The Wave Algorithms as a Self-Stabilizing Control Structure

L. Duchien[1], G. Florin[1], R. Gómez[2], L. Seinturier[1]

October 7, 1996

Abstract

Wave algorithms are presented in many papers as solution for distributed problems. On the other hand, there are numerous research teams working on self-stabilizing systems e.g. a system that, regardless of their initial state, must reach a legitimate state. In this paper we show how the wave scheme can be transformed into self-stabilizing wave and hence wave algorithms can be adapted into self-stabilizing wave algorithms.

Key-words: *distributed algorithms, self-stabilizing, wave algorithms, object-oriented approach, fault-tolerance*

CEDRIC [1] - CNAM

292 rue St Martin, Fr 75141 Paris Cedex 03

e-mail: Laurence.Duchien@cnam.fr, Gerard.Florin@cnam.fr, Lionel.Seinturier@cnam.fr

Tel: +33 (1) 40 27 25 83

♣

ITESM-CEM [2] - Dpto. Ciencias Computacionales

Apdo. Postal 2 Módulo de Servicio Postal,

Atizápan Zaragoza, 52296 Edo. de México

e-mail: rogomez@campus.cem.itesm.mx

Tel: +52 (5) 326 55 38

1 Introduction

A self-stabilizing solution was proposed for the first time by Dijkstra as a solution for a distributed election in a ring [9]. A self-stabilizing algorithm generalizes this proposition. It can be executed in an arbitary initial state, eventually it reaches a legitimate state, and stays on it. The wave algorithms are based on a so called *diffusing computation* idea. Introduced in several solutions in the seventies, presented as a general algorithm for the first time by Chang [7], they have been used in many distributed algorithms (see [19, 18, 11, 20]).

In this paper our approach is to establish that the wave concept can be used as a self-stabilizing control structure and to present a simple example. For this purpose the paper is divided as follows; the second section presents the characteristics of wave algorithms. The third one shows the principal features of a self-stabilizing system. The fourth section defines a self-stabilizing wave. We present a simple example of self-stabilizing wave to collect a global state. The last section presents our conclusions and future works.

2 The wave algorithms

One of the pattern frequently used to design distributed algorithms is the wave control structure. In Chang's paper [7], he presented the characteristics of what he called *echo* algorithms. Helary and Raynal presented the wave algorithms as a synchronization tool and as a control structure [18]. Later, Florin and al [11] introduced the recursive wave as a control design scheme based on recursivity and concurrent remote procedure call mechanisms. In [4], we find an object-oriented version of the recursive wave.

The wave concept is based on the paradigm of *diffusing computation*, that is, a distributed computation initiated by a single process called initiator, and joined by others only after receiving a message and triggered by some condition internal to the process [20]. A wave consists of two phases: a *descending* phase, in which the wave visits all the sites of the network, and a *backtracking* phase that collects an information from the visited sites. This collected information can be for instance termination indicators or simple computation results. This scheme can be used for some classical distributed problems such as broadcast, synchronization, and computing global functions.

We just briefly recall the main principles of recursive waves. In the *descending* phase a site initiates the wave sending a procedure call to its neighbours. These neighbour sites forward these calls to their neighbours and so on, until a stop condition is checked in some site(s). Once the stop condition is true the *backtracking* phase starts. Then, every site returns an answer to the site that activates it, until the wave comes back to the initiator site. When the wave returns to the initial site the wave computation is finished. Nevertheless the initial site can start a new wave that follows the same or a different path. This last option is called by Raynal *wave train* [18]. In a wave algorithm there can be one or several initiators sites. An algorithm with a single initiator is called a centralized wave. Once the algorithm can be started spontaneously by an arbitrary subset of processes, it is called a decentralized wave. Gómez [12] deals with this last case. Two or more recursive waves are executed concurrently in order to build only one structure.

3 The self-stabilizing algorithms

Proposed for the first time by Dijkstra in 1974 [9], a self-stabilizing system is such that, regardless of its initial state, (legitimate or illegitimate), eventually reach a legitimate state. These systems can be defined as correct processes that may behave transiently inconsistant but that guarantee a return to a correct behaviour within a finite time after the fault ceases. A self-stabilizing system protects for instance against the transient failures such as memory failures. This property is also very useful for systems composed of processors that may crash and recover in an arbitrary state. Due to the previous features self-stabilizing algorithms must behave according to the following rules:

1. an algorithm has no specific initial state,

2. it must be fault tolerant,

3. there is no termination procedure.

The first property means that an algorithm can be started at any time, in an arbitrary state, any site can initiate the processus, and yet eventually coordinated behaviour leading to a legitimate state is guaranteed. The second one requires full and automatic protection against transient process failures because the algorithm may recover from any configuration. As a self-stabilizing algorithm must be constantly and undefinitly maintained a legitimate state it never ends. Hence it does not have a termination procedure. So we never can observe that a legimitate configuration has been reached.

Self-stabilizing algorithms often use a particular communication scheme called *link register* communication. Each communicating process pair shares a register such that one of the process is always writer and the other is always reader. The most widely used communication mode is the classical message passing mode used in loosely coupled process networks. In [14] a method to create a self-stabilizing version of a message passing algorithm is proposed. The idea is to establish a global property P that will be verified by all the sites. The property is verified using a snapshot algorithm proposed by Chandy and Lamport [6]. If the property is false the algorithm is executed again by the site that initiates the snapshot. In the opposite the site continues its previous work, until it decides to execute another snapshot. In [1] another method to create a self-stabilizing version is based on local checking. The paper [5] proposes a general scheme based on token passing.

We propose to use the wave algorithms to design a distributed solution to build and to keep a distributed structure. The solution is transformed into a self-stabilizing wave. As far as we know no author presented such a systematic transformation with an object-oriented point of view. Details are presented in the next section.

4 Wave algorithms and self-stabilizing systems

4.1 Computational model

This paper uses a distributed object oriented approach. In these systems a collection of objects located on different sites interacts synchronously with remote method calls or asynchronously with message passing communications. We do assume that every message is eventually received and that every remote method call is eventually processed and eventually returns results.

Each objet can access to other object public methods (including local or remote objects). Objects can be dynamically created or removed. In order to represent the futur behaviour of the system (methods that will be executed) we may use the Temporal Logic of Actions (TLA) [15]. To express methods or data that can be accessed in the current environment as a result of the previous executions, we may use a knowledge logic [10].

With the Temporal Logic of Actions [15], each action act is associated with a predicate act that is true if act is executed in the current site. $\Diamond act$ means that act will eventually be true (will be executed) and $\Box act$ means that act is always true.

In the knowledge based logic used in this paper we express with an operator K that an object x exists in the environment of an object i. So i can access any method offered by x in the current state. The meaning of the knowledge operator can be precised for an object x that is a classic variable instance of a class C (e.g. a boolean, an integer). The class implements methods such as *create, read, write, invalidate, remove.* Roughly speaking, $i.K(x.read(v))$ means that an object i can read correctly the value of the variable v encapsulated by the object x. More precisely, the correctness of a read operation can be defined by a predicate: x has been created and, at least, once written since the last write operation and, was not invalidated or removed. The right value is the value written by the last write operation.

Using these logic tools, we can express invariants on local variables defined on each site. The sequel of this section presents two kinds of invariants that must be used in self-stabilizing solution. In a self-stabilizing protocol we assume that each object holds two kinds of local variables. The first kind of object assumes that values and codes are correctly initialized and cannot be corrupted. Hence a set of invariants is always maintained. For example, we assume that each object includes an object identifier that is unique in the global system. This object identifier is called *ego* in each objet. We can write that :

1. It is always true that a given site i can read its identifier value. The read method accesses to the variable v. K denotes that we can access to this variable : $\forall i \in Sites, \Box(i.K(ego.read(v)))$. *ego* can be defined itself as an object encapsulating an elementary variable v that stores the identifier.

2. The identifier does not change: $\forall i \in Sites, \Box([i.ego.read(v1) \Rightarrow \Diamond i.ego.read(v2)] \Rightarrow v1 = v2)$

   The difference between the notation $i.ego.read(v)$ and $i.K(ego.read(v))$ comes in first case we are sure of the object *ego* is intanciated. In the second case, we just may say that the object exists and it can be reached.

3. It is always true that, if a site i processes a read operation on its own identifier *ego*, it obtains a different value that another site j.
   $\forall i, j \in Sites, i \neq j, \Box([i.ego.read(v1) \wedge j.ego.read(v2)] \Rightarrow v1 \neq v2)$

This first kind of object assumption is only convenient to focus on a particular problem solved by a given protocol. If a protocol needs a unique identifier to correctly perform its task, this identifier must be initialized and maintained by a correct naming protocol.

In self-stabilizing solutions there are variables that can be incorrectly initialized. Then we consider a second kind of objects. They are associated with values that can be asynchronously modified during a given initial period. So a given invariant cannot be always but only eventually verified. Let consider an object as the neighbour set of a given site in a network topology: we may assume that this object is asynchronously modified by write operations performed by a failure detector. Hence the neighbour set is not constant during the whole life of the system. Nevertheless a self-stabilizing protocol must converge if there exists a time such that a failure never occurs after. Beyond this date the following properties are verified:

1. A given site i knows its neighbourhood: $\forall i, \Box(i.K(i.neighbours.read(v)))$

2. The neighbour set is stabilized:
   $\forall i, \Diamond(i.neighbour.read(v1) \wedge i.neighbour.read(v2) \Rightarrow v1 = v2)$

We also consider in our model that the communication links are symmetrical i.e. a site i that calls a site j necessary can be called by this site.
$\forall i, j \in sites, i \neq j, \Box([j.neighbour.read(v1) \wedge i.neighbour.read(v2)] \Rightarrow [j \in v2 \wedge i \in v1])$
Let just recall that a fault tolerant behaviour must include :

- *a failure detector:* this component must check the system to decide whether it is correct or not.
- *a failure extension assertion:* this component must decide (when a failure is detected) that different components are suspected to be also faulty.
- *a recovery strategy:* this component must decide what are the steps that must be performed to recover the failed components.

4.2 The recursive wave

The recursive wave was presented in [11] as a control structure to design distributed algorithms. It is based on the remote procedure call mechanisms (RPC), the concurrency theory and the recursivity concept.

In a wave algorithm a network of sites builds and maintains a data structure represented by the values of the local variables defined in each site. A site considers to be in legitimate state if an invariant property p_i is verified (i.e. p_i is valid for the site i).

From the previous informal description it appears that a recursive wave is a distributed programming scheme defined by a recursive procedure that calls in parallel n instances of itself on n different sites. So the programming scheme is the following:

```
METHOD Recursive_Wave() IS
BEGIN
    IF (condition) THEN
        PAR i IN Neighbours DO
            i.Recursive_Wave();
        ENPAR;
    ENDIF;
END Recursive_Wave;
```

Executing a recursive wave leads to a tree where the root represents the first execution of the procedure (i.e. the initiator of the computation), the leaves correspond to the active execution of the procedure, and the rest of the nodes are the blocked procedures, waiting for the termination of the parallel recursive calls.

The presented scheme considers concurrent executions of n procedures. Therefore a programming control structure must be available for the parallel execution of n threads on the same processor. We assume that the following language constructor **PAR**, similar to OCCAM, can be used.

```
PAR i IN <domain> DO
    < INSTRUCTION BLOCK >;
ENDPAR;
```

For each parameter value in a discrete set (<domain>), a thread associated with the instruction block is started on the processor executing the PAR instruction. This instruction is terminated when all the activated threads are finished and the next instruction is executed.

From the previous scheme it can be derived many specified waves (e.g. sequential, spanning tree, traversal algorithms) solving many differents distributed problems. For this paper we must choose one kind of recursive wave and we take the example of the flooding recursive wave. This recursive wave is specified in Figure 1. We use a passive distributed object-oriented language, close to the GUIDE language [2]. We also use control clauses and event counters to express the concurrency control. Each visited object calls in parallel all of its neighbours. PAR instruction implements a group remote procedure call [8]. This group invocation is performed only once, when the object is visited for the first time. The *Test_And_Set* method allows to know if a call is the first one or not. This method is executed in mutual exclusion, as it is pointed out by the control clause. This calling scheme limits the flooding of the wave to only one call in each direction in every logical link between two objects.

The program in Figure 1 is very simplified in order to be short and understandable. Nevertheless it records the identifier of the calling object in the *Parent* variable, and it does not call back its own parent.

The program also records in each object (variable *Local_State*) the state of the wave. The wave separates the past from the future in each site. There are three states:

1. *sleeping state:* it is the initial state of the site, (i.e. the state before the wave begins),
2. *visiting state:* it is the state reached when the wave arrives for the first time,
3. *backtracking state:* a site is in backtracking state once it receives the answer of all its neighbours. The state is reached when the wave backtracks, and leaves the considered object.

The given solution can support only one initiator and only one wave at a time. Many others facilities can be added as a wave identifier allowing several waves to be executed at the same time and an epoch number necessary to manage successive executions of the wave solution [3].

4.3 The self-stabilizing wave

In this section we propose a general scheme to create self-stabilizing algorithms in accordance with the previous scheme. We propose a simple example of using this control structure to collect a global state.

```
TYPE Frw IS
    METHOD Flooding_Recursive_Wave(Calling);
    METHOD Test_and_Set(First_Visit, Calling);
END Frw;
CLASS Frw_Implement IMPLEMENTS Frw IS
TYPE Frw_State IS (Sleeping, Visiting, Backtracking); – Possible site state
VARIABLES
    Ego: REF Frw; – Current site identifier
    Neighbours: SET OF REF Frw; – Neighbours set
    Local_State: Frw_State := Sleeping;
    Parent: REF Frw;
METHOD Test_And_Set
    (IN_OUT First_Visit : BOOLEAN, IN Calling : REF Frw)
BEGIN
    IF (Local_State = Sleeping) THEN
        Local_State:= Visiting;
        First_Visit:= TRUE;
        Parent:= Calling;
    ENDIF
END Test-and-Set;
METHOD Flooding_Recursive_Wave(IN Calling: REF Frw)
    First_Visit: BOOLEAN = FALSE;
    i: Ref_Frw;
BEGIN
    Test_And_Set(First_Visit, Calling);
    IF (First_Visit) THEN
        PAR i IN (Neighbours - Parent) DO
            i.Flooding_Recursive_Wave(Ego);
        ENDPAR
    ENDIF
    Local_State:= Backtracking;
END Flooding_Recursive_Wave;
    CONTROL current(Test_and_Set) = 0;
END Frw_Implement;
```

Figure 1: *General description of a flooding recursive wave*

4.3.1 Introduction to a self-stabilizing wave control structure

The scheme was first proposed using message passing communication in the ARPA network and it is used in the most popular routing algorithms implemented for instance on the Internet network, (RIP and BGP protocols [17, 13, 16]). In an infinite loop each node sends its routing table to its neighbours and receives the routing table from its neighbours. Then it computes its new table from all the received routing tables.

A self-stabilizing version of the recursive wave is mainly defined from the combination of a **WHILE** instruction and a group remote procedure call invocation. The programming scheme is the following:

```
WHILE < condition > DO
    PAR i IN Neighbours DO
        i.Neighbour_Exchange();
    ENDPAR;
ENDWHILE;
```

The loop is processed while the condition is checked to be true. In a self-stabilizing with no termination detection, the loop must be performed for ever. The purpose of this scheme is to ensure that a stabilizing property is eventually reached. A standard property is *after a period, a state is reached.* In a self-stabilizing recursive wave the state is computed from the state of the neighbourhood. The remote method call *i.Neighbour_Exchange* sends the state of the node to the neighbour i and receives its state. Using this result, the node state is updated. With a self-stabilizing approach, the loop is infinite to tolerate transient failures and to accept the neighbourhood variations for instance due to the variations of the network topology.

4.3.2 A self-stabilizing wave algorithm to collect a global state

An example of a self-stabilizing wave using the previous design pattern is given in figures 2 and 3. As it is impossible to give a complex example it is simply designed to collect a global state. The property P that we want to verify with this self-stabilizing algorithm is: *after a period, the root learns that the wave has reached all the sites.* They transmit data collected by the wave and the initiator of the wave collects the state of the structure built with the self-stabilizing approach. The code gives a basic version of a flooding self-stabilizing wave. It is developed in order to show the global behaviour of flooding wave in a self-stabilizing approach.

The algorithm is started by the initiator that performs a call to the *Neighbour_Exchange* method on itself with its own identifier for the *Calling* value and a *Depth* value equal to zero. This *Depth* parameter is added to maintain on each visited site its distance to the initiator site in number of hops. The *Neighbour_Exchange* method uses an asynchronous method call to *Ssfrw_Loop* to

```
TYPE Ssfrw IS
    METHOD Ssfrw_Loop();
    METHOD Neighbour_Exchange();
    METHOD Test_And_Set();
END Ssfrw
CLASS Ssfrw_Implement IMPLEMENTS Ssfrw IS
TYPE Ssfrw_State IS (Sleeping, Visiting, Backtracking); – Possible site state
VARIABLES
    Ego: REF Ssfrw; – Current site identifier
    Neighbours: SET OF REF Ssfrw; – Neighbours set
    Local_State: Ssfrw_State := Sleeping;
    Parent: REF Ssfrw;
    Depth : INTEGER;
    First_Visit: BOOLEAN:=FALSE;
METHOD Ssfrw_Loop() IS
    Neighbour_State: ARRAY OF Ssfrw_State;
    Neighbour_Depth: ARRAY OF INTEGER;
    i : REF Ssfrw;
BEGIN
    WHILE true DO
        PAR i IN Neighbours DO
            i.Neighbour_Exchange
                (Neighbour_State(i),Neighbour_Depth(i), Ego, Depth);
        ENDPAR
        FOR i IN (Neighbours - Parent) DO
            IF (Neighbour_State(i) = Visiting) ∧
                    (Neighbour_Depth(i) > Depth) THEN
                Local_State:= Visiting;
                GOTO Known_State;
            ENDIF;
        ENDDO;
        Local_State := Backtracking;
        Known_State:
    ENDWHILE;
END Ssfrw_Loop;
```

Figure 2: *General scheme of a self-stabilizing wave*

start the self-stabilizing control structure. The loop is presented as a simple WHILE loop. The loop can be also timed in order to synchronize on the basis of a given period. For the other sites, a site is visited when it is reached for the first time by the wave by the first execution of the *Neighbour_Exchange* method. This method is implemented on each site. It uses two kinds of parameters: input parameters from the calling object and answer parameters from the invoked object (i.e. callee) to point out its state and its depth.

In this way we can see in the program that the *Local_State* variable records the evolution of the state of the wave also managed by the flooding recursive wave (i.e. *Sleeping, Visiting, or Backtracking*). A site becomes *Backtracking* when all its neighbours with a depth greater than its own depth are in *Backtracking* state or when it has no neighbour with a depth greater than its own depth (i.e. it is a leaf).

4.3.3 Self-stabilizing aspects

Three self-stabilization aspects may be considered. First the algorithm must reach a stable state regardless of the initial state. Secondly, a self-stabilizing algorithm offers a protection against transient failures. Lastly, the network topology is a dynamic structure.

In a normal statement, the processes start in *Sleeping* state and reach the *Backtracking* state. But the self-stabilizing features indicate that the consistent initialization of an algorithm can be eliminated. In transient failure on data, the state can have any value at the next starting. Indeed in the next *Neighbour_Exchange* method call the state value will be updated. The state can go from *Sleeping* to *Visiting* value by the next *Neighbour_Exchange* call, and then in normal statement it becomes in *Backtracking* state. If, after the failure, the state is *Visiting*, the next *Neighbour_Exchange* method call allows to update the *State* variable to *Visiting* or *Backtracking*. Lastly, if, after a failure, the state is *Backtracking* and its neighbourhood is in *Visiting* state, the next *Neighbour_Exchange* method call allows to change it in *Visiting* state. Indeed the solution can consider recoveries. A a site can be again added and updated in the structure by its neighbours.

As in the self-stabilizing approach, we assume that uncontrolled variations on topology variables are transient. Each object performs an infinite loop of group remote procedure calls on its set of neighbours. To be actually self-stabilizing the group remote procedure call must be correctly specified. So we can assume that RPC can fail but in this case a neighbour failure must be detected. Then the neighbour set is modified. In fact the failure detector and the RPC communication protocol must be related and consistant. So the algorithm converges to a new solution after the occurrence of a change in the topology.

The network topology is a dynamic structure, we assume that links and sites can go down and come up an arbitrary number of times, and at any time. We assume that the neighbour set is set

```
METHOD Test_And_Set
    (First_Visit, Calling, Depth_Calling) IS
BEGIN
    IF (Local_State = Sleeping) THEN
        First_Visit:= TRUE;
        Parent:= Calling;
        Local_State:= Visiting;
        Depth:=Depth_Calling +1;
    ELSE
        First_Visit:=FALSE;
    ENDIF
END Test_And_Set;
METHOD Neighbour_Exchange
    (Reply_State, Reply_Depth, Calling, Depth_Calling) IS
BEGIN
    SELF.Test_And_Set(First_Visit, Calling, Depth_Calling);
    IF (First_Visit) THEN
        !A.SELF.Ssfrw_Loop(); -- Asynchronous invocation
    ENDIF;
    Reply_State:= Local_State;
    Reply_Depth:= Depth;
END Neighbour_Exchange;
    CONTROL current(Test_And_Set) = 0;
END Ssrfrw_Implement;
```

Figure 3: *Methods description used in the self-stabilizing wave*

up by a local failure detector mechanism depending on the state of links and sites.

4.3.4 Correctness proofs

We give hence a brief introduction to correctness proof of the solution.

1. Each site is visitied by the construction wave (as a given site calls again and again its neighbours).

2. All sites becomes in backtracking state (if no failure occurs (following the algorithm construction).

3. After a time period, the algorithm becomes stable. The state of site i, denoted by σ_i, is a value in the enumerated type (*Sleeping, Visiting, Backtracking*).

   Property: $\forall i \in Sites, \Box(\sigma_i = Sleeping \Rightarrow \Diamond\sigma_i = Backtracking)$

   Each site can read the state of its neighbour. By this assumption: A given site i knows its neighbourhood: $\forall i, \Box(i.K(Neighbour.read(v))$.

   We can say that: $\forall i \in Sites, \forall j \in Neighbour_i, i.K(j.read(\sigma_j))$.

   Each site computes its new state at each loop. The state of a site changes from *Sleeping* to $Visiting$ value once the site is visited the first time. Again, the state changes from $Visiting$ to *Backtracking* value once the neighbour set minus its Parent is in *Backtracking* state with a depth greater than its own depth or when it has no neighbour with depth greater than its own.

   $\forall i \in Sites, \Box((\forall j \in (Neighbour_i - Parent_i), \sigma_j = Backtracking \wedge j.depth > i.depth)$
   $\vee\ (\nexists j, (j \in (Neighbour_i - Parent_i) \wedge j.depth > i.depth)) \Rightarrow \Diamond(\sigma_i = Backtracking)$

   By the assumption : After a beginning periodhe neighbour set may be constant in a period: $\forall i, \Diamond(i.Neighbour.read(v1) \wedge i.Neighbour.read(v2) \Rightarrow v1 = v2)$

   Once the neighbour set is stable, the site state will be stable to the *Backtracking* state.

We also must prove that, if a new site is added in the structure or becomes again after a transient failure, its neighbourhood will change their state in their next *Neighbour_Exchange* method call with this new site. Their state value could become from *Backtracking* to $Visiting$. Once the new state has its neighbour set in *Backtracking* state, it becomes in *Backtracking* state and the algorithm can again converge in *Backtracking* state. To do this proof, the possibility of loosing the network connexity must be considered. To develop this proof requires space that we do not have.

5 Conclusion

This paper introduces two distributed algorithms ideas. First, we extend the wave algorithm to a new control structure: the self- stabilizing wave. Then we give the solution in an object-oriented programming approach. As far as we know these twoideas are new.

This work defines the generic control structure for the computation of a global state in a network with transient failures. The generic control structure can also be used with other wave algorithms with a self-stabilizing approach. As future works we will develop an algorithm to build a spanning tree with this structure. We also will define completely the data fault detectors to consider link and process failures and to keep the tree structure.

A last remark is to emphasize on the interest to use an object-oriented programing model to develop these algorithms. The implementation is made much easier by this approach. One of our other work is to develop a complete object process from the specification to the implementation[4].

References

[1] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S.Dolev. Self-stabilization by local checking and global reset. In *Proceedings of WDAG*. LNCS 857, Springer-Verlag, 1994.

[2] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, P. L. Dot, M. Meysembourg, H. Nguyen, E. Paire, M. Riveill, C. Roisin, X. R. de Pina, R. Scioville, and G. Vandome. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[3] L. Bonnet. Spécification et preuves d'algorithmes distribués en approche orientée objet. Master's thesis, CNAM-Cedric, Dec. 1994.

[4] L. Bonnet, L. Duchien, G. Florin, and L. Seinturier. Some specification steps of a spanning tree algorithm with an object-oriented approach. In *Proceedings of the 1st IFIP Workshop on Formal Mehtods for Open Object-Based Distributed Systems (FMOODS'96)*, Paris, France, Mar. 1996.

[5] G. Brown, M. Gouda, and C.-L. Wu. Token systems that self-stabilize. *IEEE Trans. on Computers*, 38(6), June 1989.

[6] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[7] E. J.-H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engeneering*, 8(4):391–401, July 1982.

[8] C. Coste, L. Duchien, G. Florin, and L. Seinturier. Différentes approches des relations d'ordre dans un environnement coopératif. In *Actes des Journées de Recherche sur le Contrôle Réparti dans les Applications Coopératives (CRAC'96)*, pages 17–22, May 1996.

[9] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[10] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasonning About Knowledge.* MIT Press, Cambridge, Massachusetts, 1995.

[11] G. Florin, R. Gómez, and I. Lavallée. Recursive distributed programming schemes. In *Proceedings of the ISADS*, Kawasaki, Japon, May 1993.

[12] R. Gómez-Cardenas. *Conception et spécification d'algorithmes distribués : la vague récursive.* Thèse de doctorat, Université Paris VIII, Apr. 1995.

[13] C. Hedricks. Routing information protocol. RFC 1058, June 1988.

[14] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 91–101, Quebec, Canada, 1990.

[15] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, pages 872–923, May 1994.

[16] K. Lougheed and Y. Rekhter. A border gateway protocol 3 (bgp-3). RFC 1267, Oct. 1991.

[17] J. McQuillan and al. The new routing algorithm for the arpanet. *IEEE Transactions on Communication*, May 1980.

[18] M. Raynal and J. Helary. *Synchronisation et contrôle des systèmes et des programmes répartis.* Eyrolles, Paris, France, 1988.

[19] A. Segall. Distributed network protocols. *IEEE Trans. Inform. Theroy*, 29:23–35, 1983.

[20] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

Contents