Theoretical Computer Science 49 (1987) 145-169
North-Holland

REPEATED SNAPSHOTS IN DISTRIBUTED SYSTEMS WITH SYNCHRONOUS COMMUNICATIONS AND THEIR IMPLEMENTATION IN CSP

Luc BOUGÉ

LITP, Université Paris 7, 75251 Paris Cédex 05, France, and Laboratoire d'Informatique, Faculté des Sciences, Université d'Orléans, 45067 Orléans Cédex 02, France

Abstract. A new version of the Snapshot Algorithm of Chandy and Lamport (1985) is presented. It considers synchronous communications and partially ordered semantics and allows for repeated snapshots. Its implementation in the language CSP is described: it is symmetric, generic and bounded in storage. It yields a symmetric and generic solution to the 'Distributed Termination Detection' problem of Francez.

0. Introduction

Several recent works have drawn attention to the notion of *knowledge* in distributed computing [12, 18, 21]. It has been demonstrated that not only knowledge of facts, but knowledge about knowledge as well, plays a central rôle. A distributed system can usually be seen as a set of autonomous sequential machines interconnected by some communication network. The only way machines can exchange information is by means of messages. Messages, contrarily to shared variables, have the property that they may be exchanged only if the sender agrees to send and the receiver eventually accepts it. A machine thus has only partial knowledge about the state of other machines, and also about their knowledge. Knowledge is essentially *local.*

It has been advocated for a long time that *time* should be considered as local, too. In fact, Petri nets theory stems essentially from the absence of global time. Similar remarks have been made by Lamport [19]. A machine may only acquire relative knowledge about time of other machines by means of message synchronization: 'before', 'after' and, in the case of synchronous systems, 'simultaneously'. This situation has often been compared to quantum mechanics (e.g., [22]). There, one faces an irremovable trade-off between time and space (state). Knowledge is inherently *unprecise* and *partial.*

The following examples illustrate some specific problems raised by partial knowledge in distributed computing.

0.1. The Distributed Termination Problem

A bank employs a large number of clerks managing clients accounts. Each clerk is in charge of a certain number of accounts. Orders are received at the beginning of each day, and processed during the same day. They may increase or decrease deposits or transfer funds from one account to another. Processing such a transfer order necessitates in general the cooperation of two clerks.

Clerks work in different offices and communicate via telephone. Suppose clerk A is in charge of account X and clerk B of account Y. To transfer funds from X to Y, clerk A dials up clerk B. Once they agree on the operation, A decreases account X and B increases account Y with the given amount. The rule is that a clerk may return home as soon as he has finished processing the required orders. On the other hand, insubordinate clerks are immediately fired.

A perhaps surprising consequence is that under this rule no clerk may ever return home! Consider, for example, clerk B above. Of course, he would not like to get fired. Suppose B has finished processing orders. He cannot know whether A has finished as well. Suppose B leaves now and, later on, A needs to transfer money from X to Y. Then A will not be able to achieve transfer and will thus get disturbed by B. Clerk B prefers therefore to stay and wait, possibly forever, for some phone call from A. Eventually, late in the night, all clerks will be sitting and waiting for a phone call that will never occur...

Similar situations are far from being unusual in distributed computing. Francez [13] identifies this phenomenon as the Distributed Termination Problem. It arises each time a machine B has to take some irrevocable decision, such as to terminate, on the basis of unprecise information. Taking this decision may preclude other machines from computing if they need to cooperate with B.

Usually no machine will thus ever take such a decision. The whole system will then deadlock in a situation where all machines have completed their task but none of them can take the decision to terminate.

0.2. The Snapshot Problem

The Distributed Termination Problem described above could be easily solved if some clerk could take a global snapshot of the system. Then he could detect that all clerks have finished processing orders and wait idly for phone calls. He could thereafter leave and return home.

More generally, consider the following example. The general manager of the bank wants to learn the total amount of funds. To this end, he dials successively each clerk and asks him for the current balance of the accounts he manages. Then he sums all results together.

This naive procedure may lead to inconsistent results as shown by the following scenario. Consider clerks A and B above. The balance of account X is 200 and the balance of Y is 100. A transfer of 100 from X to Y has been ordered to A. The manager asks A and is answered that the balance of X is 200. Then the manager

asks some other clerk C while A and B process the transfer from X to Y. Then the manager asks B and is answered that the balance of Y is 200. An extra amount of 100 has appeared from 'behind-the-back'!

The underlying problem is to infer the global state of a system from a collection of local samplings. Unsynchronized samplings lead to an inconsistent global snapshot. Chandy and Lamport [6] call this problem the *Snapshot Problem*. Their analysis applies to a large number of situations in distributed computing including, for instance, distributed databases consistency check or token loss detection in ring networks. Most of previous work on those subjects had led to a number of ad hoc algorithms without dealing directly with the general underlying problem.

0.3. Tentative solutions

As remarked by Morgan [23], the Snapshot Problem may be directly related to the absence of any global time reference (clock). Suppose, for instance, that all clerks can refer to the *same* clock. Then the manager can order them to record balances at precisely 10 a.m. every day. Then later in the afternoon he asks each clerk for the recorded sums. The resulting view is an actual global state of the bank at 10 a.m. Observe that even with the help of such a procedure, the manager does not get an instantaneous view of the bank.

If no global clock is available, then the manager can nevertheless simulate a global rendez-vous. He first dials up successively all clerks and order them to suspend their work. Then in a second round, he asks them for the current balances. In a third round he lets them resume their normal activity. The resulting view is a global state of the accounts at the time the manager initiated the second round. The major drawback of this procedure is that it leads to *freeze* the system. Such situations have been analysed by Francez and Rodeh [14]. Freezing procedures are obviously unsatisfactory because they introduce new synchronization conditions into the original behavior of the system. Those conditions may preclude some original computations from occurring.

0.4. The Snapshot Algorithm

Chandy and Lamport [5, 20, 6] and Dijkstra [9] have recently proposed a general algorithm to solve this problem in the general framework of distributed systems with asynchronous communications. They call it the Snapshot Algorithm. Informally, a *snapshot* of a given computation is a global state which could have been observed by some external observer equipped with a suitable global clock (any space-time reference which satisfies Lamport's Clock Axioms of [19]). A snapshot can thus be seen as a *possible past* of the system.

Consider now predicates such as "the system is deadlocked", "some process has terminated" or "all processes have completed their task". All those predicates enjoy the following property: once they hold, they continue to hold. Such predicates are called *stable* by Chandy and Lamport. To test a stable predicate on a given

computation, it suffices to 'imagine' a snapshot of this computation: if the predicate is true for the snapshot, then it is true for the current state of the system. This basically the principle of the Shapshot Algorithm. In particular, it leads to straightforward and elegant solutions to the problem of deadlock detection and distributed termination detection in a distributed environment.

In this paper, we focus on the use of the Snapshot Algorithm in the framework of distributed systems with synchronous communications. In the first section, we rework the presentation of the (Single) Snapshot Algorithm of Chandy and Lamport in this framework. An improved version of this algorithm which handles *repeated snapshots* is described in Section 2. Section 3 describes the language CSP and the basic requirements for implementing the above algorithm. An implementation of the Repeated Snapshots Algorithm in Hoare's language of communicating sequential processes CSP is given in Section 4. This implementation is used in Section 5 to describe a *symmetric* and *generic* solution with *bounded overhead* to the Distributed Termination Detection Problem. A brief description of the Repeated Snapshots Algorithm in the framework of Petri nets is given in the appendix.

0.5. Related work

Much work has been done on the Distributed Termination Detection Problem in CSP. Early solutions [13] were not satisfactory because of the 'freezing' [14]: the original computation and the distributed termination detection waves were mutually exclusive. Next, solutions were not symmetric because one prescribed process had to initiate the detection waves [10, 11, 14, 15, 24]. Then, symmetric solutions were described. Rana [26] proposed a symmetric solution using global time stamps. This solution is not bounded in storage. Moreover, it is incorrect on several points as indicated in [2]. Apt and Richier [2], [29] improved the solution of Rana by using virtual (logical) clocks instead of real ones. Again, their solutions are not bounded in storage. Also, they use a predefined Hamiltonian circuit of the graph and are thus not generic. Our solution via snapshots is symmetric, generic, and has bounded overhead.

The Snapshot Algorithm is briefly (and somewhat cryptically) described in [5]. Chandy considers asynchronous (buffered) communications and partially ordered semantics.

A precise description can be found in [9] in the framework of abstract machines with asynchronous communications where two versions of the algorithm are studied. The first one uses white and red machines. The second one uses additional markers. Dijkstra considers an interleaving semantics.

The work by Lamport [20] considers asynchronous communications and partially ordered semantics. It contains an extensive discussion of the notion of global state (see also [19]) and reworks the description of [9]. But his algorithm does not allow for repeated snapshots. Also no indication is provided for a practical implementation of the algorithm, nor a complexity evaluation.

A very elegant approach to the Snapshot Algorithm is described by Morgan [23]. It shows that above versions of the algorithm can in fact be factorized into two separate algorithms:

(1) a modified version of the Snapshot Algorithm where *global time* is available to all processors; its correctness is then straightforward;

(2) a *clock synchronization algorithm* which simulates some weak kind of global clock from local clocks (for instance, Lamport's one in [19]).

A complete presentation of the Snapshot Algorithm is given by Chandy and Lamport in [6]. It considers asynchronous communications. The problem of collecting the recorded local states is only mentioned. No provision is made for repeated snapshots.

In [7, 31] it is shown that the Snapshot Algorithm is often too complex for practical needs. A large number of problems dealing with special subclasses of stable properties (quiescent properties, locally-indicative properties) can be solved using much simpler and more efficient algorithms.

1. The Single Snapshot Algorithm

This section describes the framework of distributed systems with synchronous communications. We use a partially ordered semantics. We then sketch the (Single) Snapshot Algorithm of Chandy and Lamport [5, 6, 20, 23].

1.1. Distributed systems with synchronous communications

A *sequential process* or machine can produce atomic and discrete events sequentially. Events are assumed to be uniquely identified. There exists three kinds of events:
- internal events which occur without any reference to the outside world,
- receiving a message of a given type on a given incoming channel, and
- sending a message of given type along a given outgoing channel.

Each process maintains an internal (local) state which can only be modified by the occurrence of its own events. Initial states are fixed. A given state determines the set of events which may occur next. A given state together with the occurrence of an allowed event determine the next state. A state of a process can thus be characterized here by the sequence of events which have yielded it.

A *distributed system* is a finite directed graph whose vertices are sequential processes and edges one-way communication channels. Process P is an in- (respectively out-) *neighbor* of process Q if there is a channel from P to Q (respectively Q to P). A system has *synchronous* communications if no message of a given type can be sent along a channel before the receiver is ready to receive (that is, in a state where the next action may be a reception of) a message of this type on this channel. For an external observer, the transmission then looks *instantaneous* and *atomic*. Sending and receiving a message correspond in fact to the *same* event.

Following Lamport [20], a *computation* of a system is viewed as a set of events equipped with a partial (pre-)order. The events produced by a given process from its initial state are totally ordered (sequentiality condition). The events of sending and receiving a message are identified (synchronous communications). The resulting partial order is called the *causality order* because two events are ordered if and only if they are causally dependent. Any external observer will observe the cause *before* its effect. Two unordered events are said to be *concurrent*. An external observer may then observe them in either order depending on his observation point.

Computations are partially ordered by inclusion. A computation is finite if its set of events is finite. It is maximal if it is maximal for inclusion among all computations. Otherwise, it is partial. Partial computation are exactly left-closed subsets of maximal computations (a partially ordered set C is left-closed if $f \in C$ and $e \leq f$ imply $e \in C$).

The usual notion of a global state is meaningless in this framework. Instead, Lamport uses the notion of a *slice* ([20], also called a cut) introduced by Petri in the framework of Petri nets. Let C be a partial computation of a system. For each process P consider the internal state of P after the last event produced by P in C (it is undefined if P produced infinitely many events in C). The slice associated with C is then the collection of those internal states. A slice can be viewed as a *possible* global state of the system which *could* have been observed by some external observer. Conversely, all such possible (observable) global states are slices. An example is displayed in Fig. 1, where circles denote states and bars events. The causality order is the order induced by transition arrows.

Following the characterization of states above, partial computations characterize slices. Conversely, if slices associated with different partial computations are distin-

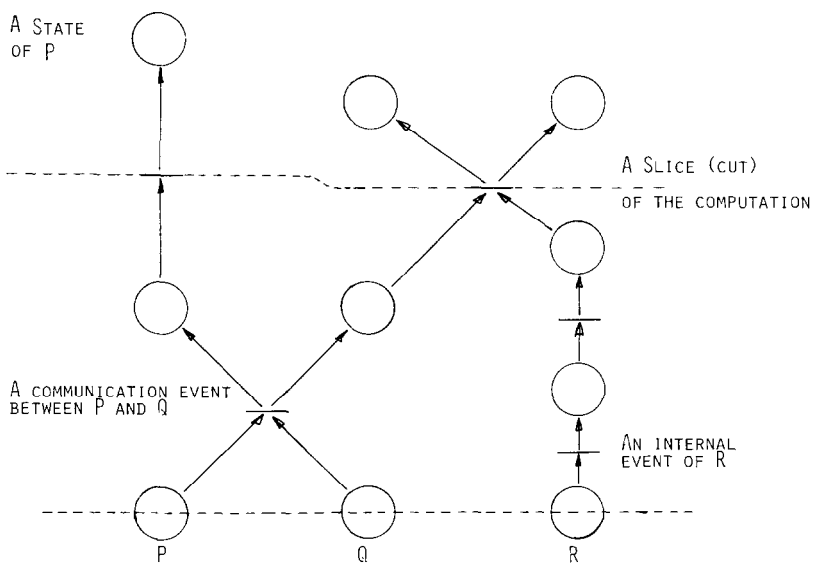

Fig. 1. A possible computation of $[P \| Q \| R]$.

guished in some way (see [5], for example), then the following property holds. Let C_0 and C_1 be two partial computations with slices S_0 and S_1. Then $C_0 \subseteq C_1$ iff S_0 is observed before S_1 by some observer which can observe both.

Let B be a predicate on the slices S of a system. B is *stable* if B is monotonic: for any computations $C_0 \subseteq C_1$ with respective slices S_0 and S_1, whenever B holds for S_0, then B holds for S_1. Examples of such predicates have been given in the introduction. We say that the system is *stable* whenever B holds. The fundamental property of stable systems is that they remain stable forever. As stated by Dijkstra [9]:

> "the purpose of the Snapshot Algorithm is to collect such (local) state information that, on the account of it, (global) stability can be detected."

1.2. White processes, red processes and marker waves

Let us look back at the manager example above. Fund transfers between clerks can be divided into three classes:

(1) both clerks have not yet been asked by the manager at the time of transfer;

(2) both have already been asked;

(3) one of them has not yet been asked whereas the other has already.

Only transfers of type (3) lead in fact to inconsistency. The problem is therefore to synchronize manager questions with respect to fund transfers so that such a case cannot occur. The idea is to associate a color to the current state of a clerk. Say, a clerk is in a white state initially. His state turns red at the time he is asked by the manager. The rule is then that fund transfers may only occur between clerks in states of the same color. Obviously, this rule prevents type (3) transfers from taking place. In particular, if a clerk in a red state wants to transfer funds to one in a white state, then he has to wait for this clerk to get asked by the manager.

This idea can be readily adapted to the general case of distributed systems with synchronous communications. Processes can now paint themselves in white or red. A process turning from white to red samples its local state. The rule is then that messages can be exchanged only between processes of the same color.

How can we enforce this last rule to hold? The Snapshot Algorithm consists in allowing processes to exchange a special extra kind of messages called *markers* along communication channels. Markers are used by red processes to ensure that their partner in communication has also turned red.

The (Single) Snapshot Algorithm is summed up by marker rules below. The word 'basic' refers here to the original system whereas 'marked' refers to the system equipped with markers.

Marker rules

(1) Initially all processes are white. A white process turns red by suspending its basic computation and sampling its *basic* internal state. Then it resumes its basic computation.

(2) A white process can turn red spontaneously at any time. A white process eventually turns red. A red process cannot turn back white.

(3) A white process is always ready to receive markers. On receiving a marker, it turns immediately red.

(4) A red process is always ready to receive markers. On receiving a marker, it ignores it. A red process sends only a finite number of markers. It eventually sends at least one marker along each channel.

(5) A red process does not accept any (basic) message on a channel before having received a marker on it. It does not send any (basic) message along a channel before having sent a marker along it.

We now state the main properties of systems equipped with markers. The *projection* of a marked computation is the partially ordered set of events obtained by forgetting all about markers and colors.

Lemma 1.1. *The projection of a marked computation is a basic (possibly partial) computation.*

Proof. Markers and colors do not modify basic transitions of processes. They may only delay or preclude certain transitions by rule (5). □

Lemma 1.2. *Basic messages are only exchanged between processes of the same color.*

Proof. Consider a basic communication from process P to process Q. Suppose P is white. If Q is red, then it cannot accept P's message before having received a marker from P by rule (5). Thus it cannot accept P's message before P has sent a marker, and thus before P has turned itself red. Q is thus white.

Suppose P is red. By rule (5), P cannot send any message to Q before having sent a marker to Q. Because communications are synchronous, P cannot send any message to Q before Q has received a marker, and thus before Q has turned red by rule (3). □

Property 1.1 (Termination). *Each process samples its local state exactly once.*

Proof. This is a consequence of rule (2). □

By Property 1.1 all processes sample eventually their local state. Let SSS (snapshot state) be the global state obtained by putting together all those local states. Property 1.2 shows that SSS is actually a consistent snapshot of the system.

Property 1.2 (Soundness). *Let C be the projection of a marked computation. Then* SSS *is a slice of C associated with a finite partial computation* SSC.

Proof. Define the following set of events:

$$\mathrm{SSC} = \{e \mid e \text{ is a basic event produced by a white process}\}.$$

We first show that SSC is well-defined. If e is an internal event of a process P, then the color of P is either white or red. If e is a communication from P to Q, then, by Lemma 1.2, P and Q have the same color. We then show that C is left-closed. Let e be an event produced by a white process P and let f be an event such that $f \leqslant e$. Because events are discrete, we may assume there is no event between f and e. Then they are necessarily produced by the same process (we identify the events of sending and receiving a message) and we can apply rule (2).

Thus SSC is actually a partial computation of C. It is finite by rule (2). The associated slice is namely SSS. □

We now have to check that basic computations are not distributed by markers and colors, and that any finite slice may be a snapshot.

Property 1.3 (Completeness). (1) *The projection of a maximal marked computation is a maximal basic computation.*

(2) *Let C be a maximal basic computation and S_0 be a slice of C associated with a finite partial computation C_0. Then there exists a maximal marked computation which has C as projection, S_0 as* SSS *and C_0 as* SSC.

Proof. To prove part (1), we first show that no deadlock is introduced by markers. Rule (2) ensures that eventually all processes turn red. Rule (3) and (4) ensure that processes are always ready to receive markers. By rule (4), at least one marker is eventually sent along each channel. Thereafter, markers cannot hinder the occurrence of any transition. By rule (4), at most a finite number of markers are exchanged. Thus, by Lemma 1.1, the projection of a maximal computation is maximal.

To prove part (2), consider the following scenario. Each process proceeds up to the last event in C_0 (it is finite by hypothesis). Eventually, all processes have reached this event. Then they turn red all together and send exactly one marker along each channel. They finally resume their basic computation. This is a valid marked computation. The resulting SSC is namely C_0. □

The purpose of the Snapshot Algorithm is to provide a general way of testing whether a (global) *stable* predicate B on slices (respectively finite partial computations) holds. We say that B is *detected* if it holds for the snapshot state SSS (respectively SSC).

Property 1.4 (Correctness). (1) *If B is detected, then it holds forever later on.*

(2) *If B holds for some slice where all processes are still white, then it is eventually detected.*

Proof. Part (1) stems from Property 1.2 and the definition of stability.

For part (2), suppose that B holds for a slice S_0 associated with a finite partial computation C_0 where all processes are white. By Lemma 1.2, $C_0 \subset$ SSC. Thus, by stability, B holds for SSS as well. □

2. The Repeated Snapshot Algorithm

The Single Snapshot Algorithm above is not fully satisfactory. If B does not hold in the snapshot state SSS, then one cannot conclude anything about its current validity. Let us return to the first example. Late in the night, when no orders are delivered to clerks any more, the predicate B "all clerks have finished their task" is stable. Then clerks may use the Snapshot Algorithm to detect that B holds and then return home. Suppose that unfortunately they initiate the algorithm too early so that B does not hold for the resulting snapshot state. Clerks cannot take any decision on this basis alone. They thus have to repeat the algorithm later to take repeated snapshot of their situation at various time. Eventually, one will be initiated in a state where B holds. By Property 1.4, detection is then guaranteed.

We therefore have to allow the system to take repeated snapshots whenever the first one is not successful. It suffices in fact to introduce a rule to let processes whiten. Yet, one must be careful enough to avoid that successive marker waves interfere. An easy solution is to label markers with the number of their wave, but the number of needed waves is obviously unbounded in general. We thus prefer to strengthen slightly the marker rules. The idea is that a red process may turn back to white only after having sent and received *all* markers it could ever exchange within the current wave. We thus substitute rules (2) and (4) with the following rules.

Marker rules (continued).

(2′) A white process can turn red spontaneously at any time. A white process eventually turns red. A red process which has exchanged *exactly one* marker on each channel eventually turns white.

(4′) A red process accepts at most one marker on each incoming channel. It eventually sends exactly one marker along each outgoing channel.

We state below some properties of the modified algorithm called the Repeated Snapshots Algorithm. A process is in its nth *phase* if it has turned red exactly n times.

Lemma 2.1. *By the time P ends its n-th phase, it has exchanged exactly n markers on each channel.*

Lemma 2.2. *Basic messages are only exchanged between processes being in the same phase.*

Proof. If P sends a basic message to Q in its nth phase, then it has sent exactly n markers to Q beforehand. Because communications are synchronous, Q has already received exactly n markers from P. Q is thus at least in its nth phase. Also Q cannot be in phase $(n+1)$ because of rule (5). □

Property 2.1 (Liveness). *For all n, P eventually enters its n-th phase.*

Proof. Suppose P gets stuck for ever in its nth phase. Then, for some neighbor Q, P will never exchange its nth marker with Q. Thus Q will never reach the end of its nth phase. If Q were in its nth phase, nothing could prevent it from exchanging an nth marker with P. So Q gets stuck in its mth phase, with $m < n$. A contradiction is then obtained by induction. □

Property 2.2 (Soundness). *Let C be the projection of a marked computation. Then for all n,* SSS_n *is a slice of C associated with a finite partial computation* SSC_n.

Proof. Applying the method used for Property 1.2, we define the following set of events:

$$\mathrm{SSC}_n = \{e \mid e \text{ is a basic event produced by a process within its } k\text{th phase, } k < n\}.$$

Lemma 2.2 shows that SSC_n is well-defined. It is left-closed. It is thus a partial computation. It is finite and its associated slice is SSS_n. □

Property 2.3 (Completeness). *Let C be a maximal basic computation. Let* (C_n) *be a strictly increasing sequence of partial subcomputations of C, associated with slices* (S_n). *Then there exists a maximal marked computation which has C as projection,* S_n *as* SSS_n *and* C_n *as* SSC_n *for all n.*

Proof. Apply the scenario described in the proof of Property 1.3 for each C_n successively. □

Property 2.4 (Correctness). *Let C be the projection of a maximal marked computation* C'. *Let* C_0 *be a finite partial subcomputation of C. Then, for some n,* $C_0 \subseteq \mathrm{SSC}_n$.

Proof. Let C'_0 (respectively C'_n) be a finite partial marked computation whose projection is C_0 (respectively SSC_n). Each process produces at least one event (possibly not a basic event) within each phase. Because C'_0 is finite, by Property 2.1, eventually $C'_0 \subseteq C'_n$ for some n. Thus $C_0 \subseteq \mathrm{SSC}_n$. □

Corollary. *If B holds for a slice associated with a finite partial computation, then B is eventually detected.*

The Repeated Snapshots Algorithm thus satisfies our initial requirement. Unfortunately, we have to pay for that because there is no counterpart to part (1) of Property 1.3. The basic computation may get flooded by markers, so that processes will spend all their time turning from white to red and then from red to white. We thus need to prevent white processes from turning red 'too often'. This may be viewed as a *fairness* requirement about the interleaving between the basic computation and markers management. A possible step toward a solution is to strengthen rule (2').

Marker rules (continued)

(2″) A white process can turn red spontaneously only if it has produced at least one basic event since the last time it turned red spontaneously (if any). If such an event has occurred, then it eventually turns red. A red process which has exchanged exactly one marker on each channel eventually turns white.

Under this rule, projections have the following properties.

Property 2.5. (1) *The projection of a maximal marked computation involving a finite number of basic events is a maximal basic computation.*

(2) *The projection of an infinite marked computation is an infinite basic computation.*

Proof. At least one basic event is produced within each phase. □

Yet it is not true that the projection of an infinite maximal computation is an infinite maximal computation because the required triggering events may be always produced by the same process whereas other processes get nevertheless overflooded by markers. At this time, we cannot see any simple condition enforcing the required fairness.

However, Property 2.5 suffices if the condition to be detected by the algorithm is *quiescent* [7]. A condition B is quiescent if whenever B holds, then no process can produce any more event (the system is deadlock). Obviously, quiescent conditions are stable too. A predicate such as "All processes have finished their job" used in the Distributed Detection Problem is a typical example of a quiescent condition. For such conditions, we have the following property. Let N be the number of processes in the system.

Property 2.6. *If a quiescent predicate B holds, then the (marked) computation deadlocks within at most N extra phases. All state snapshots after B holds are exactly the terminal state of the basic computation (the quiescent state).*

Thus if B happens to hold, then it will be detected *before* the (marked) computation deadlocks. A normal termination can thus be decided by processes instead of deadlock.

An extra point of interest is the notion of *freezing* introduced in Section 0.3. The Repeated Snapshots Algorithm is freezing in the sense of [13] because the interleaving between basic actions and nonbasic actions of a process is not arbitrary. Consider for instance a process P being in phase n which is ready to send a basic message to some process Q still in phase $n-1$. The message cannot be exchanged until both processes are in the same phase. But Q cannot end its phase $n-1$ before all its neighbors have ended their phase $n-2$ etc. In the worst case, the basic communication between P and Q will not be allowed to occur before $O(E)$ markers have been exchanged in the network, where E is the number of communication channels.

3. The language CSP

Hoare's language of Communicating Sequential Processes (CSP) allows to describe distributed systems with synchronous communications. Most work about Francez's Distributed Termination Problem has been carried out in the framework of CSP. An implementation of the Repeated Snapshots Algorithm thus provides a straightforward solution to this problem and several other ones. It will appear that this solution enjoys very interesting properties.

This section briefly describes the language CSP and the notions of symmetry and genericity for CSP distributed systems.

3.1. Distributed systems in CSP

An extensive description of the language CSP can be found in [16]. An interleaving operational semantics is given in [25] and some insights into a partially ordered one can be found in [27]. We disregard in this paper the Distributed Termination Convention of CSP.

A *CSP process* is a sequential program built with usual assignment instructions. Moreover, two *remote assignment* instructions are available:

- P!flag(v): sends the message of type flag with value v to process P;
- Q?flag(x): receives a message of type flag from process Q and stores its value into variable x.

Communications are *synchronous*. Sending and receiving a message are simultaneous and atomic. A message cannot be sent before its recipient is in a state where it can receive such a message.

Two extra control structures are available. The nondeterministic selection is denoted

[$\text{guard}_1 \rightarrow \text{command}_1$
■ $\text{guard}_2 \rightarrow \text{command}_2$
■ ...
■ $\text{guard}_p \rightarrow \text{command}_p$]

Each guard is a sequence of boolean expressions (default is the constant true) possibly followed by a communication command. A guard is *open* if
- its boolean expressions evaluate to true;
- its communication command, if any, can be currently performed.

A guard is *closed* if one of its boolean expressions evaluates to false.

The semantics of nondeterministic selection is the following. One of the open guards is chosen nondeterministically. The required communication command is performed and control is given to the corresponding command. If all guards are closed, then a failure occurs and the computation is aborted.

The nondeterministic repetition is denoted

$$
\begin{aligned}
&*[\ \text{guard}_1 \rightarrow \text{command}_1 \\
&\ \blacksquare\ \text{guard}_2 \rightarrow \text{command}_2 \\
&\ \blacksquare\ \ldots \\
&\ \blacksquare\ \text{guard}_p \rightarrow \text{command}_p]
\end{aligned}
$$

One open guard (if any) is nondeterministically chosen. It is then passed and the associated command is executed. Then the loop is entered again. If all guards are closed, then the loop is exited.

A *CSP distributed system* is a parallel composition of processes P_u, $u = a, b, \ldots$. It is denoted

$$P :: [P_a \| P_b \| \cdots].$$

We assume that P is well-formed in the sense that P_u contains an output command addressing P_v if and only if P_v contains an input command from P_u with matching type. Process P_u (respectively P_v) is then the in- (respectively out-) neighbor of process P_v (respectively P_u). The *communication graph* G of P has component processes P_u as vertices. There is an edge from P_u to P_v in G if and only if the former is an in-neighbor of the latter. We assume here that G is *strongly connected.*

As shown in [1], we may assume without loss of generality that all processes P_u are in Distributed Normal Form

$$
\begin{aligned}
&P_u :: [\text{Init} \\
&\quad *[\ \text{guard}_1 \rightarrow \text{command}_1 \\
&\quad\ \blacksquare\ \ldots \\
&\quad\ \blacksquare\ \text{guard}_p \rightarrow \text{command}_p]]
\end{aligned}
$$

where *all* communication commands are in the guards of the outer repetition statement. We assume throughout this paper that no divergence may occur within Init or command_k, $k = 1, \ldots, p$. A practical sufficient condition would, for instance, be that no repetition statement appears there. We also assume that all systems are correctly written, so that no failure occurs.

In displaying a CSP distributed system, we use the following *template notation.* In a process template P_u, symbol In (respectively Out) stands for the set of in- (respectively out-) neighbors of P_u and is ranged by index 'in' (respectively 'out'). Indexed guards stand for the set of guards obtained by substituting syntactically the formal index with each value of its range.

3.2. Symmetry

A first requirement for our implementation of the Repeated Snapshots Algorithm is symmetry. The abstract marker rules of Section 1 and 2 do not privilege any

process. All processes have the same rights and duties with respect to marker handling. More precisely, if the set of basic computation exhibits some kind of symmetry, then so will do the set of marked computations. We have developed a notion of symmetry well-suited for CSP distributed systems. A detailed presentation can be found in [3, 4]. Here we briefly sketch the main definitions.

Let P be a CSP distributed system with communication graph G. An *automorphism* of G is a permutation of vertices which preserves edges. If $u \to v$ is a directed edge of G, then so is $\sigma(u) \to \sigma(v)$. A particular automorphism is the identity on G which we denote Id. The automorphisms of G form a group which we denote Σ_G.

Let PCOMP be the set of partial computations C of P. The *trace* C_u of a computation C on a process P_u is the sequence of the typed communications that P_u performs within C followed possibly by a tag indicating the proper termination of P_u. Trace C_u is thus of one of the following forms:

$$\langle P_{v_1}\$\text{type}_1, \ldots, P_{v_p}\$\text{type}_p\rangle$$

$$\langle P_{v_1}\$\text{type}_1, \ldots, P_{v_p}\$\text{type}_p, \text{terminate}\rangle$$

$$\langle P_{v_1}\$\text{type}_1, \ldots, P_{v_p}\$\text{type}_p, \ldots\rangle,$$

where \$ stands for ? or !.

An automorphism σ of G naturally induces a mapping T_σ on traces by changing all process names from P_v to $P_{\sigma(v)}$. For instance, if C_u is

$$\langle P_{v_1}\$\text{type}_1, \ldots, P_{v_p}\$\text{type}_p, \text{terminate}\rangle,$$

then $T_\sigma(C_u)$ is

$$\langle P_{\sigma(v_1)}\$\text{type}_1, \ldots, P_{\sigma(v_p)}\$\text{type}_p, \text{terminate}\rangle.$$

Symmetry can now be expressed as follows. Let C be a computation of P with traces C_u. Let σ be an automorphism of G. Then one should be able to find a computation C' such that, for all u, $P_{\sigma(u)}$ has the same external behavior in C' as P_u in C up to renaming by T_σ. In formulas,

$$\text{for all } u, \quad C'_{\sigma(u)} = T_\sigma(C_u).$$

The association $C \to C'$ thus defines a mapping over PCOMP which we denote S_σ. This mapping should moreover be consistent with the group operations defined over Σ_G, and the ordering of the partial computations of PCOMP.

Definition 3.1. P is symmetric if, to each automorphism σ of G, there corresponds a mapping S_σ over the set PCOMP of P's partial computations such that

(1) for all u, σ, C, $T_\sigma(C_u) = (S_\sigma(C))_{\sigma(u)}$;

(2) for all σ, ρ, C, $S_{\text{Id}}(C) = C$ and $S_{(\sigma\rho)} = S_\sigma(S_\rho(C))$;

(3) for all σ, C, C', $C \subseteq C'$ implies $S_\sigma(C) \subseteq S_\sigma(C')$.

3.3. Genericity

An additional requirement for our implementation concerns the extra information about the network processes needed to run the algorithm. An algorithm is generic if processes do not need global information about the network they are embedded in such as the number of nodes, a spanning tree, a hamiltonian cycle, etc. Only local information such as the name of the process and of its in- and out-neighbors is then available to each process initially. Global information has to be explicitly learned through the cooperation of all processes. This intuition is captured by the subsequent definition [4]. In defining genericity, we identify an algorithm with the family of all its instances, one for each underlying network.

Let $\mathcal{P}$ be a family of distributed systems, built on a family of networks $\mathcal{G}$. Let P and P' be two systems of $\mathcal{P}$ built respectively on G and G'. The *intersection* of G and G', which we denote $G \cap G'$, is the set of those vertices u which belong to both G and G' and have (in both) the same in- and out-neighbors. Graphs G and G' thus cannot be distinguished at those vertices. Processes P_u and P'_u attached to those vertices should therefore be syntactically identical. This guarantees that, in the family $\mathcal{P}$, processes depend only on the local topology of the communication graph and not on global parameters (e.g., the numbers of processes in the graph).

Definition 3.2. *A family $\mathcal{P}$ of distributed systems built on a family of networks $\mathcal{G}$ is* generic *if, for any system P and P' of $\mathcal{P}$ built respectively on networks G and G', P_u is syntactically identical to P'_u for all vertices u in $G \cap G'$.*

4. Implementing repeated snapshots in CSP

Several goals are to be achieved while implementing the Repeated Snapshots Algorithm of Section 2 in CSP.

(1) Define the notion of local state and slice for CSP processes; define the notion of (local) state sampling.

(2) Implement rules (1), (3), and (4') which specify markers handling.

(3) Implement rules (2") and (5) which specify the interactions between markers and basic messages.

(4) Last but not least, provide a way in which processes can learn local states of other processes so that eventually all processes collect the *same* snapshot at each phase.

Moreover, we wish the transformation to preserve several important properties of systems. It should be *symmetric*, in the sense that it maps symmetric systems to symmetric systems. It should be *generic*, in the sense that it maps generic families of systems to generic ones. Finally, the additional space needed by the new variables and messages should be bounded.

4.1. Syntactical requirements

We define the *local state* of a CSP process P_u to be the collection of the values of its variables when the control is at the outer level of its main while-loop. This amounts in fact to considering that passing a guard and executing the associated command constitutes an *atomic* step of computation. Note that this definition makes sense because processes are in Distributed Normal Form. By the hypotheses stated in Section 3.1, a process which passes a guard will eventually complete the execution of the command. We assume that we are given a function SAMPLE which returns the values of (basic) variables in the last local state.

Let P be a CSP distributed system in normal form. The Repeated Snapshots Algorithm is implemented by adding extra initializations of variables, extra guarded commands in the main loop, and prefixing original guards and commands with extra boolean guards and assignments. Moreover, we require that *all* variables and message types used in those extra constructs are new. Those conditions then guarantee that the projection of a marked computation is actually a (possibly partial) computation of the original system.

4.2. Implementing markers

Rule (1) is implemented by adding a new variable 'red' with obvious meaning. The event of turning red spontaneously is expressed by the following new guarded command.

$$\blacksquare\ \neg\text{red} \rightarrow \text{state} := \text{SAMPLE};\ \text{red} := \textbf{true}. \tag{1}$$

Markers are implemented by messages (signals) of a new type 'marker'. Receipts and sendings of markers are recorded in two arrays 'received[In]' and 'sent[Out]'. Because of rules (2″) and (4′), at most one marker is exchanged on each channel. We therefore can use arrays of boolean values, initially set to false. Rule (3) is implemented by the following new guarded commands

$$\blacksquare_{\text{in}}\ \neg\text{red};\ P_{\text{in}}?\text{marker}(\ )$$
$$\rightarrow \text{state} := \text{SAMPLE};\ \text{red} := \textbf{true};\ \text{received[in]} := \textbf{true}.$$

Rule (4′) is implemented in a similar fashion taking into accoutn rule (2″).

$$\blacksquare_{\text{in}}\ \text{red};\ \neg\text{received[in]};\ P_{\text{in}}?\text{marker}(\ )$$
$$\rightarrow \text{received[in]} := \textbf{true};$$

$$\blacksquare_{\text{out}}\ \text{red};\ \neg\text{sent[out]};\ P_{\text{out}}!\text{marker}(\ )$$
$$\rightarrow \text{sent[out]} := \textbf{true}.$$

We can easily implement rule (5) using arrays 'received' and 'sent'. An input- (respectively output-) guarded command is a guarded command whose guard contains an input (respectively output) command. A boolean guarded command is

a guarded command whose guard is purely boolean. Consider an input guarded command of the form

■ bool; P_{in}?message → command.

We transform it into

■ (received[in] **or** ¬red); bool; P_{in}?message
→ command. (2)

Accordingly, an output-guarded command of the form

■ bool; P_{out}!message → command

is transformed into

■ (sent[out] **or** ¬red); bool; P_{out}!message
→ command. (3)

Finally, boolean guarded commands are left unchanged.

To implement rule (2″), we introduce a new variable 'moved'. This boolean variable is true if and only if a basic step has been taken by the process since the last time it turned red spontaneously, if any. Initially, it is true. We modify guarded command (1) above as follows.

■ ¬red; moved
→ state := SAMPLE; red := **true**; moved := **false**.

Also, all original guarded commands, possibly modified into (2) and (3), are modified so that variable 'moved' is set to **true** whenever the command is selected:

■ bool;communication → moved := **true**; command,

■ bool → moved := **true**; command.

4.3. Diffusing local states through the network

The last problem to be solved is to diffuse local states yielded by calls to SAMPLE through the whole network so that all processes eventually learn the *same* snapshot state. This diffusion can easily be implemented using standard *Distributed Learning Algorithms*. An example of such an algorithm is based on the following rules (see [4] for details).

Distributed learning rules

(1) I send everything new that I have come to learn to all my out-neighbors.
(2) I am always ready to learn from my in-neighbors.

(3) I terminate when I have collected all available information, when I have sent everything to my out-neighbors and I have received from all my in-neighbors all information available to them.

We let each process maintain a new variable 'snapshot' where it accumulates the local states it has learnt. Each local state is labeled with the name of the process it has originated from. A process sends out the content of its variable 'snapshot' each time it is increased. These sendings are recorded in an array 'forwarded[Out]' of boolean values. Variable 'forwarded[out]' is true whenever the current value of variable 'snapshot' has been sent to P_{out}. The array 'forwarded' is initially set to false (we make the convention that assigning a value to an array resets all its cells to this value). Rules (1) and (2) are implemented (admittedly inefficiently) as follows

```
■in Pin ?info(new snapshot)
   → old snapshot := snapshot;
     snapshot := snapshot ∪ new snapshot                    (4)
     [snapshot ≠ old snapshot → forwarded := false
     ■ snapshot = old snapshot → skip],

■out ¬forwarded[out]; Pout !info(snapshot)
   → forwarded[out] := true.
```

Also, a process P_u may learn its own local state. We then have to transform all occurrences of

```
state := SAMPLE
```

into

```
state := SAMPLE;
snapshot := snapshot ∪ {(u, state)}                         (5)
[snapshot ≠ old snapshot → forwarded := false
■ snapshot = old snapshot → skip].
```

We now implement rule (3). By hypothesis, the communication graph G of P is strongly connected. All processes eventually sample their own local state. It can then be shown that, eventually, all processes will store in their variable 'snapshot' the local states of all processes. Suppose for a while that processes know the number N of processes in the system. Then a process has collected all available information if and only if

$$|\text{snapshot}| = N,$$

where $|.|$ denotes the cardinality of a set. We define a new array 'completed[In]' of boolean values, initially set to false. 'completed[in]' is true in process P_u if and only if a snapshot with cardinality N has been received from P_{in}. Then P_{in} has collected

all available information and will never send anymore messages within the current phase. Observe also that the variable 'snapshot' in P_u then necessarily has cardinality N, and thus satisfies the condition above. Rule (3) can therefore be implemented by adding the following command to (4) and (5):

[|new-snapshot| = N → completed[in] := **true**
■ |new-snapshot| ≠ N → **skip**].

Then guarded commands of (4) need only be considered if 'completed[in]' is false. For sake of concision, we define UPDATE(new-snapshot) as follows:

old-snapshot := snapshot;
snapshot := snapshot ∪ new-snapshot
[snapshot = old-snapshot → **skip**
■ snapshot ≠ old-snapshot → forwarded := **false**]
[|snapshot| = N → completed[in] := **true**
■|snapshot| ≠ N → **skip**].

(In the case 'in' is not defined, the last alternative is simply omitted.)

We finally obtain the following forms for (1):

state := SAMPLE; UPDATE({(u, state)})

and for (4):

$■_{in}$ ¬completed[in]; P_{in} ?info(new-snapshot)
→ UPDATE(new-snapshot).

The termination of the Distributed Learning Algorithm is then detected by the following (local) condition.

$\bigwedge_{in}$ completed[in]; $\bigwedge_{out}$ forwarded[out].

Remark that processes may start learning snapshots before turning red themselves.

A process can turn back white as soon as it has exchanged all markers and completed the diffusion of the snapshot state. This is expressed by a new guarded command

■ red; $\bigwedge_{in}$ received[in]; $\bigwedge_{out}$ sent[out];
$\bigwedge_{in}$ completed[in]; $\bigwedge_{out}$ forwarded[out]
→ received := **false**; sent := **false**;
completed := **false**; forwarded := **false**;
snapshot := ∅; red := **false**.

It can be checked that, similarly to markers, this discipline guarantees that local states diffusion waves do not interfer. The value of the variable 'snapshot' when

turning white for the nth is the same for all processes and is actually the nth snapshot state of the system.

The transformation described in this section is symmetric because no test involves names of processes. The transformation is not generic because the number N of processes is used by processes explicitly. Yet, general results of [4] show that such a transformation can be turned into a generic equivalent one (and still symmetric). It suffices in a first phase to let processes *learn* the number N using a special version of the Distributed Learning Algorithm. The transformation has also a bounded storage overhead. The size of control variables and messages can be statically bounded from the original algorithm.

5. Application to the Distributed Termination Problem

We apply the implementation described in Section 4 to solve the Distributed Termination Problem of Francez. We assume that we are given a distributed system P in Distributed Normal Form whose communication graph is *strongly connected.* We assume that all computations of P deadlock. We assume, following Francez, that we can associate to each process P_u a boolean condition on its local state B_u with the following property: P deadlocks if and only if all conditions B_u are true and control in each process P_u is at the outer level of the while-loop. Observe that, contrary to [13], not all guards of processes have to be communication guards. The problem is to transform P into an equivalent distributed system which always terminates properly instead of deadlocking.

Let B be the conjunction of conditions B_u. B is a stable (even quiescent) predicate. Because P may only deadlock when control in all its processes is at the outer level of the while-loop, we may consider that B is a predicate on the *slices* of P in the sense of Section 3. We can therefore apply the algorithm of Section 4.

Here, the function SAMPLE yields a one-bit result, namely the value of condition B_u. Variable 'state' is thus a one-bit variable. We define a new variable 'halt' initially set to false. This variable is true if and only if termination has been detected. Then it forces the proper termination of the process by closing all its guards. The procedure DETECTED yields the conjunction of those one-bit states collected in variable 'snapshot', that is, B. Termination is thus detected if and only if DETECTED is true. The algorithm is displayed in Fig. 2.

This solution to the Distributed Termination Problem enjoys the following properties. It is symmetric. Using the transformation mentioned in Section 4, it can be made generic. The additional space needed for the control variables and messages is bounded. More precisely, let N be the number of processes, E the number of channels in the network, and D the maximal number of neighbors of a process. As far as markers are concerned, $O(D)$ control bits are used at each process, and $O(E)$ signals are exchanged at each phase in the network. Each snapshot state is stored on $O(N \log N)$ bits. On each channel, at most N copies of it are sent. This amounts

```
/* the transformed algorithm */
P_u ::[
/* initialization of control variables */
      received := false; sent := false;
      completed := false; forwarded := false;
      moved := true; halt := false; snapshot := ∅; red := false;
/* the original Init part*/
      Init
/* the main while loop*/
      *[
/* the original boolean guarded commands */
        ■ ¬halt; bool → moved := true; command
/* the original input guarded commands */
        ■ ¬halt; (received[in] or ¬red); bool; P_in?message
          → moved := true; command
/* the original output guarded commands */
        ■ ¬halt; (sent[out] or ¬red); bool; P_out!message
          → moved := true; command
/* to turn red spontaneously */
        ■ ¬halt; moved; ¬red
          → state := SAMPLE; UPDATE({(u, state)});
            red := true; moved := false
/* to turn red on receiving a marker */
        ■_in ¬halt; ¬red; P_in?marker( )
          → state := SAMPLE; UPDATE({(u, state)});
            red := true; received[in] := true
/* to receive a marker once red */
        ■_in ¬halt; ¬received[in]; red; P_in?marker( )
          → received[in] := true
/* to send a marker once red */
        ■_out ¬halt; ¬sent[out]; red; P_out!marker( )
          → sent[out] := true
/* to receive a snapshot */
        ■_in ¬halt; ¬completed[in]; P_in?info(new-snapshot)
          → UPDATE(new-snapshot)
/* to send one's snapshot */
        ■_out ¬halt; ¬forwarded[out]; P_out!info(snapshot)
          → forwarded[out] := true
/* to whiten or terminate */
        ■ ¬halt; red
        ⋀_in received[in]; ⋀_out sent[out];
        ⋀_in completed[in]; ⋀_out forwarded[out]
          → [DETECTED(snapshot) → halt := true
            ■ ¬DETECTED(snapshot)
              → received := false; sent := false;
                completed := false; forwarded := false;
                snapshot := ∅; red := false]
      ]]
```

Fig. 2.

to $O(N^2E \log N)$ extra bits exchanged at each phase. There are at most as many phases as basic steps, plus one preliminary exploration phase to learn the total number of processes in the network. Termination is detected at most one phase after it occurs.

6. Conclusion

Chandy and Lamport have designed the Snapshot Algorithm as a general tool for detecting stable properties of distributed systems. Unfortunately, their algorithm allows processes to take *only one* snapshot of the system. If the stable property happens not to hold in this snapshot state, then no provision is made for a later snapshot.

The Repeated Snapshots Algorithm described in this paper handles this problem. Here processes can take several snapshots of the system. Care is taken so that successive phases of the algorithm do not interfere, and that taking snapshots does not overflood the system. The algorithm is *symmetric* because all processes have similar rôles in handling snapshots. It is *generic* in that they need no global information about the system (e.g., the number of processes, a predefined Hamiltonian path or whatsoever) they participate in. It has bounded storage and bit-complexity (per phase) overhead. In particular, no unbounded sets of labels are used.

The price to pay for this is the need for synchronous communications. Nevertheless, even under this restriction, the Repeated Snapshots Algorithm applies to a number of classical problems. In this paper we have described extensively a solution to the Distributed Termination Problem of Francez in the framework of Hoare's language of Communicating Sequential Processes. We then obtain a symmetric, generic solution with bounded overhead which applies to any strongly connected graph.

The bit-complexity of this solution is rather high. This is mainly due to the diffusion and collection of local states using the Distributed Learning Algorithm, and not to markers. Nevertheless, this complexity can be easily improved by optimizing the coarse implementation we give.

As far as we know, this solution to the Distributed Termination Problem was the first to enjoy such properties. Since, Shavit and Francez [31] have designed a solution to this problem, based on a symmetrization of the algorithm of [11], which has properties similar to ours and a better complexity. Yet, their solution is specifically designed for this particular problem, whereas our solution can be adapted to detect any kind of stable properties, even non-locally indicative ones. It suffices in fact to use the ad hoc procedures SAMPLE and DETECTED.

The Snapshot Algorithm appears as one of the fundamental paradigms in distributed computing. It would be interesting to express and study it in an abstract framework, independently of its many implementations. It seems that the notion of knowledge provides the right level of abstraction to carry out this work.

Acknowledgment

This work could not have been done without the encouragement and help of Krzysztof Apt. Remarks of P. Gastin, W. Reisig and referees have been very useful. The idea of taking snapshots of Petri nets is due to W. Reisig.

Appendix A. Snapshooting Petri nets

The Snapshot Algorithm can be described quite simply in the framework of Petri nets. In this appendix, we assume from the reader some familiarity with Petri nets (see [28] for an introduction). A transition of a net fires by deleting one pebble from each input place and creating one extra pebble on each output place. An arbitrary number of input and output places is allowed for each transition. In this framework, rules can be expressed as follows.

Marker rules

(1) Initially all pebbles are white. A pebble turns from white to red by recording its current place.

(2) A white pebble can turn red on any place. Eventually, all pebbles are red.

(3) A transition may fire only if all deleted pebbles have the same color. Then, created pebbles also have this color.

Let SSS be the configuration obtained as follows. Each place owns as many pebbles as the number of times it has been recorded by virtue of rule (1). Then, the following properties hold.

Property A.1 (Soundness). *A configuration* SSS *is reachable from the initial configuration.*

Property A.2 (Completeness). *Any reachable configuration can be obtained in this way.*

References

[1] K.R. Apt and Ph. Clermont, Two normal form theorems for CSP programs, Rept. No. RC10975, IBM Scientific Center, Yorktown Heights, NY (1985).

[2] K.R. Apt and J.L. Richier, Real time clocks versus virtual clocks, in: M. Broy, ed., *NATO ASI Series F 14, Control Flow and Data Flow: Concepts of Distributed Programming* (Springer, Berlin, 1985) 475-501.

[3] L. Bougé, On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes, Rept. No. 86-18, LITP, Univ. Paris 7 (1986), to appear in *Acta Inform.*

[4] L. Bougé, Symmetry and genericity for CSP distributed system, Rept. No. 85-32, LITP, Univ. Paris 7, Paris (1985).

[5] K.M. Chandy, Paradigms for distributed computing, in: *Proc. 3rd Conf. on Foundations of Software Technology and Theory of Computer Science*, Bangalore, India (1983) 192-201.

[6] K.M. Chandy and L. Lamport, Distributed snapshots: determining the global state of distributed systems, *ACM Trans. Comput. Systems* **3**(1) (1985) 63-75.
[7] K.M. Chandy and J. Misra, A paradigm for detecting quiescent properties of distributed computations, Rept. No. TR-85-02, Dept. Computer Science, Univ. of Texas at Austin, Austin, TX (1985).
[8] K.M. Chandy and J. Misra, How processes learn, *Distr. Comput.* **1** (1986) 40-52.
[9] E.W. Dijkstra, The distributed snapshot algorithm of K.M. Chandy and L. Lamport, Letter No. EWD864a (1983).
[10] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations, *Inform. Process. Lett.* **16** (1983) 217-219.
[11] E.W. Dijkstra and C.S. Scholten, Termination detection for diffusing computations, *Inform. Process. Lett.* **11** (1980) 1-4.
[12] R. Fagin, J.Y. Halpern and M.Y. Vardi, A model-theoretic analysis of knowledge: extended abstract, *Proc. Symp. Foundation of Computer Science* (1984) 268-278.
[13] N. Francez, Distributed termination, *ACM Trans. Programm. Lang. Systems* **2** (1980) 42-55.
[14] N. Francez and M. Rodeh, Achieving distributed termination without freezing, *IEEE Trans. Soft. Eng.* **SE-8** (1982) 287-292.
[15] N. Francez, M. Rodeh and M. Sintzoff, Distributed termination with interval assertion, in: J. Diaz and I. Ramos, eds., *Proc. Internat. Coll. on Formalization of Programming Concepts*, Peniscola, Spain, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981).
[16] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.
[17] T. Herman and K.M. Chandy, A distributed algorithm to detect and/or deadlock, Rept. No. TR LCS-8301, Dept. of Computer Science, Univ. of Texas, Austin, TX (1983).
[18] J. Halpern and Y. Moses, Knowledge and common knowledge in a distributed environment, *Proc. 3rd Symp. on Principles of Distributed Computing* (1984) 50-61.
[19] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Comm. ACM* **21** (1978) 558-565.
[20] L. Lamport, Lecture notes prepared for the Advanced Course on Distributed Systems—Methods and Tools for Specification, Computer Science Institute, Technische Universität München, Munich, Germany (1984).
[21] D. Lehmann, Knowledge, common knowledge and related puzzles (extended summary), *Proc. 3rd Symp. on Principles of Distributed Computing* (1984) 62-67.
[22] G. Le Lann, Distributed systems—towards a formal approach, in: B. Gilchrist, ed., *Information Processing 77* (North-Holland, Amsterdam, 1977) 155-160.
[23] C. Morgan, Global and logical time in distributed systems, *Inform. Process. Lett.* **20** (1985) 189-194.
[24] J. Misra and K.M. Chandy, Termination detection of diffusing computations in communicating sequential processes, *ACM Trans. Programm. Lang. Systems* **4** (1982) 37-42.
[25] G. Plotkin, An operational semantics for CSP, in: D. Bjørner, ed., *Formal Description of Programming Concepts, IFIP TC-2 Working Conference*, Garmish-Partenkirchen, Germany (1982) 185-208.
[26] S.P. Rana, A distributed solution to the distributed termination problem, *Inform. Process. Lett.* **17** (1983) 43-46.
[27] W. Reisig, Partial order semantics for CSP-like languages and its impact on fairness, in: J. Paradaens, ed., *Proc. Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science **172** (Springer, Berlin, 1984) 403-413.
[28] W. Reisig, *An Introduction to Petri Nets*, Monography EATCS (Springer, Berlin, 1985).
[29] J.L. Richier, Distributed termination in CSP-symmetric solution with minimal storage, Rept. No. 84-49, LITP, Univ. Paris 7, Paris (1984).
[30] F.B. Schneider, Lecture notes prepared for the Advanced Course on Distributed Systems—Methods and Tools for Specification, Institute for Computer Science, Technische Universität München, Munich, Germany (1984).
[31] N. Shavit and N. Francez, Efficient detection of locally-indicative stable properties, *Proc. 13th Internat. Coll. on Automata, Languages and Programming*, Rennes, France (1986).