** $Proc_a$: $g_a^i(y_a) \rightarrow S_a(y_a)$,

$i = 1, \ldots, k$. This collection of actions has the same effect as the original action. Hence, we may in the sequel assume that this action splitting has been done, and that all action guards are simple.

A local guard only tests a condition on the local variables of one process. Hence, each process can test the truth of its own local guard without consulting the other processes. An action is then enabled if each process involved in the

action is free and has determined that its local guard is enabled. The first problem can now be solved by letting each process announce its willingness to participate in the actions for which its local guards evaluate to true when it becomes free from executing a previous action or the initialization statement. By listening to these announcements the processes in the system can find out when some specific action becomes enabled.

The second problem is to build a distributed mechanism to select among the enabled actions those that are to be executed. The main problem is what to do with actions that become enabled at approximately the same time, but which require some common process and hence exclude each other. There are different ways of constructing such a distributed scheduling mechanism, depending on the assumptions that one makes about the underlying communication network. This problem is essentially a generalization of the problem of implementing CSP with output guards [10]. We describe below a way of scheduling the enabled actions when the communication mechanism is assumed to be a reliable broadcasting channel.

The third problem is how to execute an action body. The body of an action is a sequential statement that may refer to local variables of all the participating processes. Conceptually, we think of the action body as being executed in the combined state space of the participating processes. This effect can be achieved by executing the body in a distributed fashion as follows. Each process keeps a copy of the local variables of all the other processes. When an action has been selected for execution, the processes involved in this action send the values of their local variables to all the other processes involved in the action, who update their local copies with the current values of these variables. Each process then executes the action body, using its own local variables and copies of the local variables of other processes. The local variables of the processes involved in the action will then be updated in the same way, as if the statement had been executed directly on the original local variables. (This solution obviously has room for improvement, but it is sufficient to show the basic idea.)

3.3 Implementation on a Local Area Network

Processors in a CSMA/CD network are connected by a common channel on which they can broadcast messages to all other processes in the system. A process waits for its turn to communicate until the channel is free (i.e., no other process is broadcasting on it) before it attempts to transmit. If two or more processes attempt to transmit at the same time, then they notice that the messages collide, and each of them will wait a random time before trying again. We will assume that the channel is reliable, so all processes will receive the same sequence of messages.

Assume now that an action system S = (*Proc*, *Act*, **Var**, **Init**, **Actions**) is given, and that each process is assigned to a different processor in the network. Each action is assumed to have a simple guard. In order to record the availability of processes for actions, each process p keeps an allocation table where it records for each action those processes that have announced their willingness to participate in the action. Each process will be in one of four phases: *ready*, *waiting*,

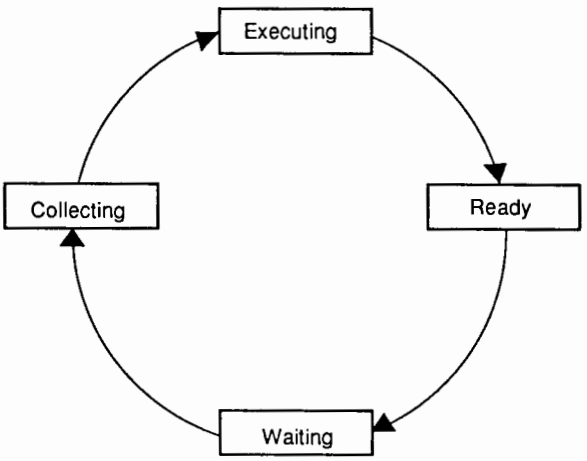

Fig. 4. Phases of implementation.

collecting, and *executing*. Initially all processes start up in the executing phase, where they perform their initialization statements. They then cycle through the four phases in the order listed above, as shown by Figure 4. The behavior of the processes in each of the phases is as follows:

Ready: The process evaluates its local guard for each action that it could participate in. Then it broadcasts a *willingness* message containing the names of those actions that it is willing to participate in. It goes into the waiting phase when it receives its own willingness message. While in the ready phase, the process does not record any messages from other processes.

Waiting: Willingness messages are observed only by those processes that are in the waiting phase. On receiving a willingness message, each of them updates its allocation table accordingly and then inspects it to see whether there is an action in which it could participate and for which also the other processes have announced their willingness. If so, the process tries to broadcast a *selection* message for such an action. For two processes, p is called *older* than q, if its last willingness message was sent earlier. A selection message for an action a can be sent only by the oldest process in $Proc_a$, because it is the only one that has seen the willingness messages of all the other processes in the action.

If two or more processes simultaneously try to transmit select messages, one of them will succeed first, and the others will cancel their selections upon receiving the conflicting select message. The action named in a selection message is marked as selected by all processes in the waiting phase. Those processes that are involved in this action go into the collecting phase, while the other waiting processes remove from their allocation tables the process names that are now committed to the selected action, and remain in the waiting phase. A process that has cancelled its own selection and remains waiting inspects the allocation table again. It tries to make a new selection if it finds actions that are still enabled.

Collecting: Each process involved in a selected action broadcasts a message containing the values of its local variables. A process remains in this phase until it has succeeded in sending this message and has received the corresponding messages from the other processes involved in the action. It then goes into the executing phase.

Executing: A process in the executing phase has sufficient information about the local states of all processes involved in the selected action to execute its own part of the joint transaction, i.e., to update its own local variables in the required manner. It does this as explained above, i.e., by executing a copy of the action body, using the values of the local variables of the other processes that it received in the previous phase. After this it returns to the ready phase, resets all entries in its allocation table to empty, and the cycle starts all over again.

The efficiency of the implementation can be measured in terms of the number of messages it requires. At least n messages must be sent for each executed action, where n is the number of processes involved in the action, as each process must at least announce its willingness to participate in subsequent actions. Our implementation requires $2n + 1$ messages for each executed action with n participating processes. This number can be reduced by sending a local state message only if this state is actually needed by some other process. If no transfer of information is needed, but only synchronization, then $n + 1$ messages are sufficient to schedule an action.

When a process selects a private action, its willingness and selection messages may seem superfluous. They are, however, necessary if one wants to guarantee a fair treatment of handshakes.

Actually, the number of messages required for each action could be reduced to $n + 1$ by including the values of the local variables in the willingness message of a process. This, however, would require that all processors listen continually to all messages sent over the network, and would also require more storage at processor nodes. The implementation above has the advantage that no communication overhead is associated with the execution phase.

An essential assumption in this implementation is that a process can cancel its attempt to transmit a select message when another select message is received. Although such an interconnection is not required to exist between the transmit and receive parts of the data link layer of local area network implementations [13], it seems reasonable that such a mechanism could be implemented without violating the existing standards. Alternative implementations of multiprocess handshaking that do not require this facility are described in [4].

3.4 Correctness of Implementation

In analyzing the correctness of the above implementation we first have to determine a unique *simulation mapping* of the sequence of broadcasting events in the implementation to computations in the concurrent execution model. We say that a sequence of broadcasting events *simulates* a concurrent computation, if each sequence of broadcasting events is mapped into some initial part of a computation, such that the mapping is monotonic with respect to the prefix

ordering of sequences, and if each completed sequence of broadcasting events is mapped to a (complete) computation.

More generally, let B and C be sets of sequences, and let B^* and C^* denote the prefix closures of these sets. The function $\gamma: B^* \rightarrow C^*$ is a *simulation mapping* if the following conditions hold:

(i) γ is monotonic, i.e., $b \leq b' \Rightarrow \gamma(b) \leq \gamma(b')$, where $\leq$ stands for the prefix ordering of sequences.

(ii) For each finite sequence $c \in C^*$ there is some finite $b \in B^*$ such that $\gamma(b) = c$.

(iii) For each $b \in B$, $\gamma(b) \in C$.

B *simulates* C if there is a simulation mapping $\gamma: B^* \rightarrow C^*$.

Notice the asymmetry that, although each sequence in B has to simulate a sequence in C, all infinite sequences in C need not be simulated. In the case of the above implementation, B consists of all possible complete message sequences in the implementation of an action system S, while C is the set of all (fair) concurrent computation sequences in $Con_S(P, A)$.

There is considerable freedom in determining the simulation mapping for an implementation. This freedom can be used to avoid unfair computations being simulated. With this in mind we choose the simulation mapping for our implementation as follows. Each select message is associated with the corresponding handshake transition, and each willingness message is associated with a release transition. The ordering of the transitions is determined by the order in which the willingness messages are broadcast. The mutual ordering of the releases is the same as that of the corresponding willingness messages. A handshake transition is, however, placed where the actual decision leading to the select message takes place, i.e., immediately after the willingness message of the last process to become free for the action, which is not necessarily at the place where the select message for the action was sent. Hence, the ordering of the handshake transitions, with respect to release transitions and other handshake transitions, is not necessarily the same as the ordering of the select messages with respect to the willingness messages and to the other select messages. The reason for this choice has to do with fairness, and will be made clear below.

No assumptions are made about the relative speeds of the processes. It is essential, however, that each processor be sufficiently fast, so that no relevant message is lost before being processed.

The fairness properties of the implementation depend on the assumptions that can be made of the communication channel. In the following it is essential that a process that infinitely often tries to transmit will also infinitely often succeed. Since collisions are resolved by random waits, this condition is met with probability one when a process is continually trying to send a message. This is the case with all willingness and local state messages.

Another kind of situation arises when a process tries to send a selection message as a response to some willingness message. The decision to send a selection requires inspection of the allocation table, and several processes might be simultaneously making competing decisions that exclude each other. In order

to give each process a fair chance in this competition, a random wait should be inserted between the decision and the sending of the select message. Such a wait is automatically included if the p-persistent carrier sense protocol [37] is used with $0 < p < 1$. With this technique the required condition is met with probability one also in the case of select messages.

Let us now define the simulation mapping γ of this implementation more precisely. Let b be a message sequence containing a willingness message w_p by process p. Consider the situation that w_p announces the willingness of p to participate in an action a such that all other processes in $Proc_a$ have announced their willingness in a already before p. There are now three possibilities: (i) If a is selected (on the basis of this w_p) in b, then w_p is said to be *satisfied by* a in b; (ii) if w_p is not satisfied by a, but each prefix of b containing w_p has an extension where this is the case, then w_p is said to be *open* in b; (iii) otherwise the selection of some other action precludes the possibility of selecting a (on the basis of this w_p).

Restricting to message sequences b' without open willingness messages, the transition sequence $\gamma'(b')$ is defined as the sequence obtained from b' by replacing

(1) each occurrence of a willingness message w_p that is satisfied by a in b by $rel_p\ hs_a$,
(2) every other willingness message w_p by rel_p, and
(3) all other messages by the empty sequence ϵ.

The states in $\gamma'(b')$ are uniquely determined by the initial states and the sequence of transitions. Obviously, γ' is monotonic, and the construction of the implementation guarantees that $\gamma'(b')$ is a properly initialized and admissible computation sequence, whenever b' is some prefix of a message sequence generated by the implementation.

For an arbitrary message sequence $b \in B^*$, let $\phi(b)$ be the maximal prefix of b with no open willingness message in b. The simulation mapping of the implementation is now defined as $\gamma(b) = \gamma'(\phi(b))$.

LEMMA 1. *The implementation of the action system S given in the preceding section simulates the concurrent execution model $Con_S(Proc, \emptyset)$, provided that the processors are sufficiently fast to handle each message in time.*

PROOF. First we check that each finite computation sequence $c \in Con_S(Proc, \emptyset)$ is simulated by some message sequence b. Such a b is obviously obtained from c if we replace each rel_p by an appropriate willingness message, and each hs_a by the corresponding selection and local state messages. The monotonicity of γ is clear, as it was constructed as the combination of two monotonic mappings. What remains to be shown is that each complete message sequence is mapped into a computation that is complete and fair with respect to each process $p \in Proc$.

A finite complete message sequence b must contain a last willingness message w_p for each $p \in Proc$. This can neither be open nor satisfied. This means that

no action is enabled after b, and $\gamma(b)$ is then a finite and properly ending computation.

For an infinite message sequence b we first notice that none of its willingness messages can be open for any actions a. Otherwise the oldest process in $Proc_a$ would continually try to send a selection message for a and would eventually succeed with probability one. This implies that $\gamma(b)$ must also be infinite. It remains to be shown that such a $\gamma(b)$ belongs, in fact, to $Con_S(Proc, \emptyset)$, i.e., $\gamma(b)$ is fair with respect to each process p in $Proc$.

Suppose that an infinite $\gamma(b)$ is not process fair, and let $Proc' \subseteq Proc$ be the (maximal) set of processes treated unfairly. There is then a prefix of b after which no process $p \in Proc'$ is transmitting any more. The last message of each $p \in Proc'$ is a willingness message; let p_0 be the oldest of these processes. Because of unfairness with respect to p_0, there is an action a such that $p_0 \in Proc_a$, and all processes $p \in Proc_a$ are infinitely often in a state where they are simultaneously free and have all indicated their willingness to participate in a. Since p_0 is the oldest in $Proc'$, and it eventually becomes older than any other process in $Proc_a$, it will infinitely often record all willingness messages and is therefore infinitely often trying to send a select message. This will succeed with probability one, which leads to contradiction. Hence, $\gamma(b)$ must be fair with respect to all processes $p \in Proc$. □

No assumptions were made in the protocol as to which select message is sent by a process when several alternatives exist. If an alternative is infinitely often possible, then a *fair decision* must also choose it infinitely often. Using random choice this can be achieved with probability one. Next we shall show that our implementation is handshake fair if all decisions are fair.

Suppose that a computation is not handshake fair with respect to an action $a \in Act$, and consider the suffix of the computation where a is no longer executed. It is now infinitely often the case that all processes $p \in Proc_a$ are free to participate in a and have also indicated their willingness to do this. Some process p_0 must also infinitely often be oldest among them. A fair decision by p_0 will now infinitely often choose a, leading to a contradiction. Hence, we conclude that the implementation with fair decisions ensures handshake fairness.

The reason for associating the handshake transition with the moment of decision and not with the moment when the actual select message is sent has to do with the fairness requirements. It is possible that the process p_0 above, when a specific willingness message is broadcast, already has made a decision to select some action a on the basis of some earlier willingness message, although it has not yet succeeded in sending the select message. The willingness message arriving after the decision has been made but before the select message is sent may then seemingly enable some new action a' (involving processes in the selected action) that p_0 cannot consider anymore. The way in which the simulation mapping was defined does not, however, consider this to be unfair, since the handshake transition $hs_{a'}$ is not considered to be enabled in such a situation.

Since the original actions may be split in the implementation to make the guards simple, we finally check that fair decisions also ensure handshake fairness with respect to the original actions. This is obviously the case, since handshake

fairness with respect to the split actions is a stronger requirement than handshake fairness with respect to the original actions.

LEMMA 2. *The above implementation of action systems is process fair and, if the select decisions in the processes are fair, then the implementation is also handshake fair.*

We can summarize the results of this section in the following theorem.

THEOREM 1. *The implementation of an action system S of Section* 3.3 *simulates the concurrent execution model* $Con_S(P, A)$ *(up to probability one), when the choice between actions in A is carried out by the processes in a fair manner.*

4. FAIR SERIALIZABILITY

In this section we study the relationship between the two models for execution of an action system, the sequential execution model of Section 2 and the concurrent execution model of Section 3. Our aim is to find out to what extent the sequential execution model is valid as a basis for reasoning about properties of the action system, when the computation is, in fact, done according to the concurrent execution model.

4.1 Serialization of Computations

In order to relate the sequential and the concurrent execution models to each other, we first define the *serialization* of a concurrent computation c,

$$c\colon \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots \sigma_{i-1} \xrightarrow{t_i} \sigma_i \cdots.$$

This is the sequence c' that is obtained by replacing each handshake transition $t_i = hs_a$ in c by the corresponding action transition act_a, by omitting all release transitions $t_i = rel_p$ together with their associated states σ_i, and by deleting the state component *free* from the remaining states.

In the sequel, we will talk about a *sequential computation* of an action system S, when we mean a computation in the sequential execution model $Seq_S(\varnothing)$, i.e., no action fairness assumptions are made for the computation. Similarly, a *concurrent computation* of an action system S is a computation in the concurrent model $Con_S(\varnothing, \varnothing)$ for which no process or handshake fairness assumptions are made. Whenever fairness assumptions are made for a computation, these will be explicitly stated.

Serialization is a simulation mapping from the set of concurrent executions of an action system S to the set of sequential executions of S. Actually, an even stronger result holds, which says that each sequential execution is simulated by some concurrent execution, if fairness requirements are not considered. More precisely, we have the following result.

LEMMA 3. *Let S be an action system. Then (i) the serialization of a concurrent computation of S is a sequential computation of S, and (ii) each sequential computation of S is the serialization of some concurrent computation of S.*

The proof of this lemma is straightforward given the definitions, and is therefore omitted here.

Action fairness is not a fairness property in the concurrent model. We can, however, define a requirement on computations in the concurrent model that corresponds to action fairness in the sequential model. We say that a concurrent computation c is *action fair* for action a, if the following condition is satisfied: if $g_a(y_a)$ holds infinitely often in c, then transition hs_a is taken infinitely often. The following is an immediate consequence of the definitions.

LEMMA 4. *A concurrent computation of an action system S is action fair for a if and only if its serialization is action fair for a.*

We write $\mathscr{AF}(a)$ to denote that each computation c of S is action fair with respect to a, and $\mathscr{AF}$, if $\mathscr{AF}(a)$ holds for each $a \in Act$. Similarly, we write $\mathscr{HF}(a)$ and $\mathscr{HF}$ in the case of handshake fairness. We write $\mathscr{PF}(p)$ to denote that each computation of S is process fair for process p and $\mathscr{PF}$, if $\mathscr{PF}(p)$ holds for each $p \in Proc$.

Let us say that a temporal logic formula R is *insensitive to stuttering*, if the following holds: R holds for a computation sequence c if and only if R holds for any computation sequence c' that can be constructed from c by adding or deleting a finite number of *stuttering transitions*. A stuttering transition is a transition that does not change the state (i.e., an identity function). A temporal logic formula that only uses the eventuality and always operators (and specifically does not use the next state operator) is an example of a formula that is insensitive to stuttering.

The concurrent execution model is a faithful implementation of the sequential execution model, as far as observation of changes in the state y are concerned. In fact, apart from possible observations of stuttering, any temporal logic property that holds for all the sequential executions of an action system will also hold for all concurrent executions, provided that the same action fairness assumptions hold.

THEOREM 2. *Any temporal property that holds in the sequential model $Seq_S(A)$ of an action system S and that is insensitive to stuttering, will also hold in the concurrent model $Con_S(\emptyset, \emptyset)$, provided that $\mathscr{AF}(a)$ holds for each $a \in A$. The converse is true if the temporal property does not refer to the state component free.*

PROOF. Let c be a concurrent computation in $Con_S(\emptyset, \emptyset)$ and let c' be the serialization of c. By Lemma 4, c is action fair for each action in A if and only if c' is action fair for each action in A. Let R be a temporal property that is insensitive to stuttering. Then R will be true for computation c if and only if R holds for the sequence of states $\sigma = \sigma_0, \sigma_1, \ldots$, generated by the computation c. This is the case if and only if R holds for the sequence of states σ^* that we get from σ by dropping the state component *free* from all states, because R can only refer to variables in the state component y. This in turn is equivalent to R being true for the sequence of states σ' that we get from σ^* by dropping all repetitions of states in the sequence, because R is assumed to be insensitive to stuttering. This finally holds if and only if R is true for the computation sequence c', as σ' is the sequence of states generated by the computation c'.

In conclusion, R holds for a sequential computation c' that is fair for actions A if and only if it holds for every concurrent computation c of which it is a

serialization and that is fair for the actions A. The required conclusion now follows by Lemma 3. □

The two models are thus otherwise equivalent, but their fairness notions are different. The natural fairness notion in the serial model is not a proper fairness notion in the concurrent model, and it leads to stronger requirements than it seems possible to enforce in distributed implementations.

The relationships between the three notions of fairness are as follows:

THEOREM 3. *Let S be some action system, and let a be an action in S. Then*

(*i*) *$\mathscr{AF}(a)$ in the concurrent execution model implies $\mathscr{KF}(a)$,*

(*ii*) *$\mathscr{KF}$ implies $\mathscr{PF}$,*

(*iii*) *neither implication holds in the reverse direction.*

PROOF. For part (i) consider any concurrent computation c that is not $\mathscr{KF}(a)$. Transition hs_a is then enabled infinitely often in a suffix of c without being executed. This implies also that g_a holds infinitely often and, hence, c is not $\mathscr{AF}(a)$.

For part (ii), if c is a concurrent computation that is not $\mathscr{PF}(p)$, then there must be an action $a \in Act_p$ such that hs_a is infinitely often enabled but never executed in a suffix of c, which means that c is not $\mathscr{KF}$.

Part (iii) is demonstrated by examples. For part (i) consider the possibility for starvation in the concurrent execution of the action system in Example 1. For part (ii), the action system of Example 3 allows concurrent computations where all philosophers eat infinitely often but none of the reserve actions are ever executed. These computations are obviously $\mathscr{PF}$ but not $\mathscr{KF}$. □

The problem is that these implications hold in the wrong direction. What we actually need is that those fairness notions that can be guaranteed in a distributed implementation (i.e., process fairness and handshake fairness) imply the fairness notion that we want to use in reasoning about action systems, i.e., action fairness. This, however, is not true in general, as shown by the theorem.

Different solutions are possible to this problem. One solution is to accept some form of centralized scheduling, and in this way build implementations with stronger fairness properties, such that action fairness can be guaranteed. Another solution is to abandon the sequential execution model, and use the concurrent execution model instead, together with the fairness notions provided with this model. Either way, we would give up something quite important, either the distributedness of our communication mechanism or the ease of reasoning that the sequential model gives us.

We will here consider a third solution to this problem. We will try to identify those classes of action systems for which handshake or process fairness are sufficient to guarantee action fairness in the concurrent model. More formally, let us say that an action system S is *fairly serializable* for an action $a \in Act$, if $\mathscr{AF}(a)$ holds in the concurrent execution model $Con_S(P, A)$.

Fair serializability of an action system will be a consequence of the specific way in which the actions interact with each other. In essence, it means that there is some mechanism in the system that prevents unfair executions from occurring.

Thus, given a specific action system, we may either prove that the action system is fairly serializable for the set of actions for which we need action fairness or, if this is not the case, we might try to transform the action system to another one, which in all important aspects is similar to the original one, but which is also fairly serializable. In the latter case, we are essentially building a scheduling mechanism into the action system, to guarantee the kind of action fairness we want.

4.2 Static Conditions for Serializability

There are three related phenomena that can prevent process fairness from ensuring handshake fairness or action fairness, or prevent handshake fairness from ensuring action fairness in the concurrent execution model. These will be termed *competition*, *omission*, and *conspiracy*. We will in the sequel characterize these phenomena formally, and give conditions that guarantee that they cannot occur. We consider the possibility for these phenomena at two different levels, *static* and *dynamic*. The static level is concerned with membership properties of the sets $Proc_a$ and with predicates concerning effects of single actions in the space of all states Σ_y. Predicates about the possible execution sequences of actions belong to the dynamic level and involve temporal analysis. Here we shall consider the static possibilities for competition, omission, and conspiracy. The results are extended to the dynamic level in the next subsection.

Let us first introduce some notations. Two actions *exclude* each other, i.e., cannot be executed simultaneously, if they have some process in common. The set of actions excluded by an action $a \in Act$ is denoted

$$Exclude_a = \{b \in Act \mid a \neq b,\ Proc_a \cap Proc_b \neq \emptyset\}.$$

Within this syntactically determinable set we define the subsets consisting of actions that may *compete* with a, *enable* a, *disable* a, or *keep* a enabled:

$$\begin{aligned}
Compete_a &= \{b \in Exclude_a \mid \exists y \in \Sigma_y\colon g_a(y) \wedge g_b(y)\},\\
Enable_a &= \{b \in Exclude_a \mid \exists y \in \Sigma_y\colon g_b(y) \wedge \neg g_a(y) \wedge g_a(f_b(y))\},\\
Disable_a &= \{b \in Exclude_a \mid \exists y \in \Sigma_y\colon g_b(y) \wedge g_a(y) \wedge \neg g_a(f_b(y))\},\\
Keep_a &= \{b \in Exclude_a \mid \exists y \in \Sigma_y\colon g_b(y) \wedge g_a(y) \wedge g_a(f_b(y))\}.
\end{aligned}$$

Competition. By competition we understand the possibility that several actions are simultaneously enabled and compete for overlapping sets of processes. Absence of competition for an action is a strong requirement, implying that the processes involved in the action have no other alternatives. In Example 1 it is the competing eating actions that cause problems; private thinking actions have no competition.

An action $a \in Act$ is defined to be *statically competition safe*, if

$$Compete_a = \emptyset,$$

i.e., if no other action is ever competing with a. We now have the following result.

LEMMA 5. *$\mathcal{PF}$ guarantees $\mathcal{AF}(a)$ for an action $a \in Act$, if a is statically competition safe.*

PROOF. Consider a suffix of a concurrent computation where g_a holds infinitely often for a statically competition safe a, but hs_a is never taken. Since g_a can be turned false only by an action competing with a, g_a will stay continually true. If some process $p \in Proc_a$ is engaged in another action, it will eventually be released by rel_p, after which competition safeness of a guarantees that p will stay free. All $p \in Proc_a$ will thus eventually become free, and $\mathscr{PF}$ then guarantees the execution of hs_a, leading to a contradiction. □

Omission. If an enabled action involves at least one process that has no alternative actions to choose from, then $\mathscr{PF}$ with respect to this process guarantees that the action is not neglected forever. Omission is the phenomenon that, in the absence of such a process, an action is indefinitely neglected by giving systematically preference to some other actions occupying the same processes. In Example 3 the reserve actions have no such dedicated process and their omission is therefore possible.

An action $a \in Act$ is defined to be *statically omission safe*, if there is a process $p \in Proc_a$ such that either p is not involved in any other action, i.e.,

$$Act_p = \{a\},$$

or none of the actions requiring p is competing with a, and a is also not disabled by any other action, i.e.,

$$Act_p \cap Compete_a = \emptyset, \quad \text{and}$$
$$Disable_a = \emptyset.$$

We have the following result.

LEMMA 6. *$\mathscr{PF}$ guarantees $\mathscr{HF}(a)$ for an action $a \in Act$, if a is statically omission safe.*

PROOF. Consider a suffix of a concurrent computation where infinitely often g_a is true and all $p \in Proc_a$ are available for action a, but hs_a is not taken. By definition, statical omission safeness of a implies the existence of a process $p \in Proc_a$ for which the computation is not process fair. □

Conspiracy. By conspiracy we understand the phenomenon that an action is prevented from execution by two or more other actions, each holding in turn some of the processes that the action needs, so that all the processes are never simultaneously available. Starvation of a philosopher in Example 1 is caused by a conspiracy of the eating actions of his two neighbors.

Let us consider static safeness conditions against conspiracy. Conspiratorial situations against an action $a \in Act$ involve an overlapped execution of a sequence of at least two actions, each holding some processes in $Proc_a$. A finite or infinite sequence of actions $\xi = (b_1, b_2, \ldots)$ for which such a behavior is possible has to satisfy the syntactic conditions

$$b_i \in Exclude_a,$$
$$Proc_a \cap Proc_{b_i} \not\subseteq Proc_{b_{i+1}}.$$

The latter condition states that it is possible for action b_i to release all the processes that b_{i+1} needs, before releasing some process which action a needs.

Thus, action b_{i+1} can start before action a becomes enabled. Such a sequence ξ will be called a *cover* of a.

Consider now a situation where an action $a \in Act$ is forever prevented from execution by the conspiratorial behavior of other actions. Then each sequence of states where g_a holds must be included in a cover of a. Two different situations arise, depending on whether g_a holds continually or not.

First, if g_a holds indefinitely from some point on, there must be an infinite cover. This has to contain elementary cyclic covers $\xi = (b_1, \ldots, b_k, b_1)$ consisting of k different actions, $k \geq 2$, that satisfy the static conditions

$$b_i \in Keep_a, \quad i = 1, \ldots, k.$$

Such a finite cover is called a *conspiratorial cycle* against a.

Secondly, if g_a is infinitely often turned on and off, then each interval when g_a is enabled must be included in a finite cover. The last action b_k of such a cover turns g_a off; it is turned on either by the first action b_1 of the cover or by an action b_0 that starts between b_1 and b_2 but is not part of the cover. This leads to the definition of a *conspiratorial chain* $\xi = (b_0, b_1, \ldots, b_k)$, $k \geq 2$, against a as a sequence where $(b_1, \ldots, b_k)$ is a cover of a not containing internal cycles, $b_0 = b_1$ or $Proc_a \cap Proc_{b_1} \not\subseteq Proc_{b_0}$, and the following static conditions are satisfied:

$$b_0 \in Enable_a,$$
$$b_i \in Keep_a, \quad 1 < i < k,$$
$$b_k \in Disable_a.$$

An action $a \in Act$ is defined to be *statically conspiracy safe*, if there is no conspiratorial cycle or conspiratorial chain against a.

The following lemma follows from the above analysis of conspiracy situations.

LEMMA 7. *$\mathscr{KF}(a)$ for an action $a \in Act$ guarantees $\mathscr{AF}(a)$, if a is statically conspiracy safe.*

Obviously static competition safeness is stronger than the two other properties, implying both static omission safeness and static conspiracy safeness.

Example 4. We show how static competition or conspiracy safeness could be imposed on the eating actions in Example 1. Let us attach a new variable to each fork, indicating a "left" or a "right" fork. If eating is allowed only with the proper forks in both hands, and the two forks are always exchanged after eating, the system is obviously competition safe. Hence process fairness guarantees that action fairness holds for the eating actions, by Lemma 5. This means that all actions are treated fairly in the system.

The sequential model with $\mathscr{AF}$ can then be used to prove that starvation is not possible, unless all forks are of the same kind. Starvation freedom is expressed by the property

$$\Box(Philosopher[i].hungry \Rightarrow \Diamond \neg Philosopher[i].hungry).$$

This property is not sensitive to stuttering. Hence, by Theorem 2, if this property holds in the sequential execution model assuming action fairness for all actions, it will also hold in the concurrent execution model, when the assumption of

action fairness is satisfied. Hence process fairness is sufficient to guarantee that starvation is not possible.

The static criteria for competition, omission, and conspiracy safeness are quite strong, and there are many situations in which they would not be applicable. These criteria are therefore to be taken as a first check, by which one identifies those cases for which more refined methods are required.

4.3 Temporal Conditions for Fair Serializability

The discussion in the previous section suggests that static conditions for serializability are too strong to handle all cases that can occur in practice. We therefore proceed to extend the static analysis to a *temporal* or *dynamic* analysis of action systems. While static safeness conditions ensure that certain unwanted situations cannot possibly arise, the dynamic conditions state that they do not actually occur in system executions.

The temporal expressions for our dynamic conditions will be liveness properties in *reduced* action systems where some of the original actions are omitted. Let

$$S = (Proc, Act, \textbf{Var}, \textbf{Init}, \textbf{Actions})$$

be an action system. We define the *reduction* of S by actions $A \subseteq Act$ to be the action system

$$S \backslash A = (Proc, Act - A, \textbf{Var}, \emptyset, \textbf{Actions}')$$

where **Actions**′ is constructed from **Actions** by deleting the actions in A. Note that there are no initialization statements in the reduced system, so all processes start with arbitrary values for their local variables. (Actually, the initial values of the process variables will in the sequel always be restricted by some additional initial condition that is imposed on the reduced action system.)

We modify some of the definitions to cover collections of actions instead of single actions. For an arbitrary set of actions $A \subseteq Act$, we write g_A for the disjunction $\bigvee_{a \in A} g_a$. We also extend $\mathcal{AF}(a)$ and $\mathcal{KF}(a)$ to cover sets of actions. By $\mathcal{AF}(A)$ we understand that some action in A is executed infinitely often if g_A holds infinitely often. Similarly, $\mathcal{KF}(A)$ means that some action in A is executed infinitely often, if some transition in $\{hs_a \mid a \in A\}$ is infinitely often enabled.

Notice that $\mathcal{AF}(A)$ and $\mathcal{KF}(A)$ do not imply $\mathcal{AF}(a)$ or $\mathcal{KF}(a)$ for any single action $a \in A$. The converse does, however, hold. More generally, we have that $\mathcal{AF}(A)$ and $\mathcal{AF}(B)$ implies $\mathcal{AF}(A \cup B)$, and similarly for handshake fairness.

Competition. Consider a situation where actions $A \subseteq Act$ are not executed from some point on, although g_A holds infinitely often. Assuming $\mathcal{PF}$ it must then be the case that some competing action $b \notin A$ is executed infinitely often instead. This kind of competition is not possible, if $g_A \wedge g_{ba}$ eventually becomes and remains false unless some action in A is executed.

More formally, a set of actions $A \subseteq Act$ is *competition safe* in action system S, if there is an invariant I of S such that the implication

$$\Box\Diamond g_A \Rightarrow \Diamond\Box\neg(g_a \wedge g_b)$$

holds for all $a \in A$, $b \in \text{Compete}_a - A$ in the reduction $S \backslash A$ with initial condition I, when $\mathscr{AF}(b)$ is assumed. We now have the following result.

LEMMA 8. *$\mathscr{PF}$ guarantees $\mathscr{AF}(A)$ for a set of actions $A \subseteq Act$, if A is competition safe.*

PROOF. Similar to Lemma 5, where competition was prevented once and for all, except that competition is permitted here, but is guaranteed to stop eventually. □

Omission. Consider a collection of actions $A \subseteq Act$ with a common process $p \in Proc$, and let B denote its complement in Act_p, $A \cup B = Act_p$, $A \cap B = \emptyset$. The set of actions A is *omission safe* by process p in action system S, if there is an invariant I of S such that the implication

$$\Box\Diamond g_A \Rightarrow \Diamond\Box\neg g_B$$

holds in the reduction $S \backslash A$ with initial condition I, when $\mathscr{AF}(B)$ is assumed. More generally, a collection of actions $A \subseteq Act$ is defined to be omission safe if it is a union $A = \cup A_i$ where each A_i is omission safe by some process p.

We now have the following result.

LEMMA 9. *$\mathscr{PF}$ guarantees $\mathscr{HF}(A)$ for a set of actions $A \subseteq Act$, if A is omission safe.*

If A is competition safe, then $A \cap Act_p$ is omission safe for all processes p involved in A. This shows that competition safeness implies omission safeness also in the dynamic case.

Conspiracy. Extending the static conditions for conspiracy safeness to the dynamic case means, in fact, that the existentially quantified state y in the definitions of the enabling, disabling, and keeping predicates is replaced by the current state:

$$\begin{aligned} Enable_a(y) &= \{b \in Enable_a \mid g_b(y) \wedge \neg g_a(y) \wedge g_a(f_b(y))\}, \\ Disable_a(y) &= \{b \in Disable_a \mid g_b(y) \wedge g_a(y) \wedge \neg g_a(f_b(y))\}, \\ Keep_a(y) &= \{b \in Keep_a \mid g_b(y) \wedge g_a(y) \wedge g_a(f_b(y))\}. \end{aligned}$$

In order for a cyclic cover $\xi = (b_1, \ldots, b_k, b_1)$ to cause conspiracy against a, the condition $b_i \in Keep_a(y)$ must then hold for the current state y when b_i is about to participate in the conspiracy. Let this condition, for an arbitrary conspiratorial cycle ξ against a, be denoted by $\varphi_{\xi,i}$,

$$\varphi_{\xi,i}(y): \quad g_{b_i}(y) \wedge g_a(y) \wedge g_a(f_{b_i}(y)).$$

Correspondingly, for a conspiratorial chain $\xi = (b_0, \ldots, b_k)$ against a, the counterpart of the static conditions are

$$\begin{aligned} &\varphi_{\xi,0}(y): g_{b_0}(y) \wedge \neg g_a(y) \wedge g_a(f_{b_0}(y)), \\ &\varphi_{\xi,i}(y): g_{b_i}(y) \wedge g_a(y) \wedge g_a(f_{b_i}(y)), \quad 1 < i < k, \\ &\varphi_{\xi,k}(y): g_{b_k}(y) \wedge g_a(y) \wedge \neg g_a(f_{b_k}(y)). \end{aligned}$$

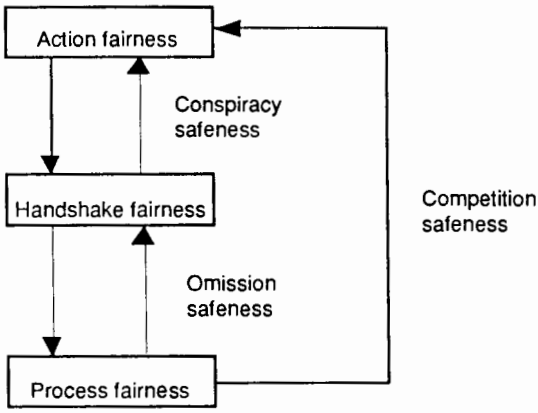

Fig. 5. Relationship between fairness notions.

Let $A' \subseteq A$ be a subset for which

$$\Box\Diamond g_A \Rightarrow (\Box\Diamond g_{A'})$$

holds in the reduction $S\backslash A$ with initial condition I, for some invariant I of S. To prevent conspiracy against the actions A it is then obviously sufficient to prevent conspiracy against the subset A'. Such a subset A' is called a *core* of A.

A set of actions $A \subseteq Act$ is now defined to be *conspiracy safe* in action system S if there is a core A' of A such that for each conspiratorial cycle or chain ξ against $a \in A'$ with actions $b_i \notin A$ there is at least one condition $\varphi_{\xi,i}$ which will eventually become and stay false, if a is infinitely often enabled. This means that for a conspiratorial cycle the condition

$$\Box g_a \Rightarrow \Diamond\Box\neg\varphi_{\xi,i}$$

must hold in the reduction $S\backslash A$ with initial condition I, when $\mathscr{AF}(b)$ is assumed for each action $b \in \xi$. For each conspiratorial chain the condition

$$\Box\Diamond g_a \Rightarrow \Diamond\Box\neg\varphi_{\xi,i}$$

must hold in the reduction $S\backslash A$ with initial condition I, when $\mathscr{AF}(b)$ is assumed for each action $b \in \xi$. As before, I is here some invariant of S. We now have the following result.

LEMMA 10. *$\mathscr{HF}(A)$ for a set of actions $A \subseteq Act$ guarantees $\mathscr{AF}(A)$, provided that A is conspiracy safe.*

Competition safeness of A implies that both $Keep_a(y) - A$ and $Disable_a(y) - A$ will eventually stay empty for all $a \in A$, if g_a holds infinitely often without any action in A being executed. Therefore competition safeness implies conspiracy safeness also in the dynamic case.

We summarize the results of Sections 4.2 and 4.3 in the following theorem.

THEOREM 4. *Let S be an action system. Then*

(*i*) *$\mathscr{PF}$ guarantees $\mathscr{AF}(A)$ for a set of actions $A \subseteq Act$ in S, if A is competition safe,*

(*ii*) $\mathscr{PF}$ *guarantees* $\mathscr{HF}(A)$ *for a set of actions* $A \subseteq Act$ *in* S, *if* A *is omission safe,*

(*iii*) $\mathscr{HF}(A)$ *for a set of actions* $A \subseteq Act$ *guarantees* $\mathscr{AF}(A)$, *provided that* A *is conspiracy safe.*

Although the conditions in this theorem are sufficient but not necessary, they seem to provide reasonably powerful criteria for proofs of fair serializability. The relationship between the fairness notions studied in this section are shown in Figure 5.

5. PROVING PROPERTIES OF CONCURRENT EXECUTIONS

Let us now consider how to prove temporal properties of concurrent executions of action systems. Our aim is to be able to prove that an actual distributed execution of an action system has some desired property. Proving this would basically require that we argue about the properties in terms of an implementation of the action system, using for instance the implementation on local area networks given in Section 3. This, however, would be very tedious, and we prefer to prove properties with respect to the execution models that we have given for action systems. In particular, we would like to use the sequential model, because this is simpler and easier to reason about. Below we will tie together the results of the preceding section and show how to prove properties that hold for the concurrent model, by proving corresponding properties for the sequential execution model. We will also discuss more carefully how properties that hold for the concurrent execution model can be interpreted as properties of an actual distributed execution of an action system.

5.1 Proving Liveness Properties

Assume that we have an implementation of an action system S that guarantees process fairness for all processes, and handshake fairness for some actions A in Act. We want to prove that some temporal property R is satisfied by each concurrent execution of the execution system that is $\mathscr{PF}$ and $\mathscr{HF}(a)$, $a \in A$. Assume that the property R is insensitive to stuttering. We may now proceed as follows.

(i) We first prove that the property R holds for each sequential execution of S, assuming $\mathscr{AF}(A')$ for some sets of actions A' in Act.

(ii) We then prove that each $\mathscr{AF}(A')$ condition holds for concurrent executions of S, assuming $\mathscr{PF}$ and $\mathscr{HF}(a)$, $a \in A$.

Let us consider both these steps in more detail.

From step (i) it follows by Theorem 2 that property R will also hold for each concurrent execution of S for which the conditions $\mathscr{AF}(A')$ hold. Hence if we can prove step (ii), then R will obviously hold for each concurrent execution of S that is $\mathscr{PF}$ and $\mathscr{HF}(A)$, $a \in A$.

Consider now step (ii). Action fairness can be established using the results of the previous section. Initially, we start with some collection of action fairness obligations $\mathscr{AF}(A_1), \ldots, \mathscr{AF}(A_m)$, that we need to establish. By the results of the preceding section we may establish these fairness properties by proving that

the action system has certain safeness properties:

(1) A fairness obligation $\mathscr{AF}(A')$ is established directly, if we can prove competition safeness for this set of actions.
(2) We can reduce a fairness obligation $\mathscr{AF}(A')$ to a fairness obligation $\mathscr{KF}(A')$, if we can prove conspiracy safeness for this set of actions.
(3) A fairness obligation $\mathscr{KF}(A')$ can be established directly, by proving omission safeness for this set of actions.

The second step is established, if by using these rules we can reduce the fairness obligations to those handshake fairness assumptions that we may assume for the implementation, i.e., $\mathscr{KF}(a)$, for each $a \in A$. These rules may be applied directly to the sets A_i, or we might partition these sets into smaller sets, and prove fairness assumptions separately for each of these smaller sets. (This latter approach is further elaborated in [8].)

Static competition, omission, and conspiracy safeness are expressed as properties in ordinary first-order calculus. Dynamic competition, omission and conspiracy safeness are expressed as temporal logic properties of the concurrent execution model. In the latter case we have to show that some temporal property Q holds for each concurrent execution of the reduced system $S\backslash A_i'$, assuming $\mathscr{AF}(C)$ for the reduction, for some sets of actions A_i' and C in Act. The property Q is in all these cases insensitive to stuttering and makes no reference to variables *free* in the concurrent model. Hence, Q will hold in each concurrent execution of $S\backslash A_i'$ assuming $\mathscr{AF}(C)$, if and only if it holds in each sequential execution of $S\backslash A_i'$ assuming $\mathscr{AF}(C)$. Thus, to establish the required temporal properties, we need only to consider sequential executions of action systems.

If the property Q cannot be established for the reduced system $S\backslash A_i'$, even if $\mathscr{AF}(C)$ is assumed, then we may repeat the above proof procedure, but now applied to the system $S\backslash A_i'$ with $\mathscr{AF}(C)$ and $\mathscr{KF}(a)$, $a \in A$. The sequence of reductions must eventually terminate, as at least one action is removed from the system in each reduction.

In conclusion, by using the safeness notions and the notion of action fairness, we are able to establish that some desired property holds for any concurrent execution of an action system S by only reasoning about temporal properties of sequential executions of S. We have thus achieved the goal that we set ourselves in the introduction.

Example 5. We apply the above proof method to the dining philosophers program of Example 3. We want to show that individual starvation cannot occur in a concurrent execution of the example program, if each execution is process fair. Thus we want to show that

$$\Box(hungry_i \Rightarrow \Diamond \neg hungry_i) \tag{1}$$

for each $i = 1, \ldots, n$.

By the proof method above, we divide the proof into two steps:

(i) We first prove (in the sequential execution model) that (1) holds, assuming action fairness for all eating actions.

(ii) We then show that action fairness holds for all eating actions, assuming only process fairness.

Part (i) of the proof has already been established in Section 2.2. Hence it only remains to show part (ii), that the action system is fair for all eating actions.

Omission safeness. We show first that *Eating*[i] is omission safe. This will be guaranteed by the process *Philosopher*[i]. The two sets of actions in the definition of omission safeness are in this case

$$A = \{Eating[i]\},$$
$$B = \{Reserve[i], Thinking[i]\}.$$

We have to show that $I \wedge \Box\Diamond g_A \Rightarrow \Diamond\Box\neg g_B$ holds in the reduced system, with *Eating*[i] removed, assuming action fairness for {*Reserve*[i], *Thinking*[i]}. This amounts to showing that, if $hungry_i$ holds infinitely often in the reduced system, then eventually $reserved_i$ will become true. This is obviously the case when action *Reserve*[i] is treated fairly.

Conspiracy safeness. Action fairness is now guaranteed if we can show that {*Eating*[i]} is conspiracy safe. Figure 6 shows all possible covers of the action *Eating*[i]. In the figure, there is an arrow from action b to b', if $b, b' \in Exclude_a$ and $Proc_a \cap Proc_b \not\subseteq Proc_{b'}$. We indicate whether an action can enable (E) or disable (D) action *Eating*[i]; all actions except *Thinking*[i] can keep it enabled. Analyzing these covers we see that any conspiratorial cycle must contain either *Reserve*[i] or *Eating*[$i - 1$], and any conspiratorial chain must contain *Reserve*[$i + 1$] and either *Thinking*[i] or *Eating*[$i + 1$].

Let us first consider the conspiratorial cycles. We will prove that for any conspiratorial cycle $(b_1, b_2, \ldots, b_k, b_1)$,

$$(I \wedge \Box g_{Eating[i]}) \Rightarrow \Diamond\Box\neg\varphi_{\xi,j}$$

holds for some j in the reduction $S\backslash\{Eating[i]\}$ of S, with $\mathscr{AF}(b_j)$ assumed for $j = 1, \ldots, k$. It is obviously sufficient to replace $\varphi_{\xi,j}$ here by the guard of b_j. After what was shown above about *Reserve*[i], it now suffices to prove that the guard of *Eating*[$i - 1$] will eventually turn false in the reduced system without action *Eating*[i]. This can be done by showing that $hungry_i$ eventually implies both $token_i$ and $reserved_i$ in this reduced system.

We use well-founded ranking to establish this. We select a mapping $f(y) = (m, b)$ of system states y into a set W,

$$W = \{0, \ldots, n - 1\} \times \{true, false\},$$

such that m is the smallest integer for which $token_{i-m} = true$ and b is the value of $reserved_{i-m}$. The existence of m is ensured by the invariant proved in Section 2.2, which states that there is always exactly one fork with an associated token. The well-founded ordering in W is defined as follows: $(m, b) > (m', b')$, if $m > m'$, or if $m = m'$, $b = false$ and $b' = true$. The least element in W is $(0, true)$, and the required situation has been achieved when $f(y)$ has this value.

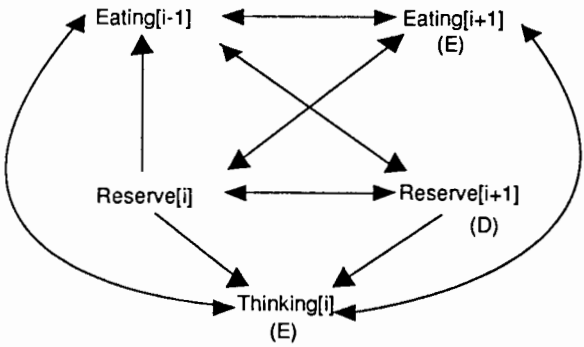

Fig. 6. Covers for action *Eating* [*i*].

Let y be an arbitrary state satisfying the invariant, and let $f(y) = (m, b) > (0, true)$. Consider the following three different cases:

(i) $m > 0$, and $b = false$: the value of $f(y)$ is decremented by the helpful actions $\{Eating[i - m], Reserve[i - m]\}$,

(ii) $m > 0$, and $b = true$: the helpful action is $\{Eating[i - m]\}$,

(iii) $m = 0$, and $b = false$: the helpful action is $\{Reserve[i]\}$.

Thus the situation where $m = 0$, and $b = true$ will eventually be reached, provided that we have action fairness for all sets of helpful actions in the reduced system. Hence, we need to apply our fairness proof methods once again, but now for the reduced system in which action *Eating*[*i*] has been removed. Thus, for case (i), we need to prove that the reduced action system, with initial condition $m > 0$, and $b = false$, is action fair for $\{Eating[i - m], Reserve[i - m]\}$, and similarly for the other cases.

In order to check this we first notice that each set of helpful actions is omission safe. Conspiracy safeness is straightforward in cases (ii) and (iii). In case (i) this can be shown by selecting $\{Reserve[i - m]\}$ as the core of A.

Let us then consider the conspiratorial chains. The only actions that can enable *Eating*[*i*] are *Eating*[*i* + 1] and *Thinking*[*i*], and the only action that can disable it is *Reserve*[*i* + 1]. Hence a conspiratorial chain must either contain *Thinking*[*i*] and *Reserve*[*i* + 1] or it must contain *Eating*[*i* + 1] and *Reserve*[*i* + 1]. We have to show that for each conspiratorial chain $(b_0, b_1, \ldots, b_k)$,

$$(I \wedge \Box\Diamond g_{Eating[i]}) \Rightarrow \Diamond\Box\neg\varphi_{\xi,j}$$

for some j in the reduction $S\backslash\{Eating[i]\}$ of S, with $\mathscr{AF}(b_j)$ assumed for $j = 0, 1, \ldots, k$.

Consider first a chain with actions *Thinking*[*i*] and *Reserve*[*i* + 1]. Action *Thinking*[*i*] enables *Eating*[*i*] only if $\neg hungry_i \wedge \neg(reserved_i \wedge token_i)$ holds. If the enabling condition for *Eating*[*i*] holds infinitely often in the reduced system without *Eating*[*i*], then $hungry_i$ holds infinitely often. This condition can only be turned off by the action *Eating*[*i*], once it starts to hold in the reduced system, so eventually it will hold forever. Consequently, the action *Thinking*[*i*] cannot

repeatedly enable the condition for action *Eating*[i] in the reduced system. Hence, no chain with action *Thinking*[i] can achieve conspiracy against *Eating*[i].

Consider now a chain with actions *Eating*[$i + 1$] and *Reserve*[$i + 1$]. The action *Reserve*[$i + 1$] disables action *Eating*[i] only if $hungry_i \wedge hungry_{i+1} \wedge \neg reserved_{i+1} \wedge token_{i+1}$. Assume that the enabling condition for action *Eating*[i] holds in the reduced system, i.e., $hungry_i \wedge (\neg reserved_{i+1} \vee \neg token_{i+1})$. If $token_{i+1} = false$, then this will hold forever, as it can only be set to *true* by the removed action *Eating*[i]. If $token_{i+1} = true$, then we have two cases. Either the condition $hungry_{i+1}$ does not hold infinitely often, in which case the condition $\neg hungry_{i+1}$ will eventually hold forever, or $hungry_{i+1}$ does hold infinitely often. In the latter case action *Eating*[$i + 1$] will be infinitely often enabled, so by the fairness assumption it will eventually be executed. Once executed, it will set $token_{i+1} = false$, which thereafter will stay false forever. In all these cases one of the conditions required for the action *Reserve*[$i + 1$] to disable action *Eating*[i] eventually becomes false forever. Hence no chain with actions *Eating*[$i + 1$] and *Res*[$i + 1$] can achieve conspiracy against *Eating*[i].

This concludes our proof that the action system of Example 3 is action fair for each eating action, and hence, by combining this result with the result of Section 2.2, that the action system avoids starvation of the philosophers.

5.2 Interpretation of Correctness Proofs

The effect of an action $a \in Act$, i.e., the updating of the variables of the participating processes, is associated with the handshake transition hs_a in the concurrent model. All local variables are considered to be immediately updated when the decision to execute action a is made. The fact that updating the local variables in a process p takes time is modeled by the interval from handshake hs_a to the releases rel_p, $p \in Proc_a$. Since this does not directly correspond to the behavior of a real implementation, the validity of the model requires some justification.

To make things more concrete, let us consider an action system with two actions, $Act = \{a, b\}$, and three processes, $Proc = \{p_1, p_2, p_3\}$. Each process p_i has one local Boolean variable y_i. The actions are defined by

action a **by** p_1, p_2:
$y_1 = y_2 \rightarrow y_1, y_2 := \neg y_1, \neg y_2$
action b **by** p_2, p_3:
$y_2 = y_3 \rightarrow y_3 := \neg y_3$.

A possible sequence of events in a real implementation is given in Table I. The question marks indicate situations where the value of a variable is not known for certain, because an update is in progress.

Handshakes and releases in Table I correspond roughly to the respective transitions in the concurrent execution model shown in Table II. In the concurrent model the local variables y_i are immediately updated by the handshake transitions. In an actual implementation the updating is done sometime during the execution of the action body, and need not happen at the same time in different processes. As a result, properties like $(y_1 = y_3)$ could have different values in the concurrent model and in reality. In the model this expression is

Table I. Snapshots of the Real Operation of An Action System

	y_1	y_2	y_3
initially:	true	true	false
handshake a:	?	?	false
release p_2 from a:	?	false	false
handshake b:	?	?	?
release p_3 from b:	?	?	true
release p_2 from b:	?	false	true
release p_1 from a:	false	false	true

Table II. Global States in Concurrent Execution Model of an Action System

	y_1	y_2	y_3
initially:	true	true	false
handshake a:	false	false	false
release p_2 from a:	false	false	false
handshake b:	false	false	true
release p_3 from b:	false	false	true
release p_2 from b:	false	false	true
release p_1 from a:	false	false	true

turned true by hs_a and false by hs_b; in a real execution it might as well be first turned true by p_3 and then false by p_1. Because of simultaneity it is not even clear whether any such changes could be seen in a real execution.

This presents us with a problem of how to interpret properties that we have proved to hold for a concurrent execution of an action system. Consider the property $\Box(y_1 = y_2)$, which can be proved to hold for this action system. In a real execution, when action a is executed, either p_1 or p_2 would update its local variable before the other, thus temporarily destroying this invariant. Similarly a liveness property like $\Diamond(y_1 = y_2 = y_3)$, which can be proved to hold in the action system, might never actually hold in a real execution of the action system.

What should be the truth of an assertion about the global state, when some program variables are in the middle of being updated? It is not satisfactory to say that the assertion is only defined when all variables in it have a well-defined value. In a real execution of an action system there need not be a single moment of time (except for the initial and final states) when all program variables have well-defined values. This is a consequence of the parallelism in execution. Still, we would like to say that any execution of the action system in some real and observable way does have the safety or liveness property that we have proved for the concurrent execution model.

In our description of the behavior of a real system above we have assumed a universal time that determines a complete ordering of events. There need not, however, exist any means to determine this ordering by observing the system. For observations one could assume a relativistic notion of time where no other timing constraints are present except those determined by the handshakes [30]. Any sequence of observations that is not in conflict with this partial ordering is then possible. A model of execution, such as our concurrent model, would be

considered valid if any sequence of global states generated by the model is consistent with these constraints.

The concept of a global state is, however, obscured if the relativistic notion of time is adopted in the way suggested above. Although the sequence of global states in the model is not in conflict with what might be observed of the real system, it need not agree with any particular observations either. It can be questioned whether reasoning with such a global state has any relevance. In other words, if we prove that some safety property holds throughout the execution in the concurrent model, this would only say to us that even if this property is not true of our present observation of the system, there is some other way of observing the system, in which the property does hold. To make some inference about the actual behavior of the system it seems reasonable to require that the sequence of states in the model will also be observed in the real execution of the action system.

We will therefore propose another approach to this problem: We specify the way in which one should observe the system, so that one sees a sequence of global states that agrees with the sequence generated by the concurrent execution model. A possible arrangement for such observations is as follows. A request is broadcast simultaneously to all processes, asking them to give their local states. If not engaged in a transaction, a process replies immediately to the request; otherwise it delays the reply until the current action has been completed. The replies are collected to form a global state, which will correspond to a global state used in our model. The complete sequence of global states in the concurrent model is obtained, if this sampling of the global state is done frequently enough, and successive exact duplications of the global states are removed. Consequently, any property that holds in the concurrent execution model of an action system will also hold for the sequence of global states observed in this way of an actual execution of an action system.

It is worth noticing that the particular view of time we selected for the model has no effect on the fairness notions in the concurrent model, since no handshake transition can be enabled while any of the processes in question is involved in another action.

6. RELATED WORK

The generalized handshake mechanism and our approach to describing system behavior in terms of joint actions was originally inspired by the Petri net approach to system modeling. If action guards are ignored, then the generalized handshake corresponds directly to the firing rule for transitions in Petri nets. An action system can be understood as a special kind of Petri net where each process is represented by a token, considered to carry the local variables of the process with it. The addition of guards to Petri nets has been proposed in [14]. Also, the action system approach is quite close to the shared variable transition model for describing the behavior concurrent systems, as presented in [31]. They are, in fact, identical for the sequential execution model. The main thing that distinguishes our approach from these is the requirement that the action guards be separable, and that these transition systems are to be executed in parallel in a distributed fashion.

The work reported here generalizes previous work in [5]. There the notions of centralized and decentralized action systems were defined, but only the serial execution model was given. What we here call an action system corresponds to a decentralized action system in that paper. An implementation in CSP was given for the special case when each action involves at most two processes. Although the fairness questions of the implementation were ignored, this CSP implementation can easily be shown to be $\mathscr{PF}$ or $\mathscr{HF}$, depending on whether the implementation of CSP supports process or channnel fairness [29]. The implementation given here is an improvement over the CSP implementation, as it is given directly for a local area network, thus avoiding the problem of finding an efficient implementation of CSP with output guards.

The idea of synchronizing more than two processes by common handshakes has also been proposed by Francez and Hailpern in their work on *scripts* [22]. These are primarily seen as higher level procedure-like mechanisms for organizing the cooperation of processes in a distributed system. It is also possible to achieve the effect of joint actions with the script mechanism. The mechanism is not, however, seen as a primitive communication mechanism, but is rather thought to be built on existing communication mechanisms, such as CSP or Ada handshaking. Proposals for synchronizing multiple processes that are somewhat similar to ours have also been proposed by Ramesh [34] and Forman [20]. A somewhat different approach to multiparty interaction is taken in [12].

Providing a distributed implementation of our mechanism is a generalization of the problem of giving a distributed implementation of CSP with output guards, which has been treated quite extensively in the literature (see [2, 10] for a reference to work in this area). None of the implementations that we are aware of is based on using a shared broadcast channel. The implementation by Schneider [35] seems to be closest to ours, as it also uses broadcasting, but the channel is not shared, and the necessary ordering of the messages is imposed by timestamps.

The notions of process fairness and handshake fairness are, in fact, the same as the process fairness and channel fairness that were introduced in [29] in the context of CSP. The notion of action fairness, although quite natural in the case of a shared variable interpretation of action systems, does not seem to have been considered before in the context of distributed systems. It does not come up so naturally in connection with conventional approaches, as they lack higher-level notions for structuring process cooperation.

The validity of the sequential model with action fairness led us to the problem of fair serializability. Comparing this with the analogous notion in database transactions we note that fair serializability has no effect on the result of the transactions, only on the fairness notion that can be supported. This notion was first proposed and studied in [7]. It was subsequently applied by Grumberg, Francez, and Katz [23] to study the fair termination problem of CSP programs. An alternative presentation of this basic idea is given in [21, Section 5.2]. Another recent study on this topic is [3].

An important advantage of the action system approach is that it makes it easy to construct distributed programs by stepwise refinement. The action system approach was originally developed with this purpose in mind. The stepwise

refinement method for action systems was originally described in [5]. A later application of this method to the problem of removing virtual channels in a distributed program is described in [36]. The action system approach is also very useful for a Dijkstra-style way of developing programs by incrementally adding invariants for a distributed system, and then giving actions that preserve these invariants, while simultaneously achieving some liveness conditions (see, e.g., [16, 17]). A case study of this approach, in the context of action systems, is described in [6].

An approach similar to action systems has recently been proposed by Chandy and Misra [11]. They also argue for the necessity to describe the overall behavior of a distributed system, rather than looking at the behavior of processes in the system individually. The possible behaviors in their systems are described by guarded iteration statements, which, as we have shown above, are equivalent to the sequential execution model of action systems. They use this approach to program a distributed algorithm in a stepwise manner, similar to the way in which the distributed sorting algorithm is developed in [6]. We think that this work provides additional support for the fruitfulness of the action system approach to the programming of the algorithmic behavior of a distributed system.

7. CONCLUSIONS

The main purpose of this paper has been to present and analyze the action system approach to constructing distributed systems. We have tried to show that this approach gives a good basis for describing the overall behavior of a distributed system, and that it fits nicely into the temporal logic framework for verifying safety and liveness properties. Our analysis of action systems was divided into three parts. First, we presented a simple sequential execution model for action systems, on which the temporal reasoning about action systems is based. The parallel execution is here modeled by sequential, nondeterministic execution, in a way similar to the sequential execution of guarded iteration statements. The notion of action fairness was introduced to allow proofs of liveness properties.

Next, we showed how to implement action systems in a distributed fashion. For this purpose we defined another execution model, the concurrent execution model, which corresponds to the way in which action systems are executed in the implementation. The notions of handshake and process fairness were introduced in this model. We showed that action systems can be implemented efficiently on a broadcasting network, and that the required notions of process and action fairness can be guaranteed by this implementation.

In the last part we studied the relationship between the two models of execution. We showed that except for fairness, the two models are equivalent in the sense that any temporal property that can be proved to hold for each sequential execution of an action system will also hold for each concurrent execution of the action system. The handshake and process fairness notions guaranteed by the implementation are, however, weaker than the action fairness notion that we need in proving properties in the sequential execution model. We identified a class of action systems that we called fairly serializable, for which process or handshake fairness does imply action fairness. We have given proof rules by which fair serializability of action systems can be shown. By using these

proof rules one can prove both safety and fairness properties for action systems within the simpler sequential execution model.

Let us finally wind up with some problems that we find interesting and worth further study. One problem that the action system approach shares with other endogenous models of program behavior is the modularization of large systems. The strength of the process-based approaches, especially those based on communicating processes, is that they support a very nice, object-oriented way of constructing a large program from smaller parts. The problem of finding a good modularization mechanism for action systems is very similar to the corresponding problems for Prolog and production system languages such as OPS5, as well as for the modular construction of Petri nets.

The design of higher level control structures for action systems is also a topic worth further study. Sequencing is a good example of the need for such constructs: enforcing some actions to be executed in sequence requires unattractive manipulation of Boolean flags, which makes the program harder to understand and prove correct.

Distributed implementation of action systems on point-to-point networks forms another class of interesting problems. This application area is quite important, especially as the action system approach seems to be well suited to construct multiprocessor algorithms on loosely coupled MIMD machines. Some work in this area has been done in [18] and more recently in [9, 33]. Another important topic has to do with efficient distributed execution of action bodies. The body should be partitioned among the processes in such a way that redundant computations are avoided.

Finally the problems involved in relating the sequential execution model to the true concurrency model seem important and well worth studying in more detail by using a more abstract model for true concurrency. Our results show that the simple interleaving model of communication, in which communication events are modeled by atomic actions that do not take any time, is insufficient as a way of modeling the real behavior of distributed systems. Especially when considering fairness properties, it is necessary to take into account that the communication events, even in the case of synchronous handshakes, do take time.

The methods for proving fair serializability that we described in Section 4 seem to provide a reasonably good collection of tools to tackle realistic problems. However, they do not necessarily provide a complete set of tools, and the proof method involved in using them can be quite cumbersome. A promising topic for research would be to extend these methods and build a special-purpose proof system by which the method of fair serializability could be used to establish properties of real distributed executions of action systems in a more rigorous fashion.

ACKNOWLEDGMENTS

We would like to thank the referees for their insightful comments on the paper. Discussions with Mats Aspnäs, Luc Bouge, Nissim Francez, Eeva Hartikainen, Leslie Lamport, and Fred Schneider have also been very helpful.

REFERENCES

1. *Ada Programming Language.* ANSI/MIL-STD-1815A-1983.
2. ANDREWS, G. R., AND SCHNEIDER, F. B. Concepts and notations for concurrent programming. *ACM Comput. Surv. 15*, 1 (March 1983), 3–43.
3. APT, K., FRANCEZ, N., AND KATZ, S. Appraising fairness in languages for distributed programming. In *14th ACM Conference on Principles of Programming Languages* (Munich, Jan. 1987), ACM, New York, 1987, 189–198.
4. BACK, R. J. R., HARTIKAINEN, E., AND KURKI-SUONIO, R. Multi-process handshaking on broadcasting networks. In *Reports in Computer Science 42*, Abo Akademi, Abo, Finland, 1985.
5. BACK, R. J. R., AND KURKI-SUONIO, R. Decentralization of process nets with a centralized control. In *Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 131–142.
6. BACK, R. J. R., AND KURKI-SUONIO, R. A case study in constructing distributed algorithms: Distributed exchange sort. In *Proceedings of Winter School on Theoretical Computer Science* (Lammi, Finland, Jan. 1984). Finnish Society of Information Processing Science, 1–33.
7. BACK, R. J. R., AND KURKI-SUONIO, R. Co-operation in distributed systems using symmetric multi-process handshaking. In *Reports in Computer Science 34*, Abo Akademi, Abo, Finland, 1984.
8. BACK, R. J. R., AND KURKI-SUONIO, R. Serializability in distributed systems with handshaking. In *Automata, Languages and Programming. Lecture Notes in Computer Science 317.* T. Lepistö amd A. Salomaa, Eds. Springer Verlag, Berlin, 1988, 52–66.
9. BAGRODIA, R. On the design of high performance distributed systems. Ph.D. dissertation, Univ. of Texas, Austin, 1987.
10. BUCKLEY, G. N., AND SILBERSCHATZ, A. An effective implementation for the generalized input-output construct of CSP. *ACM Trans. Program. Lang. Syst. 5*, 2 (April 1983), 223–235.
11. CHANDY, M., AND MISRA, J. An example of stepwise refinement of distributed programs: Quiscence detection. *ACM Trans. Program. Lang. Syst. 8*, 3 (July 1986), 326–343.
12. CHARLESWORTH, A. The multiway rendezvous. *ACM Trans. Program. Lang. Syst. 9*, 2 (July 1987), 350–366.
13. *CSMA/CD Access Method and Physical Layer Specifications.* IEEE Standard 802.3, IEEE, New York, July 1983.
14. DE CINDIO, F., DE MICHELIS, G., POMELLO, L., AND SIMONE, C. Superposed automata nets. In *Application and Theory of Petri Nets. Informatik-Fachberichte 52*, C. Girault and W. Reisig, Eds., Springer-Verlag, Berlin, 1982.
15. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
16. DIJKSTRA, E. W. Invariance and nondeterminacy. In *Mathematical Logic and Programming Languages.* C. A. R. Hoare and J. C. Shepherdson, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1985, 157–165.
17. DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., AND SCHOLTEN, C. S. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–975.
18. EKLUND, P. Synchronizing multiple processes in common handshakes. In *Reports in Computer Science 39*, Abo Akademi, Abo, Finland, 1985.
19. FORGY, C., AND DERMOT, M. C. OPS, a domain independent production system language. In *Proceedings of Fifth International Joint Conference on Artificial Intelligence* (Cambridge, Mass., Aug. 1977), Morgan Kaufmann, 1977, 933–939.
20. FORMAN, I. R. Raddle, an informal introduction. Tech. Rep. STP-182-85, Microelectronics and Computer Technology Corp., Austin, Tex., 1986.
21. FRANCEZ, N. *Fairness.* Springer-Verlag, Berlin, 1986.
22. FRANCEZ, N., AND HAILPERN, B. Script: A communication abstraction mechanism. In *Second ACM-SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 213–227.
23. GRUMBERG, O., FRANCEZ, N., AND KATZ, S. Fair termination of communicating processes. In *Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Vancouver, Aug. 1984). ACM, New York, 1984, 254–265.
24. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

25. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666-677. Reprinted in *Commun. ACM 26*, 1 (Jan. 1983), 100–106.
26. HOARE, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1985.
27. INMOS LTD. *Occam Programming Manual.* Prentice-Hall, Englewood Cliffs, N.J., 1985.
28. KROEGER, F. *Temporal Logic of Programs. EATCS Monographs on Theoretical Computer Science, vol. 8.* Springer-Verlag, Berlin, 1986.
29. KUIPER, R., AND DE ROEVER, W. P. Fairness assumptions for CSP in a temporal logic framework. In *Formal Description of Programming Concepts—II*, D. Bjørner, Ed. North-Holland, Amsterdam, 1983, 159–167.
30. LAMPORT, L. Time, clocks, and ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
31. MANNA, Z., AND PNUELI, A. How to cook a temporal proof system for your pet language. In *Tenth ACM Conference on Principles of Programming Languages* (Austin, Tex., Jan. 1983). ACM, New York, 1983, 141–154.
32. PNUELI, A. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency. Lecture Notes in Computer Science 224*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., Springer-Verlag, Berlin, 1986, 510–584.
33. RAMESH, S. A new and efficient implementation of multiprocess synchronization. In *PARLE Parallel Architectures and Languages Europe. Lecture Notes in Computer Science 259*, Springer-Verlag, Berlin, 1987, 387–401.
34. RAMESH, S., AND MEHNDIRATTA, S. L. A new class of high-level programs for distributed computing systems. In *Proceedings of Fifth Conference on FST-TCS. Lecture Notes in Computer Science 206*, Springer-Verlag, Berlin, 1985, 42–72.
35. SCHNEIDER, F. R. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst. 4*, 2 (April 1982), 125–148.
36. SERE, K. Stepwise removal of virtual channels in distributed algorithms. In *Second International Workshop on Distributed Algorithms* (Amsterdam, 1987).
37. TANENBAUM, A. S. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

Received August 1987; revised May 1988; accepted June 1988