ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Antony Bryant and Lesley Semmens (Eds)

Methods Integration

Proceedings of the Methods Integration Workshop, Leeds, 25-26 March 1996

Paper:

A Spanning Tree Object-Oriented Distributed Algorithm: Specification and Proof

L. Bonnet, L. Duchien, G. Florin and L. Seinturier

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

A Spanning Tree Object-Oriented Distributed Algorithm: Specification and Proof

L. Bonnet, L. Duchien, G. Florin, L. Seinturier

CNAM-Laboratoire CEDRIC
292, Rue St Martin, FR-75141 Paris Cedex 03
e-mail : {bonnet, duchien, florin, seintur}@cnam.fr

Abstract

This paper presents a way to specify, to prove, and to implement object-oriented distributed cooperative algorithms. Our method gathers together three kinds of tools or formal concepts. It is derived from the B Method. So it uses a refinement approach and a proof of each specification step. The second set of concepts comes from distributed artificial intelligence. One main aspect of parallel and distributed computation is the uncertainty of the knowledge state of other sites. To express it, an epistemic logic is used to define knowledge of sites. Finally we consider some interesting practical results of distributed algorithms research. The method defines an application from three points of view (group, object, and method levels). A particular state automata formalism is used for each specification step. Finally, a distributed spanning tree algorithm is developed.

1 Introduction

Interest in the object paradigm has been growing in the software world. This structuring of applications improves modularity, maintenance and program reusability. The success of the object approach is mainly in the sequential programming domain. Furthermore, there are several studies that design or extend object languages with parallel and concurrent control clauses (for example Concurrent Smalltalk, Parallel Eiffel, Guide). What is more, distributed operating systems have been designed in order to consider object technology (for example Arjuna, Amoeba, Cool, Emerald, Guide, SOS). Their main feature is to implement directly a performing multi-threaded, transparently distributed virtual machine that supports the object paradigm.

In parallel or in distributed programming the need for specifications and proofs is yet stronger than in the sequential domain due to the complexity of the design steps. Our purpose is to suggest a model of specifications and a method of proofs suited to distributed algorithms with an object-oriented approach. As far as we know, the design of distributed applications in object-oriented environments is a new and difficult task. The distributed objects normalisation propositions (CORBA [18], ODP [15]) do not really cover the domain of specifications and proofs. The object-oriented methodologies (for example [5] or [6]) do not deal with concurrent and distributed control or cover only a particular aspect (real time [2], distributed passive objects).

Two main features are introduced in our method. First, to express the concurrent control and the possible execution flows, we use a state/transition (condition/action) representation. This condition/action model has a particular semantics. It differs from the synchronised finite state automata used in network protocol modelling. Roughly speaking, we define each state by a predicate about the different objects used in a model. If necessary, we use epistemic modal logic predicates; that is to say, predicates on knowledge and beliefs of variables values. A transition is viewed as the description of a condition to check and an action to perform. As soon as a state is enabled (it is reached and its predicate is true) the executing engine can check the different transitions to be executed. Notice that such a model description can be considered as a particular temporal logic of actions model [16]. Such a model can also be seen as an abstract machine in the B Method [1]. So the proof technique for each abstract machine model can use, as in B, a set theory approach.

Second, the refinement notion allows one to transform an underspecified model into a more precise one. Refinement is a technique used to transform the mathematical model of a software system (its specification) into an executable one (its refinement). Refinement has been studied intensively for many years [8]. Each refinement step is part of a proof construction in the B Method [1]. In the distributed object context we divide the specification into three parts: group, object and method levels.

The highest level of specification, called group level, is intended to represent the group behaviours of concurrent and distributed sites or objects. To this end, these specifications describe the knowledge evolution of the group of entities. As the distributed algorithm is performed successfully, knowledge is acquired by the group of objects. The group specification describes the different phases of the evolution of the distributed knowledge. We use the knowledge operators defined by [12] to model the state predicates of the group specifications.

At the same time, the group behaviours are implemented by the concurrent actions of sites, objects, or agents. The object specification represents the behaviour of one member of the group. The evolution of the global variables

of the object and the synchronisation needed for its internal concurrence control are defined in the different refinement steps of object level.

Finally, specifications at the last level present the detailed description of the object methods. The purpose of this last level is to specify the internal mechanisms that realise the desired actual behaviour. Then a specification can be translated into a distributed object-oriented language and then executed.

Our target distributed object environment is the Guide system [3] that is an object-oriented distributed system defined by Grenoble University. In this paper we describe in Section 2 the main concepts that are used in our method. Section 3 presents the model of specifications. Section 4 defines the semantics and syntax of the model. In Section 5 we propose several techniques of refinement. Then, we introduce the proof techniques associated with the specification method in Section 6. An example using this methodology is presented in Section 7. It is a spanning tree construction algorithm by recursive waves. Only some significant steps are developed. Conclusion and perspectives are given in Section 8.

2 Sources: Formal Methods, Temporal Logic of Actions, Knowledge Predicates and Distributed Control Structures

To build our specification method we propose to gather together several kinds of tools or formal concepts, such as the B Method, temporal logic of actions, knowledge predicates, and distributed control structure.

2.1 The B Method

Among the many existing formal methods that can be used, we choose the B Method [1]. It is difficult to summarise the main features of this method in a few words. It mainly relies on refinement, logic and set theory. As it is defined for sequential algorithms, this method must be extended and modified to cope with the parallel and distributed context. We propose to add or to modify some features to consider this context.

The B Method is based on the abstract machine notion (similar to objects). The software is composed of several abstract machines. Each of them implements data and operations (objects and methods in the object world). Data are encapsulated in a machine and can only be accessed through operation calls (method invocation). The proof technique by assertions uses a set theory formalism. The refinement is made step by step. For each refinement step, the specification must be more precise. Each specification step is a part of a proof construction with this technique. The initial machine transforms itself in hybrid machines and so on, and finally, in a program.

To summarise we borrow the two ideas of specification and assertion proofs from the B Method. A state/transition diagram notation is substituted for the abstract machine notation (see Section 3). The logic operators are extended to modal logic operators.

2.2 Temporal Logic of Actions

The possible interleaving of operations in a distributed and concurrent control scheme must be formally defined and expressed in a user friendly way. We suggest to use state transition diagrams. So in our approach the temporal logic aspect is expressed using order relations between actions. We consider distributed algorithms where everything is action (for example instruction blocks, methods, threads). An action takes some time and has two major events : the begin and the end events.

In this object-oriented design context we must consider an interval logic as the underlying modal logic. We choose the Temporal Logic of Actions (abbreviated TLA)[16]. Lamport writes that an execution of an algorithm is thought of a sequence of steps. Each step produces a new state by eventually changing the values of one or several variables. Therefore, reasoning about algorithms requires reasoning about sequences of states. An algorithm is described by a TLA formula with operators such as *always* ($\Box$) or *eventually* ($\Diamond$). In such an universe $\Box A$ can be defined as A is always true (from time 0 to all time t). By the same way, $\Diamond A$ can be defined as there exists a time t such that A is true from this time. Then safety, liveness or fairness conditions can be specified. Furthermore the causal ordering of actions A and B can be defined as $(\Box(A \Rightarrow \Diamond B))$. Lamport proposes proving correctness properties using this temporal logic approach.

2.3 The Distributed Artificial Intelligence Aspects

One main aspect of parallel and distributed computation is the uncertainty about the knowledge of other sites (or agents) states. This is due either to transmission delays or to failures. Hence the performance or fault tolerance features must be correctly considered. In our method, an epistemic logic approach is used to define knowledge (and then belief) of sites [12]. It allows one to model environments of entities, knowledge levels and groups behaviours.

Halpern argues that the right way to understand distributed systems is by considering how messages modify the knowledge of sites. Knowledge is defined by a hierarchy of knowledge operators. Distributed knowledge is the

weakest level. The highest level is common knowledge. It is the communication between processes that increases the level of knowledge of the group.

The most important operator $K_i\phi$ means that the process i has the knowledge of a given fact ϕ . The distributed knowledge of ϕ is denoted $D_G\phi$. No process of a group G knows ϕ , but the group can deduce ϕ from its current knowledge. A simple example is a process i that knows ψ and another process j can deduce ϕ from ψ . The next hierarchical level is the ‘everybody knows’ level where all the processes of a group have the same knowledge. This level of knowledge is denoted $E_G\phi \triangleq \bigwedge_{i \in G} K_i\phi$

Halpern denotes $E^2\phi$ to express that all processes know that all know ϕ . Then, $E^k\phi$ means that all processes know that all know that ... that all know ϕ (k times).

$$E_G^k\phi \triangleq \begin{cases} E_G^1\phi \triangleq E_G\phi \\ E_G^{k+1}\phi \triangleq E_G(E_G^k\phi) & k \geq 1 \end{cases}$$

Finally, the highest state of knowledge, the common knowledge of the group is denoted $C_G\phi$.

By definition: $C_G\phi \triangleq E_G\phi \wedge E_G^2\phi \wedge \dots \wedge E_G^n\phi \wedge \dots$

2.4 Distributed Control Structures

Finally we are interested in some practical interesting results of distributed algorithms research. For example, order properties can be implemented either by timestamps or by directed traversals of network graphs. We also use the general control structures developed for distributed algorithms (e.g. phased protocols, group remote procedure calls [7], recursive distributed programming or wave algorithms [9, 10]). They can be powerfully gathered to facilitate the implementation of new solutions.

3 Model of Specification for Cooperative Systems

The distributed object-oriented paradigm helps designers to master the complexity of cooperative systems. We propose to observe a distributed algorithm from three points of view: the group of objects (a set of distributed entities involved in a distributed computation), objects (a local entity), and their methods (an action that can be performed).

Refinement transforms step by step an abstract model (in the remaining of the article we use invariably the terms specification and model) of a software system into an executable code. The group specifications describe the global states and behaviours. The object specifications provide the concurrent behaviour and the local variables of the object. The last level of specifications describes and analyses the blocks of code and their local variables.

It must be emphasized that, by our different steps of refinement, each model of specification inherits the behavioural and knowledge aspects from higher levels. For instance, when knowledge is defined in a group specification, the corresponding knowledge will be found in the object specification level.

3.1 Group Specifications

First, the group of objects specification level handles global states, global knowledge, and the global interactions between these objects to reach a given goal. So the highest level must define the chosen strategy to solve the problem e.g. the global view of the distributed system behaviour. The global interaction defines a collective action that corresponds to the coordination between the set of objects. The definition of this collective action can be complex. In distributed algorithms, we use distributed global control structures such as recursive waves, phased algorithms or group remote procedure calls. These different kinds of structure can be combined.

To represent the different states encountered by the distributed system, we use predicates that include universal and existential quantifiers over the group membership. For example, $\forall i \in \text{sites}, K_i(Id(i))$ defines the predicate ‘each member of the group “sites” knows its own identifier’. K_i is the knowledge predicate defined in Section 2.3 [12]. Such a predicate must be interpreted by the following informal sentence : “In order to run correctly in this state, each object must be able to access its identifier variable”.

3.2 Object Specifications

In our methodology an object is an autonomous entity with its own knowledge. It interacts with the other objects of its environment. For instance the local knowledge of an object is the set of variables managed and accessed by its methods. We also deduce some local variables from the global knowledge of the cooperative system. Using the group example $\forall i \in \text{sites}, K_i(Id(i))$, we can say that each object knows its own identifier and defines then a variable such as “site_identifier” to represent this knowledge.

The behaviour of an object defines different types of coordination such as synchronisation or scheduling. As we must consider the parallelism, the global data of an object must be studied from the concurrence control approach. Indeed, to know how the group behaves helps to design the behaviour of an object.

3.3 Method Specifications

The method level is the last step. It specifies the internal mechanisms of the object that characterise its behaviour. We must only use knowledge that can be managed by a single method. The local variables of a method can be accessed without restriction. The correct update of global variables is guaranteed by the concurrence control defined at the object level. Then the specification of the method provides basic algorithmic structures such as loops, tests, arithmetic operations, reads and writes of variables that can be straightforwardly translated into a programming language.

4 Semantics and Syntax of the Methodology

The aim of methodology is to improve the specification of distributed algorithms. So, in this section we define more precisely several syntax and semantics aspects of a model. One of the key features of our methodology is to use a common paradigm (a state-transition approach) for the three levels of specification. The same formalism is used to write model describing the behaviour of a group, an object, and a method.

Figure 1 presents the basic components of a specification in graphical representation. The model is composed of states and transitions. States are associated with a predicate. Transitions are edges between states and are labelled by a condition, an action, and a post-condition. Each model owns a predicate that describes its environment. The semantics of the model that is discussed in the next sub-sections supports concurrence. So the model must be clearly distinguished from the sequential finite state model used in message passing protocols.

Figure 1: Basic Components of the Model

4.1 Hypotheses about the environment

The hypotheses about the environment must be seen as program invariants e.g. a predicates that must be checked in order to run correctly the model. They are associated with believes. Indeed, these kinds of features are necessary to the correct execution but are most of the time hard to verify. According with this remark we should rather denote an invariant by $B(\text{predicate})$ [12]. It means that, if the predicate is verified, then a model can be checked and, if true, it can be correctly executed. An example is the assumption about the uniqueness of the identifiers in a distributed system. Each site knows its own identifier and there is no homonym. The status of such a hypothesis is to be a belief. It is considered as a true property but it is not really checked. Hence in an invariant a belief appears to be a predicate always true, but never verified.

4.2 States

A state represents a significant point in the run of a behaviour. It is characterised by a Boolean valued predicate. A state is enabled or disabled. It is defined to be enabled (or assessable) when reached along a sequence of states starting from the initial state of a behaviour. The aspect describes the precedence rules of distributed phased algorithms or sequential programs. Once a state is enabled its consequent transitions can be checked. Therefore several states can be

enabled simultaneously, so concurrent behaviours can be represented. A state is assessable as long as its predicate is still verified. If a designer wants several states to be enabled at the same time, the only constraint is that their state predicates do not conflict e.g. they just have to be true simultaneously. Each model (of the group, object, or method level) owns a particular state that is called the initial state and that is enabled at the creation of the group, the instantiation of the class, or the beginning of the method.

A state predicate is associated with two kinds of knowledge. The first kind of knowledge represents the evolution of the behavior. It is then represented by an enumerated variable that records the evolution. The second kind of the knowledge represents the state of local (objects included in the current defined object) or remote (objects located on a distant site) objects. In the latter case transmission delays or sites failures lead to a situation where the state predicate is a belief about the value of variables. According to this remark we should rather denote such a state predicate by $B(\text{predicate})$ [13]. In this case the predicate must be transformed into an executable assertion to guarantee the correct execution of the model.

4.3 Transitions

A transition delimits phases between two states in the execution of a behaviour. It is composed of three elements: a condition, an action and a post-condition. The pair condition/action defines a guarded command [8]. Starting from an initial state, a transition allows one to reach a consequent state by evaluating a condition and performing the associated action that leads to the result specified by the post-condition. Moreover if the post-condition does not conflict with the predicate of the initial state, then the initial state stays assessable.

The condition is a Boolean expression. Conditions are formula based on the traditional negation, conjunction and disjunction connectors. They use group, object and method variables (shared and local variables). An action can be seen as a predicate transformer. It changes the initial state-predicate into the predicate of the post-condition. An action can be the invocation of another action using various semantics of group communication (i.e. message passing, remote procedure call, group procedure call, ordered multicast, causal multicast), different operations as the read or the write of a variable, or any kind of algorithmic structure (i.e. tests, loops). The post-condition is a predicate related to an action. It specifies the result of the executed action. This is an enhancement of an initial state-predicate by the effect of an action.

5 Refinement Features

In our methodology we propose to specify a distributed algorithm by a hierarchy of models. We call refinement the process that consists in deriving a parent model to produce a child model creating by this way a kind of inheritance hierarchy of behavioural models. Our basic assumption about the refinement process is that it must reduce the space of solutions of the model (e.g. the behaviour that can satisfy the constraints specified by state predicates and transitions). To put it more precisely we assume that the set of solutions of a refined model must be a sub-set of the set of solutions of a parent model. The set of solutions is defined as the set of all correct implementations (behaviour compatible with the final specification).

Two situations need to be examined. We must validate a refinement between two models within a same level (group, object, or method) and between two models of different levels.

5.1 Refinement within a Same Specification Level

For every algorithm specification the first step of specification consists in defining the initial and final states. The action associated to this first transition is the algorithm itself. Then, the refinement process splits the action in some sub-actions. There are two ways of refining an action : we can split up it into several sub-actions or we can enhance the post-condition defining its results.

Figure 2: A Model to refine

Let us consider an action A (Figure 2) that we want to refine with two or more sub-actions. In Figure 3 and 4 A is refined into two sub-actions A1 and A2 that can be executed sequentially or concurrently. In the former situation A1 and A2 are executed sequentially. We introduce an intermediate state whose predicate is the post-condition of the first sub-action A1. In the latter situation the condition Cond is duplicated on each transition; A1 and A2 can be executed concurrently. In both cases, the predicates of the initial and of the final states (Pred1 and Pred2 respectively) are unchanged.

Figure 3: Sequential Composition of Sub Actions **Figure 4: Parallel Composition of Sub Actions**

The second way of refining an action is conducted by enhancing its post-condition. Let us consider an underspecified behaviour where an action is not yet designed. For instance, in Figure 5 the action “modify i” has no concrete algorithmic structure. This situation is rather common when the algorithm is complex. We do not master all its complexity at once. Most of the time we leave some underspecified parts and we assume they will be specified later. So these particular actions are only defined by their result (e.g. their post-condition). In Figure 5 the result of action modify i is: $i > 0 \wedge i < 10$. Then a refinement step consists in defining more precisely the post-condition: $i = 1$. Thus, $i = 1$ is a correct refinement of the post-condition.

Figure 5: Refinement of an Action by Enhancing the Post Condition

5.2 Refinement between Two Different Specification Levels

First, we consider the refinement from a group level to an object level. We are now trying to find practical rules that help users to realise the transition between group level and object level but there are many difficulties to give a general methodology. Mainly, this refinement depends on two distribution criteria.

We can distribute the entities that cooperate to solve the problem according to the physical reality. In this situation, each agent or site has the same behaviour. Section 7 develops an example of a spanning tree distributed algorithm where each site that takes part in the construction has the same code. More intuitively, when the distributed application is symmetric (e.g. the same code runs on each site of the environment with different initialisation parameters) then, only one model is needed to bridge the gap between the group and the object level.

The distribution of a group can be viewed with a competence approach. We split up the problem according to the modularity principle. Each object can perform a set of different actions. Let us consider a distributed object system whose common goal is to exchange information and where an object plays the role of a data base server. It appears that the model describing the common goal of the application needs to be refined into two distinct object level models: one for the server and one for the client. Each of these models refines a part of the global behaviour of the group.

6 Proof Technique

Proofs must be associated with specification models in order to assert user defined properties. A characteristic step is to prove a transition. The proof technique is the same for each specification level.

$H \wedge \text{Condition} \wedge \text{Action} \Rightarrow \text{Post-Condition}$

Hypotheses H are defined by the state predicates. Starting from an initial state of one model the proof consists in establishing that, given the hypotheses H , if the condition is verified and the action is processed, then the post-condition can be deduced. This proof generally implies to examine refinements of actions. Various theoretical results are applied to carry such proofs. This process can be seen as Hoare axiomatic [14] and its extensions for parallel algorithms [19, 11]. Furthermore we use techniques such as recursive proofs and classical results from fix point theory.

7 An Example of a Spanning Tree Construction Specification and Proofs

The spanning tree construction algorithm is a distributed algorithm where a set of processes interconnected through a connected network co-operates to get organised into a tree topology. The proposed solution is a simple example of a recursive wave algorithm [17]. A site that decides to begin the construction of the tree is called the initiator. He sends a request to his neighbours notifying them that they are his potential children (Figure 6). A potential child receives several propositions. In the solution he decides to accept the first one. Hence he knows who his parent is. Then each child spreads the wave to his neighbours minus his parent (Figure 7). If a request reaches a site that already knows his parent he rejects the proposition. At the end of the **construction wave** each site knows his parent and children. The initiator is now the root of a spanning tree. He is the only one to know that the construction is finished (Figure 8). Then to notify the end of the construction to all sites it spreads a **termination wave** using the constructed tree structure. If a site is reached by this final wave then he knows the root of the tree. Moreover, his potential parent and children become his definite parent and children.

Figure 6

Figure 7

Figure 8

Figure 9

Spanning tree construction
algorithm with two initiators $I1 > I2$
 S belongs to $I1$'s construction,
 S receives a request from $I2$,
 S returns the request

Figure 10

Spanning tree construction
algorithm with two initiators $I1 < I2$,
 S belongs to $I1$'s construction,
 S receives a request from $I2$,
 S records its new parent $I2$,
 S spreads the wave initiated by $I2$

Notice that a correct solution must deal with a situation where two or several sites decide concurrently to build a spanning tree. Each site has a distinct identifier. These identifiers are totally ordered. Each initiator starts a wave construction. When a site S that belongs to the spanning tree of an initiator I_1 receives a request from another initiator I_2 he compares the two identifiers. If I_2 identifier is lesser than I_1 's, the wave initiated by I_2 loses and S returns a wave-loser response to I_2 (Figure 9). If I_2 identifier is greater than I_1 's then S records I_2 as its new potential root and spreads the wave initialised by I_2 (Figure 10). Once an initiator receives a normal return statement he knows that he wins. All the other initiators must receive a wave-looses response. Then he can spread a **termination wave** throughout the network.

The following Network Hypotheses (NH for short) can be considered as pre-conditions that must be satisfied to correctly perform the algorithm. We denote by K_i the knowledge operator of a site i .

H1: The network is defined locally by the knowledge of the neighbourhoods:

$$\forall i \in \text{sites}, K_i(\text{neighbours}(\{i\}))$$

H2: The communication links are symmetric:

$$\forall i, j \in \text{sites}, j \in \text{neighbours}(\{i\}) \Rightarrow i \in \text{neighbours}(\{j\})$$

H3: The sites identifiers are totally ordered:

$$\forall i, j \in \text{sites}, \text{Id}(i) = \text{Id}(j) \vee \text{Id}(i) > \text{Id}(j) \vee \text{Id}(i) < \text{Id}(j)$$

H4: Naming: each site knows his identifier and his neighbours identifier:

$$\forall i \in \text{sites}, K_i(\text{Id}(i)) \wedge \forall j \in \text{neighbours}(\{i\}) K_i(\text{Id}(j))$$

H5: No homonyms:

$$\forall i, j \in \text{sites}, i \neq j \Rightarrow \text{Id}(i) \neq \text{Id}(j)$$

H6: Network connectivity:

$$\text{neighbours}^k(\{\text{choice}(\text{sites})\}) = \text{sites}$$

$\text{choice}()$ is the choice axiom that allows to take a particular element in a set

H7: No failures

The network topology is preserved until the end of the algorithm

This is the beginning of the full specification of the solution [4]. As it is very long and detailed we only give the first steps as an example of the method. Hence these steps include many details that are not used in the paper.

7.1 Group Level Specification #1 of the Spanning Tree

7.1.1 The First Level Specification

In a group level we are not interested in describing the evolution of any particular site of the network. We must give a general understanding of the evolution of the group of sites. The goal of the algorithm is to organise a network satisfying the Network Hypotheses into a tree structure. Straightforwardly our first specification of the group level is:

Given the Network Hypotheses,
if some sites decide to start the algorithm
then there exists an action that allows the group to reach a state
where the spanning tree is built
and where each site knows the root, his parent and his children.

It is important to notice one key point of our model semantics: the edge between the *Begin* and *End* states does not necessarily represent only one thread of execution as in a traditional finite state automata. Indeed, one or several occurrences of the action "computing of a spanning tree called s_tree " can be witnessed simultaneously at the group level. At this step of refinement no indication on the interaction between these different occurrences is given. All that one can deduce is that each action occurrence leads to a state where the post-condition is verified:

$$\text{NH} \wedge \text{spanning_tree}(\text{network}, s_tree) \wedge E(\text{root}(s_tree)) \wedge \forall i \in \text{sites}, K_i(\text{parent}(\{i\})) \wedge K_i(\text{children}(\{i\}))$$

This predicate translates the distributed knowledge associated with a state where the distributed algorithm is finished. Here for this state:

- $\text{spanning_tree}(\text{network}, s_tree)$ translates that s_tree is a spanning tree of the network of sites. To put it more precisely with a standard graph notation: $\text{network} = (\text{sites}, \text{edges})$, $s_tree = (s_sites, s_edges)$

$$\text{spanning_tree}(\text{network}, s_tree) \Leftrightarrow$$

$$s_sites = \text{sites} \wedge s_edges \subseteq \text{edges} \wedge \text{connected}(s_tree) \wedge |s_sites| = |s_edges| + 1$$

- $E(\text{root}(s_tree))$: each site knows the root of the spanning tree.
- $\forall i \in \text{sites}, K_i(\text{parent}(\{i\})) \wedge K_i(\text{children}(\{i\}))$: each site knows his parent and children.

Figure 11: Group Level Specification #1

7.1.2 Comparison of our Method with the Temporal Logic Approach and the B Method

The transition results specified in Figure 11 can be described using the logical temporal operators $\Box$ and $\Diamond$.

$(\Box(NH \Rightarrow \Diamond(\text{spanning_tree}(\text{network}, s_tree) \wedge E(\text{root}(s_tree)) \wedge \forall i \in \text{sites}, K_i(\text{parent}(\{i\})) \wedge K_i(\text{children}(\{i\}))))$

By this way, we express the causal evolution of the algorithm when such a transition is fired. This is an abstract specification of the behaviour used in a proof step. Hence the specification of executed behaviour is $\Box(\text{condition} \Rightarrow \Diamond \text{action})$. The former predicate considers the hypotheses and the conclusion of the model, but doesn't describe the action. We only say that, at the time t , NH is verified, and, at the time $t+d$, we obtain the action result. By $\Box$, we precise that the algorithm computes the result.

The B Method is a formal specification method that emphasises on invariants and data. For example, the spanning tree algorithm specification includes a first abstract machine. We define the network by its sites and its links between the sites. The set site is denoted *site* and its cardinality n . The set of links is denoted by *link* and its cardinality by m . It is a binary relation between sites.

```

MACHINE
  Spanning_tree
CONSTRAINTS
  n ∈ NAT
VARIABLES
  sites, links, neighbours, root, initiators
INVARIANT
  sites ∈ 0..n
  links ∈ P(sites,sites)
  initiators ⊆ sites
  root ∈ sites
  parent ∈ sites
  children ∈ sites
  NH
DEFINITIONS
  neighbours ( sites)
  ran(sites ◁ links)
OPERATIONS
  Election
    THEN
      IF there exists one or several initiators
        THEN computing a spanning tree
    END
  END
END
END

```

The proof obligation in this abstract machine is that the invariant is preserved after the operation. The B method emphasises the operation definition and the invariant proof on these operations. With our model, we prove that we obtain the result. We are more interested in defining the result and its proof. Hence, the operations stay imprecise through the first steps of modelling and become more precise after few steps.

Once this first group level specification is defined we can introduce a first refinement. The chosen solution performs the construction of the spanning tree in two phases (e.g. a construction wave and a termination wave).

7.2 Group Level Specification #2

Let us give more details about specification #2. The **construction wave** and the **termination wave** are expressed using a new state definition *Construction finished* and two transitions. These transitions are associated with two edges respectively between the *Begin* and *Construction finished* states and between the *Construction finished* and *End* states.

Figure 12: Group Level Specification #2

Begin State : its predicate is " $\exists i \in \text{sites} \text{state}_i = \text{sleeping}$ ".

state = (.., state_j, ..) is a vector that represents the situation of the n sites.

In a cyclic spanning tree construction, each site can be in four states. So each component can take four values:

- sleeping if the site is not involved in a **construction wave** nor a **termination wave**
- construction if the **construction wave** is being processed
- constructed if the **construction wave** is finished
- terminated if the **termination wave** is finished

The new predicate is a refinement of the previous one. This predicate defines a situation where no site of the network is involved in any computing of a spanning tree. Indeed it is the conjunction of the old predicate and of the clause “ $\exists i \in \text{sites } \text{state}_i = \text{sleeping}$ ”. As no clause of the Network Hypotheses deals with the vector state, the two sub-expressions of the predicate are not in conflict. Hence for all the cases where the new predicate is true, the old one is also true. So the introduced process of refinement is correct.

Transition 1 : the new condition is the same as the old one. When a site decides to start the algorithm, the transition is fired and Action 1, which is the first sub-part of the refinement of the action of group level model #1, is processed. Post-condition 1 becomes true. Its interpretation is: once Action 1 is processed correctly the Network Hypotheses are still verified and the spanning tree of the network is constructed. But no site except the winning initiator knows it. Indeed every site waits for a potential better **construction wave**. The wave of the winning initiator flows back successfully. So he is the only one to know that the construction is finished. Furthermore, in this particular example, the predicate of *Begin State* becomes false so it is deactivated.

Construction finished State : its predicate defines the intermediate level of knowledge introduced in this new model. As its predicate is the same as the one of Post condition 1 it becomes active as soon as Post 1 is verified.

Transition 2 : according to Condition 2, the winning initiator is the only one to fire this transition. Action 2 is processed. Then, according to Post 2, the reached level of knowledge is the same as the reached one after the action of group level model #1. One element is added: “ $\forall i \in \text{sites } \text{state}_i = \text{terminated}$ ”. This addition does not introduce any conflict with the other parts of the old predicate. So it is a correct refinement. Its interpretation is:

- the Network Hypotheses are still verified and the spanning tree of the network is constructed,
- everyone knows that the algorithm is finished and his parent and children,
- everyone knows the root.

End State : its predicate becomes true as soon as Post condition 2 is verified.

Transition 3 : is added to specify that the algorithm can be restarted. As every program that computes a spanning tree many times can compute it at least once the new model is a correct refinement of group level model #1.

The refinement process of the group level specification can be carried on. For example we can introduce more details in the **construction wave**. From a theoretical point of view this process does not differ from the one presented above.

7.3 Group Level Specification #3

Each branch of the tree is composed of some nodes. These nodes are added one by one. As several branches perform this work, from a group level point of view the tree is constructed step by step. So we introduce the *Tree k_constructed* state to represent this mechanism. In Figure 13 the level of knowledge associated with this intermediate state is:

$$k_tree(k_sites, k_edges) \wedge \forall i \in k_sites, K_i(\text{parent}(\{i\})) \wedge \forall i \in k-1_sites, K_i(\text{children}(\{i\}))$$

$k_tree(k_sites, k_edges)$ is a predicate that expresses the construction advance of the spanning tree. k_tree is a sub-network of the global network: it is composed of the set of sites k_sites and of the set of edges k_edges ($k_sites \subseteq \text{sites}$ and $k_edges \subseteq \text{edges}$). As k_tree is also a tree, the following property is verified:

$$\text{connected}(k_tree) \wedge |k_sites| - 1 = |k_edges|.$$

If an initiator starts the algorithm then the transition (*Begin, Tree k_constructed*) is fired. Hence the knowledge of the partial tree k_tree is obvious (this is Post 3). It is composed of only one initiator site without edges. Next the algorithm performs a "growth" stage where new sites are incorporated into k_tree (this is Transition 4). The k_sites set is constructed with the union of the sites from the previous stage ($k-1_sites$) and in adding ($k-1_sites$) children that accept to be included in the current branch. We denote by $\text{accepted_children}(k-1_sites)$ the set of these particular children. Finally all the sites are reached by the **construction wave**. This is checked by Condition 5: $k_sites = \text{sites}$. So Transition 4 is disabled and Transition 5 is enabled. The algorithm reaches a level of knowledge where the spanning tree is completely built and where each site knows his parent and children.

Sketch of proof

From a theoretical point of view each model should be proved. For example the proof of group level specification #1 (Figure 11) stems from the fact that the building of the spanning tree of a network is a computable problem. Indeed applying the scheme of proof to this model leads to:

Hypothesis: NH

if “There exists one or several sites that decides to start the algorithm” and

the action “construction of a spanning tree called s_tree ” is performed

then the conclusion is $\text{spanning_tree}(\text{network}, s_tree)$

$$\wedge \forall i \in \text{sites}, K_i(\text{parent}(\{i\})) \wedge K_i(\text{children}(\{i\})) \wedge \text{state}_i = \text{constructed}$$

Other techniques are also needed. For example we apply a recurrent scheme for group level specification #3 (Figure 13): Transition 4 is the initialisation step. Transition 5 is the proof that if the property is verified at step n then it is verified at step $n+1$ and Transition 6 is the stop condition.

Figure 13: Group Level Specification #3 of the Construction Wave

• **Transition 4: initialisation step**

Hypothesis: $\exists i \in \text{sites} \text{ state}_i = \text{sleeping},$

Condition 4 is verified and Action 4 is performed.

$k_tree(k_sites, k_edges) = k_tree(\{\text{initiator}\}, \rightarrow)$

The graph $(\{\text{initiator}\}, \rightarrow)$ is obviously a sub-network of the main network, it is connected and its number of edges is the number of sites minus one. So k_tree is a tree.

$\forall i \in \{\text{initiator}\}, K_i(\text{parent}(\{i\}))$: because the initiator knows that he is an initiator and he is his own parent.

$\forall i \in \rightarrow, K_i(\text{children}(\{i\}))$: this is obviously true.

Conclusion: $k_tree(k_sites, k_edges) \wedge \forall i \in k_sites, K_i(\text{parent}(\{i\})) \wedge \forall i \in k-1_sites, K_i(\text{children}(\{i\}))$

• **Transition 5: recurrence step**

Hypothesis: $k_tree(k_sites, k_edges) \wedge \forall i \in k_sites, K_i(\text{parent}(\{i\})) \wedge \forall i \in k-1_sites, K_i(\text{children}(\{i\}))$

Condition 5 is verified and Action 4 is performed.

k is increased by Action 5.

• $k+1_sites = k_sites \odot \text{accepted_children}(k_sites)$

$k_sites \subseteq \text{sites}$ because $k_tree(k_sites, k_edges)$ is verified,

$\text{accepted_children}(k_sites) \subseteq \text{sites}$ by definition, hence $k+1_sites \subseteq \text{sites}$

$k+1_edges = k_edges \odot \text{edges}(\text{accepted_children}(k_edges)),$

$k_edges \subseteq \text{edges}$ because $k_tree(k_sites, k_edges)$ is verified,

$\text{edges}(\text{accepted_children}(k_sites)) \subseteq \text{edges}$ by definition, hence $k+1_edges \subseteq \text{edges},$

hence $(k+1_sites, k+1_edges)$ is a sub-network of the main network.

• By the same way we prove $\text{connected}(k+1_sites, k+1_edges)$ and $|k+1_sites| = |k+1_edges| + 1$

Hence $k+1_tree(k+1_sites, k+1_edges)$

As $k+1_tree$ is constructed incrementally from k_tree by incorporating children that accept a request from a potential parent we deduce that: $\forall i \in k+1_sites, K_i(\text{parent}(\{i\}))$ and $\forall i \in k_sites, K_i(\text{children}(\{i\}))$

Conclusion: $k+1_tree(k+1_sites, k+1_edges) \wedge \forall i \in k+1_sites, K_i(\text{parent}(\{i\}))$
 $\wedge \forall i \in k_sites, K_i(\text{children}(\{i\}))$

• Transition 6: stop condition

Hypothesis: $k_tree(k_sites, k_edges) \wedge \forall i \in k_sites, K_i(\text{parent}(\{i\})) \wedge \forall i \in k-1_sites, K_i(\text{children}(\{i\}))$

Condition 6 is verified and Action 6 is performed

As $k_sites = sites$ (this is Condition 6) we are going to prove recursively that:

$\forall k \in D, k_tree(k_sites, k_edges)$

$k = 0$: $k_tree(k_sites, k_edges)$ is verified. Indeed, it has been proved with Transition 3.

$k_tree(k_sites, k_edges) \Rightarrow k+1_tree(k+1_sites, k+1_edges)$. This is Transition 4.

Conclusion: $\text{spanning_tree}(\text{network}, s_tree) \wedge \forall i \in sites, K_i(\text{parent}(\{i\})) \wedge K_i(\text{children}(\{i\}))$

The compliance rule and the proof that are presented with the example need to be defined more formally. In the next section we present how from a group specification we can give an object specification.

7.4 Object Specification

Now we present how starting from a group specification we can give an object specification. Unless our point of view is different (object rather than group level) the general idea of our methodology highlighted by this example is that the properties defining the overall behaviour of a distributed algorithm should be transmitted along all the inheritance hierarchy of specifications. By this way we consider that the refinement process used with the three levels of our methodology is twofold: a refined model uncovers some undeterminism contained in previous models and a refined model enforces the behavioural constraints defined in previous models.

The spanning tree construction algorithm is a symmetrical algorithm. Each site runs the same behaviour in the network. Then, we have only to describe one object model with as many refinement steps as needed. The model is a refinement of the group level ones. Objects (or object classes) are made of methods and global variables.

7.4.1 Identification of global variables

We identify three kinds of variables. According with the H1 and H4 Hypotheses, an object has at least two essential variables named **Self** and **Neighbour**. **Self** is the identifier of the site and **Neighbour** is the set of neighbours identifiers. The value of these variables must be initialised at creation time to respect the initial hypotheses.

The group specification indicates the index variables **State(i)** and **Epoch_num(i)**, i being the site identifier. They help in the construction of the spanning tree. **State(i)** defines the step where the site is during the algorithm running (sleeping, construction, constructed, termination). **Epoch_num(i)** allows to distinguish two successive constructions of the tree. Each object must therefore have a variable for **State(i)** and **Epoch_num(i)** named **State** and **Epoch_num**. Details about the epoch number values can be found in [4]. We intentionally omit to develop this discussion in this paper to clarify the text.

All knowledge noticed $K_i(\dots)$ are represented by variables. The object must have therefore variables to define $K_i(\text{father}(\{i\}))$, $K_i(\text{sons}(\{i\}))$, $K_i(\text{root}(s_tree))$ named **Father**, **Sons** and **Root**. These data are modified by the construction recursive wave, and finally, they receive the result of the algorithm.

7.4.2 Identification of main methods

The construction and termination phases are the two recursive waves. The aim is to build the tree structure as a wave associated with the construction phase. The wave associated with the termination phase ends the algorithm. Therefore each site must own two methods, one for each phase. We call then **Construction** and **Terminate** methods. They are the interface of the object class:

Method **Construction**(Param_root, Param_father, Param_epoch) : Response

Method **Terminate**

The Construction method can return three kinds of result. The proposal receiver (invoked site) returns a favourable notification when he accepts the sender of proposal as father. He returns a unfavourable notification when he refuses the proposition, or he returns a particular value corresponding with a better initiator. So this last kind of result has the sense of reject. We use a vector **Response** to store the invoked Construction methods responses from the set of neighbours when the wave comes backward and to build the returned result of the method. The value of Response(i) is OK if the neighbour accepts to be the son, i being the neighbour number. If the neighbour has the same Root value than the proposed father, the neighbour refuses the proposal and Response(i) is NOK. If the value of

the neighbour Root is better than that one proposed by the father, the value of the Response(i) is BEST. Denote that the **Sons** variable does not change when the response of at least one neighbour is BEST.

The action associated with the construction method is triggered when the receiver object is either in the sleeping phase or in the construction phase. The Terminate method has no output result. The action associated with the Terminate method is triggered if the object receiver is in construction phase.

7.4.3 Object specification #1

The model is a refinement of the ones presented in figures 12 and 13. Figure 14 gives the first object specification.

Figure 14: Object Level Specification

In Figure 14 states Begin, Construction finished and End introduced in the group specification are mapped onto their object counterparts. Unless their predicates are not given explicitly, they can be considered as the object versions of group predicates.

Two separate behaviours are defined for the refinement of the **construction** and the **termination waves**. On the left side of Figure 14 we present the behaviour of an initiator and on the right side the one of an intermediate site.

First an initiator decides to perform the algorithm. Then he spread the wave on his neighbours (this is action Neighbours.Construction). If he receives a better proposal of wave from another initiator (this is the transition between the states Initiator and Node) he knows that he is no longer the winning initiator of the tree and he must perform the behaviour of an intermediate node. If no better wave arrives he reaches the Construction finished state and can then spread a **termination wave** (this is action Children.Terminate).

An intermediate node starts his behaviour by receiving a proposal of wave. He spreads it to his neighbours. While he has not received a **termination wave** he does not know whether the construction is completed or not. This is why our model translates the fact that the states "Node" and "Potential children known" can be active simultaneously. As soon as a Terminate method call arrives he reaches a state where the construction is finished. Then he can spread the Terminate method call to his children (this is action Children.Terminate).

7.4.4 Minimal synchronisation scheme

As previously, models of the object level are concurrent. Hence several actions can be performed simultaneously. As these actions perform read and write operations on shared variables, some synchronisation scheme is needed to ensure the coherence of these data. In order to carry on the object specification we need to determine all the possible sets of transitions that require some synchronisation. Two actions that modify a same shared variable cannot be executed in parallel. So we must serialise these operations. Each action is at least in mutual exclusion with itself. Indeed, the associated post-conditions show that all the actions modify at least a global variable. We must therefore serialise these operations. But we will like to minimise the serialisation to benefit from the parallelism. We can already rule out the Termination method. Because the termination wave occurs only one time by spanning tree construction, it is unnecessary to serialise the Termination method.

So, we single out two kinds of synchronisation. One comes from the behaviour of our model and the other comes from the semantics of the algorithm that we specify. First, both the transition between the states "Begin" and "Initiator" (Figure 14) and the one between "Begin" and "Node" use the shared variable Father. Nevertheless, from the definition of the semantics of a state given in Section 4, we can deduce that the synchronisation needed for this situation is provided by the execution engine of the object system. Indeed we can translate the model as follows: the behaviour of an object starts in the state Begin with the predicate "State = sleeping". One of the two transitions starting from this state is fired and its action is performed.

- On the one hand, if the transition that leads to the state "Initiator" is chosen, the initial state "Begin" is disabled (because its predicate and the one of the state "Initiator" "State = construction $\wedge$ Father = Self" can not be true simultaneously). Then the other transition can not be fired.
- On the other hand, if the transition that leads to the state "Node" is chosen, the initial state "Begin" is disabled (because its predicate and the one of the state "Initiator" "State = construction $\wedge$ Father = caller" can not be true simultaneously). Then the other transition can not be fired.

The second kind of synchronisation can be witnessed with the transition between the states "Individual sons known" and "Sons known". Indeed, as several construction waves with different initiators can be performed concurrently on one site, several threads can try to fire this transition. As the shared variable Sons is updated we must synchronise this transition. So, the following actions need to be synchronised:

- The action associated with the condition "to declare an object as an initiator" between the states "Begin" and "Initiator",
- The action associated with the condition "to accept a proposal from a father" between the states "Begin" and "Node",
- The actions associated with the condition "construction end" between the states "Individual Sons known" and "sons known", and the condition "Potential individual Sons known" and "Potential sons known".

Moreover all the actions except "Neighbours.Construction" method are *a priori* in mutual exclusion. Indeed they share variables. However, we know that when the notification of termination happens, a new proposition can not be accepted. Likewise, it is not possible to declare itself initiator, and this, until the concerned object will in sleeping phase. We can put out the **Terminate** method. Finally, these synchronisation clauses can be written as follows:

- As the actions with the condition "to declare an object as an initiator", "to accept a proposal from a father" and "sleeping phase" share the variable State:

```
current(to declare an object as an initiator)+ current(to accept a proposition from a
father)+current(sleeping phase)=0
```

- As the action with the condition "construction end" modifies the global variable Sons, several occurrences of this action can not occur concurrently:

```
current(construction end)=0
```

current(m) is the number of method occurrences m that are being executed.

7.4.5 Object refinements

We now describe more precisely the object transitions. Each transition is specified in two parts. First, we analyse the object specifications in splitting up the blocks of code by functionalities. Then we refine the object behaviour in accordance with the synchronisation scheme (expressed in condition) and the different blocks.

7.4.5.1 To declare an object as an initiator

First, an object becomes an initiator when he proposes to himself to construct a spanning tree with himself as the root. He invokes the Construction method on himself. Previously, we have determined three parameters for this

method. These parameters are the identifier of the initiator site, the identifier of the caller, and the epoch number at the time of the call. They are denoted Param_root, Param_father, Param_epoch. To be an initiator is invoked by the construction method as follows:

```
self.Construction(Self, Self, Epoch_num)
```

This method begins by determining whether the object is an initiator or not. It is the case when the proposed father (Param_father) is equal to the site identifier value. In other cases, the proposal comes from an external site.

```
1.      /*Test of the knowledge of initiator */
2.      IF Param_father=Self THEN
3.          /* The object is an initiator */
4.      ELSE
5.          /* External proposal */
6.      FI;
```

Block 1 : Knowledge of the initiator

Furthermore a site can declare himself initiator only if he is not performing any construction (e.g. if his variable State is equal to the value "sleeping").

```
1.      /*Test checking the possibility to declare initiator itself
2.      IF State= sleeping THEN
3.          State:=construction;
4.          Root:=Self;
5.          Father:=Self;
6.      ELSE
7.          /* to refuse to declare itself initiator */
8.      FI
```

Block 2 : Able to be an initiator

7.4.5.2 To accept a proposal from a father

When a request comes from an external site, the value of the parameter Param_father is different from Self. The clause ELSE in Block 2 is performed. This corresponds to the behaviour specified in the state "Node" in Figure 14.

The condition "to accept a father" is tested by a sequence of instructions. If this sequence is checked, then the object is in the state "Node". This request comes from an external invocation of Construction method. In accordance with the Block 1, it is the clause ELSE.

To accept a proposal of father, the object must be in the same epoch. If it is not the case, the object waits until it will be in this epoch.

```
1.      /* Constraint to satisfy before to accept a proposal from father*/
2.      WHILE Epoch_num< Param_epoch DO nothing OD;
3.      /* test of acceptance of the father proposal */
```

Block 3 : To be in the same epoch

The object analyses the proposal from the past epoch e.g. it is a lost wave doubled by a winner wave. Then, every proposal from a corresponding epoch is agree if the object is in the initial state "Begin". In the other case, the object is in the previous states "Initiator", "Node", "Potential sons known", then he has already a father and he is in the construction phase. The new father is accepted if the initiator of the wave has a better identifier than the potential root of this object.

```
1.      /* Constraint to satisfy before to accept a proposal from father */
2.      IF Param_epoch<Epoch_num THEN
3.          /*Reject the proposal by a response of BEST type*/
4.      FI
5.      IF State= sleeping THEN
6.          /* first proposal */
```

```

4.      State:=construction;
5.      Root:=Param_root;
6.      Father:=Param_father;
7.      ELSE
8.      IF Best_proposal(Param_root) THEN
9.      /* others proposals of best initiator*/
10.     Root:=Param_root;
11.     Father:=Param_father;
12.     FI
13.     FI

```

Block 4 : Some constraints to accept before

7.4.5.3 Construction end

To satisfy the condition "Construction end", the wave must be created, spread, and the response must be analysed. From the initiator view point, this condition means that he is the winner and he becomes in the state "Known sons". The other sites are in the state "Potential sons Known". The wave is initialised or spread by the parallel invocation of the Construction method on its neighbours.

```

1.      /* Initialisation of local variables*/
2.      My_response:=OK;
3.      Local_sons:=empty set;
4.      Concerned_neighbour:=Neighbour minus Potential_father;
5.      /* parallel call on neighbour */
6.      PAR i in Concerned_neighbour DO
7.          Response(i):=Concerned_neighbour.Construction
                                (Param_root, Self, Param_epoch);
8.          /* response analyze*/
9.          IF Response(i)=BEST THEN
10.             My_response:=BEST;
11.          ELSE
12.             IF Response(i)=OK THEN
13.                 /* the neighbour i accepts to be my son */
14.                 Local_sons:= Local_sons v i;
15.             FI
16.          FI
17.      ENDPAR

```

Block 5 : Spreading of the wave and analysing of the result

When a construction wave is finished we must record the new sons. But when a site receives a better proposal of wave the old one is performed till the end. Thus to avoid recording the old values we must check whether a better proposal of wave is arrived or not.

```

1.      /* Final Block*/
2.      IF My_response _ BEST AND Root=Param_root
                                AND Epoch_num=Param_epoch THEN
3.          /* Satisfied Post_condition */
4.          Sons:= Local_sons;
5.      FI

```

Block 6 : End of the wave computation

7.4.5.4 Termination phase

The knowledge of the root allows to go to the state "End".

```

1.      /* Termination Step */
2.      State:=terminated;
3.      /* spread to the sons */
4.      PAR i in Sons DO

```

```

5.          Sons(i).Terminate;
6.      ENDPAR

```

Block 7 : Terminate wave

7.4.5.5 Sleeping Phase

The spread of the wave comes back to each site. Each site becomes in the sleeping phase and the epoch number is increased.

```

1.      /* Go back to the initial State */
2.      State:= sleeping;
3.      Epoch_num:= Next_Epoch();

```

Block 8 : Sleeping step

7.4.5.6. Summary of synchronisation schemes

The next table summarises our knowledge about synchronisation rules between blocks. We must try to refine this synchronisation scheme.

Events	Actions	Control clauses
to declare an object as an initiator	block 1 and block 2	current (block 1 ; block 2) + current (block 3 ; block 4) + current (block 8)=0
to accept a proposal from a father	block 3 and block 4	current (block 1 ; block 2)+ current (block 3 ; block 4)+ current (block 8)=0
construction end	block 5 and block 6	current (block 5 ; block 6)=0
termination phase	block 7	NIL
sleeping phase	block 8	current (block 1 ; block 2) + current (block 3 ; block 4) + current (block 8)=0

Table 1 : Scheme synchronisation

- To declare an object as an initiator (blocks 1 and 2):

The block 1 uses a local variable (Param_father) and the global variable Self is accessed only on reading. We don't need to protect this block. Then the control clause of the block 2 can be written as follows:

```
current (block 2) + current (block3; block4) + current (block 8) = 0
```

- To accept a father proposal (blocks 3 and 4)

The block 3 must not be in mutual exclusion. Indeed, it should caused a deadlock. The block 4 must respect the control clauses because it uses shared variables as Father, Root and State.

```
current (block 4) + current (block 2) + current (block 8) = 0
```

- Sleeping phase

The mutual exclusion of the block 8 depends on the block 2, 4, and 6. They share the update of State variable.

```
Current (block 2) + current (block 4) + current (block 6) + current ( block8)=0
```

Then, we have define eight blocks and their associated control clauses. In the language Guide, the control clauses are applied to methods. So, the blocks 2, 4, 6 and 8 become methods (respectively To_be_initiator, Proposal_father, Register, and End_terminate) and they have an associated control clause of mutual exclusion written as follows:

```
EXCLUSIVE (To_be_initiator, Proposal_accept, Register, End_terminate)
```

Blocks 1 and 3 must be in the Construction method and the block 7 in the Terminate method. The block 5 uses no global variables and has no conflict with others methods. It is grouped with a procedure called Spread.

7.4.7 Object specification #2

Figure 15 refines the first object specification. We obtain this specification after studying all the actions and their interactions.

Figure 15 : Object Specification #2

In the next step, we specify the operations of each action. The method specification performs a text expansion of the method calls found in the object specification. Proofs must be associated with the specification methods. This proof obligation is mainly obtained by substitution of variables[4]. Finally, the pseudo code is obtained. Implementation of this pseudo-code was written and tested in Guide Language.

8 Conclusion and Perspectives

In this paper we define a method that introduces guidelines useful for the specification of distributed algorithms. Its originality is to merge concepts derived from algebraic formal methods (B Method), distributed artificial intelligence (knowledge operators and reflexivity) and distributed algorithms (distributed control structures and protocols for order properties control).

We introduce the notion of refinement levels: group, object and method level. Next we adopt a particular syntax and semantic to describe distributed behaviours. This semantic with its graphic syntax or language is derived from the B Method. It is used for each refinement step.

There are many problems that must still be solved. From a practical point of view, we have to improve our approach by developing several different kinds of distributed algorithms. From a theoretical point of view, it is important to associate in a more formal model the different modal logic tools (e.g. epistemic logic, deontic logic and temporal logic of actions). The behavioural reflexive approach of distributed artificial intelligence must be correctly integrated in our work. In the presence of fault or performance modification, it is necessary to change dynamically the behaviours. This last behaviour must be clearly distinguished from the normal behaviour. Another large project is to develop a set of tools that takes into account and implements all the previous concepts.

9 References

- [1] J.R. Abrial. The B-Book. Cambridge University Press, 1995. To appear.
- [2] M. Awad et al.. Object-Oriented Technology for Real-Time systems, Prentice Hall, 1996.
- [3] R. Balter, J. Bernadat, D. Decouchant, et al. Architecture and implementation of Guide, an Object-Oriented Distributed Systems. Computing Systems, 4(1):31-67, 1991.
- [4] L. Bonnet, Spécifications et Preuves d'Algorithmes Distribués en Approche Orientée Objet. Mémoire CNAM, Rapport de Recherche CEDRIC 94-22, Décembre 1994.
- [5] G. Booch. Object Oriented Analysis and Design. 1994.
- [6] Bochmann, Barbeau, Bean, et al. The Description of the Specification Language MONDEL V1. Technical Report, Centre de Recherche Informatique de Montréal, April 1991.
- [7] C. Coste. Appel de procédure sur groupe en approche objet réparti. Mémoire CNAM, Juin 1995.
- [8] E.W. Dijkstra. A Discipline of Programming. Prentice Hall, 1976.
- [9] G. Florin, R. Gomez, I. Lavallée. A distributed Recursive Solution to the shortest path problem. Congresso IO 94, Braga, Portugal, April 1994.
- [10] R. Gomez. Conception et spécification d'algorithmes distribués: la vague récursive. Thèse de doctorat, Université Paris VIII, Mars 1995.
- [11] D. Gries. The Science of Programming. Springer Verlag, 1981.
- [12] J.Y. Halpern, Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. Communications of the ACM, 1990.
- [13] J. Halpern, Y. Moses. A Guide to Completeness Complexity for Modal Logics of Knowledge and Belief. Artificial Intelligence, 54 (1992), pp 319-379.
- [14] C.A.R. Hoare. The Axiomatic Basis of Computer Programming. Communications of the ACM, 1969.
- [15] ODP: Open Distributed Processing. Reference Model ISO10746.
- [16] L. Lamport. The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems, 16(3):872-923, May 1994.
- [17] I. Lavallée. Arbre Couvrant et Vague Récursive. Communication personnelle.
- [18] CORBA, Object Management Group, Framingham Corporate Center.
- [19] S. Owicki, D. Gries. An Axiomatic Proof Technique for Parallel Programs. Acta Informatica, Vol 6, 1976, pp 319-336.