9

Some Specification and Proof Steps of a Spanning Tree Algorithm with an Object-Oriented Method

L. Bonnet, L. Duchien, G. Florin, L. Seinturier
CNAM-Laboratoire CEDRIC
292, Rue St Martin, FR-75141 Paris Cedex 03
e-mail : bonnetl@micronet.fr, {duchien, florin, seintur}@cnam.fr

Abstract

This paper presents a way to specify, to prove, and to implement object-oriented distributed cooperative algorithms. These algorithms present several particular aspects due to their parallel and distributed environment. In order to specify and to prove distributed object-oriented algorithms our method gathers three kinds of tools or formal concepts [Bonnet 94]. It is derived from the B Method [Abrial 95]. We also use the Temporal Logic of Actions [Lamport 94] to express some properties. The third set of concepts comes from distributed artificial intelligence. A main aspect of parallel and distributed computation is the uncertainty on the knowledge of other site states. To express it, an epistemic logic is used to define knowledge of sites [Halpern 90]. The method is illustrated by the first specification and proof steps of a spanning tree distributed algorithm.

Keywords

B Method, object-oriented method, distributed knowledge, spanning tree algorithm.

1. INTRODUCTION

Interest in the object paradigm has been growing in the software world. This structured application development improves modularity, maintenance and program reusability. The success of the object approach is mainly in the sequential programming domain. Nevertheless, there are several studies that design or extend object languages with parallel and concurrent control clauses. Furthermore, distributed operating systems have been designed in order to consider object technolog. Their main feature is to implement a multi-threaded, transparently distributed virtual machine that supports the object paradigm.

E. Najm et al. (eds.), *Formal Methods for Open Object-based Distributed Systems*

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 propose a model of specifications and proofs approach suited to distributed algorithms with an object-oriented approach.

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 synchronized finite state automata used in network protocol modeling. Roughly speaking, we define each state by a predicate about the data of the different objects used in a model. If needed we use epistemic modal logic predicates; that is to say, predicates on knowledge about variables values. Notice that such a model description can be considered as a particular temporal logic of actions model [Lamport 94]. Such a model can also be seen as an abstract machine in the B Method [Abrial 95]. So the proof technique, for each abstract machine model, can use as in B, a set theory approach.

Second, 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 behaviors of concurrent and distributed sites or objects. To this end, these specifications describe the knowledge evolution of the group of communicating 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 [Halpern 90] to model the state predicates of the group specifications.

At the same time, the group behaviors are implemented by the concurrent actions of sites or objects. The intra-object specification represents the behavior of one member of the group. So the evolution of the global variables of the object and the synchronization needed for its internal concurrence control are defined in the different refinement steps of this level.

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 realize the desired actual behavior. Finally a specification can be translated into a distributed object-oriented language, and then executed. Our target distributed object environment is the Guide system [Balter 91].

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

The original feature of our specification method is to propose to gather together several kinds of tools or formal concepts, such as the B Method, temporal logic of actions, knowledge predicates.

2.1. The B Method

Among the many existing formal methods that can be used, we choose the B Method. 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 take this context into account.

One main characteristic of parallel and distributed algorithms is the existence of partial order relations that generally allow one to prove correctness properties. For example, a causal relation between events or actions [Lamport 78] is frequently considered in accordance with the definition of consistent global states or with message ordering. In this way, to extend the B Method for parallel and distributed applications is interesting because of its set theory background. Indeed, it allows easily one to define and to use order relations between sets of objects, or sets of events, or sets of actions.

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 (contained objects and methods in the object world). Data are encapsulated in a machine and can only be accessed through services (method invocations). A service is specified by a substitution on data. A general substitution language with numerous features (such as guards, pre-conditions, choice commands) is provided. The proof technique by assertions uses a set theory formalism.

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. 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 : its begin and its end events.

In this context we must consider an interval logic as the underlying modal logic. We choose the Temporal Logic of Actions (abbreviated TLA)[Lamport 94]. Lamport writes that the execution of an algorithm is thought as a sequence of steps. Each step produces a new state by changing the values of one or several variables. An execution is the resulting sequence of states. An algorithm is the possible execution set. Therefore, reasoning about algorithms requires reasoning about sequences of states. An algorithm is described by initial value definitions and sequences of actions expressed by TLA formula with modal operators such as *always* ($\Box$) or *eventually* ($\Diamond$). Then safety, liveness or fairness conditions can be specified. Lamport proposes to prove 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 agent) states. This is due either to transmission delays or to failures or to the dynamic behavior of interacting agents that learn knowledge when running. Hence the performance or fault tolerance features or uncertainty aspects must be correctly considered. In our method, an epistemic logic approach is used to define knowledge of sites [Halpern 90]. It allows one to model environments of entities, knowledge levels and groups behaviors.

Halpern argues that the right way to understand distributed computing is to consider 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 $K_i\varphi$ means the process i has the knowledge of a given fact φ.. The distributed knowledge of φ is denoted $D_G\varphi$. 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 $E_G\varphi \equiv \bigwedge_{i\in G} K_i\varphi$.

Finally, the highest state of knowledge, the common knowledge of the group is denoted $C_G\varphi$. $C_G\varphi \equiv E_G\varphi \wedge E_G^2\varphi \wedge \ldots \wedge E_G^m\varphi \wedge \ldots$

3. MODEL OF SPECIFICATION FOR COOPERATIVE APPLICATIONS

The distributed object-oriented paradigm helps designers to master the complexity of cooperative systems. We 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).

3.1. Group specifications

First, the group specification level handles global states, global knowledge, and the global interactions between these objects to reach a given goal in a defined environment. So the highest level must define the chosen strategy to solve the problem e.g. the global view of the distributed system behavior. 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. We use distributed global control structure such as recursive waves, phased algorithms or group remote procedure calls as a complex global behavior. We describe these global behaviors by state/transition models (see Section 4). The group level specifies and proves the global states and global actions that the group must successively encounter and realize to solve a problem. In relation with the refinement aspects of the B Method, as long as we do not point out the behavior of a precise object, we stay at the group level.

3.2. Object specifications

In our method 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 methods of the object or synchronization variables such as event counters. For example *head, queue, put(), get()* for a synchronized buffer are variables and methods that must be managed by a synchronization scheme. Too, we can define several order properties (e.g. local, causal, global). We also deduce some local variables from the global knowledge of the cooperative system. For example from the group predicate $\forall\, i \in$ *sites,* $K_i(Id(i))$, we can say that each object knows its own identifier and defines a variable to store this knowledge.

The behavior clauses of an object defines the coordination between different actions such as synchronization, scheduling of actions, and sharing of resources. It also ensures the object coherence. The object coordination can be defined as an active entity that manages the set of

object methods. As we must consider parallel executions, the global data of an object must be studied from the concurrence control point of view. Furthermore, as objects belong to a group, some parts of the global behavior must be reflected on each object specification. For instance, if we find in the group specification a synchronization state where every object must get through, then the object specification must reflect this behavior. The behavior of an object is also modeled by a state/transition model as defined in Section 4.

3.3. Method specifications

The behavior of method level is the last of the refinement steps. It specifies the internal mechanisms of the object that characterize its behavior. Thus we must only use knowledge that can be managed by a single method. Local variables of a method can be accessed without restriction because they are only managed by the considered method. To update global variables of an object is only possible with the guarantee of a concurrence control scheme defined in the object level. Then, the specification of the method level respects the synchronization scheme. The specification of the method level provides basic algorithmic structures such as loops, tests, arithmetic operations, reads and writes of variables. The result must be a sufficiently precise specification and a proof of an algorithm that is translated into a programming language.

4. SEMANTICS AND SYNTAX OF THE METHODOLOGY

The aim of our methodology is to improve the specification of distributed algorithms. So we define more precisely several syntax and semantics aspects of a model. One of the key features of our method is to use a common paradigm (a state-transition approach) for the three levels of specification. Figure 4.1 presents the basic components of a specification in a graphical representation. States are associated with a predicate. Transitions are edges between states and are labeled by a condition, an action, and a post-condition. We add a description of the environment by the hypotheses defined by a predicate. The semantics of the model supports concurrence. So the model must be clearly distinguished from the sequential finite state model used in message passing protocols. Several transitions can be fired simultaneously.

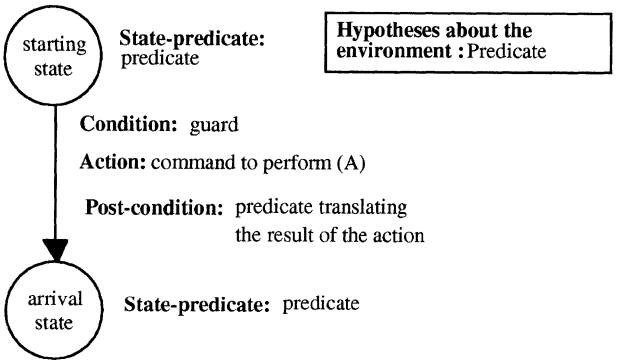

Figure 4.1 : Basic components of the model.

4.1. Hypotheses about the environment

The hypotheses about the environment must be seen as a program invariant. They must be preserved by all the operations defined in the model. Nevertheless some part of them may deal with features that are preserved when you study your application but that may be influenced by other applications or by the environment itself. For instance in the spanning tree construction algorithm (Section 7), we assume that each site of the distributed system owns an unique identifier. This assumption is preserved by all the actions but there are no guarantee that external actions do not modify it. According with this remark we could say these kind of assumptions need to be checked permanently. As this solution is too expensive, we adopt the belief modality [Halpern 92] for them to make it clear that they may be influenced by external actions.

4.2. States

A state represents a significant point in a distributed run, in an object behavior, or in a method execution. A state can be enabled or disabled. It is defined to be enabled (or assessable) when two conditions are fulfilled.

First and foremost a state must be reached along a sequence of states (a sequence of causal actions) starting from the initial state of a specified behavior. The first aspect describes the behavior using precedence rules as it is done for instance in distributed phased algorithm or in a simple program. This is a control scheme point of view. If a state is just used to represent such a simple point in a sequence, it is said to be sequential and, in this case, it is disabled when its consequent actions are activated. Nevertheless a state can also be used to activate concurrently several actions. In this case it is called parallel.

In a second time a predicate is associated with each state. Once a state is reached and, as long as its predicate is true, it stays enabled and its consequent transitions can be evaluated. The predicate allows to model conditions on data. As we consider an object approach, data must be accessed using methods. Hence state predicates should only be defined as knowledge about methods.

A state is characterized by a relation from a set of variables to a set of values. This relation is symbolically represented by a Boolean valued predicate. Once a state is enabled and as long as its predicate holds, its consequent transitions can be evaluated. 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. By this way concurrent behaviors can be represented. 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 instanciation of the class, or the beginning of the method.

4.3. Transitions

A transition defines a step or a phase between two states in the execution of a behavior. It is composed of three elements: a condition, an action and a post-condition. The pair condition/action defines a guarded command [Dijkstra 76]. 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 initial state predicate, then this state stays assessable.

The condition is a Boolean expression and acts as a guard that must be true. Conditions are

formula based on the usual 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 post-condition predicate. An action can be decomposed into its invocation that is local and its execution that can be local or remote. 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) can be used.

The post-condition is a predicate related to an action. It is mainly defined to prove the correctness of an action run. It specifies the result of the executed action. A post condition is similar to a state predicate and includes knowledge. When several transitions lead to the same state, thi state predicate is the disjunction of post-conditions associated with each transition.

5. REFINEMENT FEATURES

We propose to specify a distributed algorithm by a hierarchy of models. Our basic assumption about the refinement process is that it must reduce the space of solutions of the model. 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 (e.g. compatible with the final specification).

We mentioned above the possibility of several specification levels. Indeed, each level is one step of refinement that can be itself refined. The designer assumes that the original model is does not contradict the refined one. The refined elements are the actions or post-conditions.

5.1. Refinement within a same specification level

For every algorithm specification a first specification model may consist of defining the initial and final states of the problem. The action associated with this first transition is the algorithm itself (Figure 4.1). 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 result.

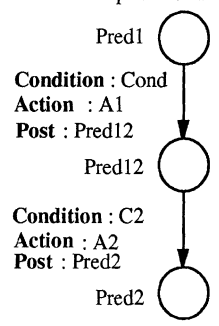

Figure 5.1 : Sequential composition of sub-actions.

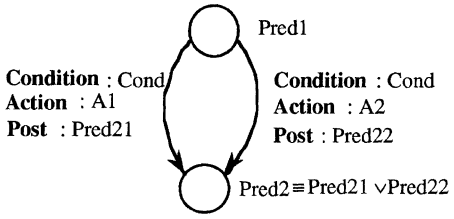

Figure 5.2 : Parallel composition of sub-actions.

The second way of refining an action is conducted by enhancing its post-condition. Let us consider an underspecified behavior where an action is not yet designed. For instance, in

Figure 5.3 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.

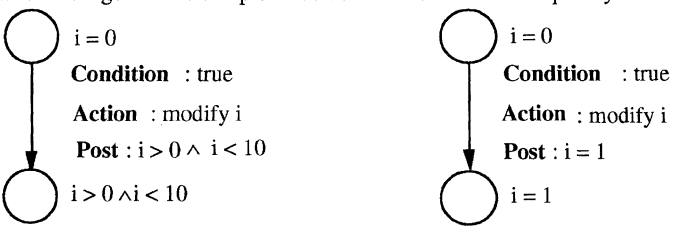

Figure 5.3 : Refinement of an action by enhancing the post-condition.

5.2. Refinement between two different specification levels

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 case, each agent or site has the same behavior. When the distributed application is symmetric (e.g. the same code runs on each site of the environment with different initialization 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 with the modularity principle. Each object perform a subset of different actions. Let us consider a object-oriented distributed 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 models: one for the server and one for the client. Finally, the refinement between an object level and a method level details each block of the object behavior. Only, some local details must be refined and proved.

6. PROOF TECHNIQUE

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

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 action.

Various theoretical results are applied to carry such proofs. This process was initially studied by Hoare for sequential algorithms [Hoare 69] until [Abrial 95]. It was modified for parallel algorithms [Owicki 76][Gries 81].

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 (or sites) interconnected through a connected network cooperates to get organized into a tree topology. The proposed solution is a simple example of a recursive wave algorithm [Lavallée 94]. A site that decides to begin the construction of the tree is called the initiator. He sends a

request to its neighbors notifying them that they are his potential children (Figure 7.1). A potential child may receive several propositions. So he must choose between them in order to have finally only one parent. In the proposed solution he decides to accept the first one. Hence he knows who is his parent. Then each child spreads the wave to his neighbors minus his parent (Figure 7.2). If a request reaches a site that already knows his parent, then he rejects the proposition. Once all the sites has been visited the wave flows back to the initiator. At the end of the **construction wave** each site knows his parent and children.

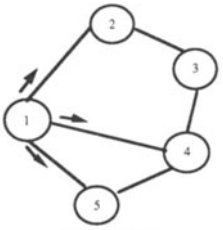

Figure 7.1

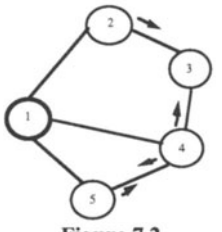

Figure 7.2

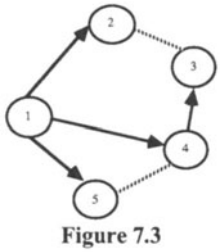

Figure 7.3

The initiator is now the root of a spanning tree. He is the only one to know that the construction is finished (Figure 7.3). Then to notify the end of the construction to all sites he 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.

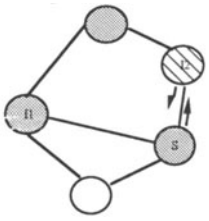

Figure 7.4: Spanning tree construction algorithm with two initiators I1 > I2
S belongs to I1's construction,
S receives a request from I2,
S returns the request.

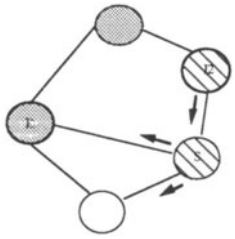

Figure 7.5: Spanning tree construction algorithm with two initiators I1 < I2,
S belongs to I1's construction,
S receives a request from I2,
S records its new parent I2,
S spreads the wave initiated by I2.

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 looses and S returns a wave-loser response to I_2 (Figure 7.4). If I_2 identifier is greater than I_1's then S records I_2 as its new potential root and spreads the wave initialized by I_2 (Figure 7.5).

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.

7.1. Hypotheses about environment

The Network Hypotheses (NH for short) are pre-conditions that must be satisfied to correctly perform the algorithm. These hypotheses are defined as believes about the distributed execution environment. They are not verified but considered as always true. We denote by K_i the knowledge operator of a site i.

H1: The network is defined locally by the knowledge of the neighborhoods:
$\forall i \in sites, K_i(neighbors(\{i\}))$
H2: The communication links are symmetric:
$\forall i,j \in sites, j \in neighbors(\{i\}) \Rightarrow i \in neighbors(\{j\})$
H3: The sites identifiers are totally ordered:
$\forall i,j \in sites, Id(i) = Id(j) \vee Id(i) > Id(j) \vee Id(i) < Id(j)$
H4: Naming: each site knows its identifier and its neighbors identifier:
$\forall i \in sites, K_i(Id(i)) \wedge \forall j \in neighbors(\{i\}), K_i(Id(j))$
H5: No homonyms:
$\forall i,j \in sites, i \neq j \Rightarrow Id(i) \neq Id(j)$
H6: Network connexity:
$\bigcup_{k=0}^{\infty} neighbors^k(\{choice(sites)\}) = sites$
choice() is the choice axiom that allows to take a particular element in a set.
H7: No failures
$\forall t>0, \forall k, (sites,neighbor)_t = (sites, neighbor)_0$

This is the beginning of the full specification of the solution [Bonnet 94]. As it is very long and detailed we only give the first steps as an example of the method. Hence these first steps include many details that are necessary not used in the paper.

7.2. Group level specification of the spanning tree #1

The goal of the algorithm is to organize a network satisfying the Network Hypotheses (NH) into a tree structure. We must give a general understanding of the evolution of the group of sites. Our refinements are sequential and denoted by #1, #2, #3. Our first group specification is:

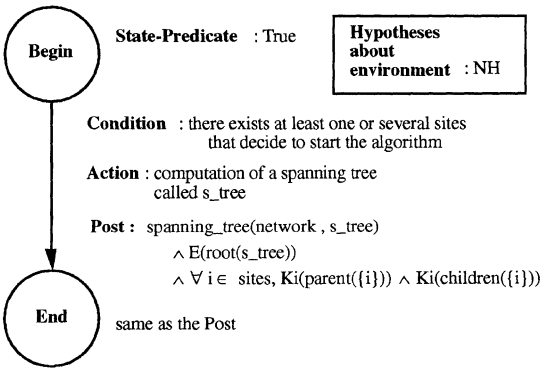

Figure 7.6: Group level specification #1.

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 usual 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-predicate is verified:

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

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

- **spanning_tree**(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:
  network = (sites, edges), s_tree = (s_sites, s_edges).
  spanning_tree(network,s_tree) $\Leftrightarrow$ s_sites = sites $\wedge$ s_edges $\subseteq$ edges $\wedge$
  connected(s_tree) $\wedge$ |s_sites| = |s_edges| + 1
- **E(root(s_tree))**: each site knows the root of the spanning tree.
- $\forall$ **i** $\in$ **sites,** $\mathbf{K_i}$**(parent({i})) $\wedge$** $\mathbf{K_i}$**(children({i}))**: each site knows its parent and children.

7.3. Group level specification #2

Let us give more details about specification #2. The chosen solution performs the construction of the spanning tree in two phases. 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. A transition between the states *Begin* and *End* allows us to express that the algorithm is cyclic. Indeed, once it is finished, the construction can be restarted.

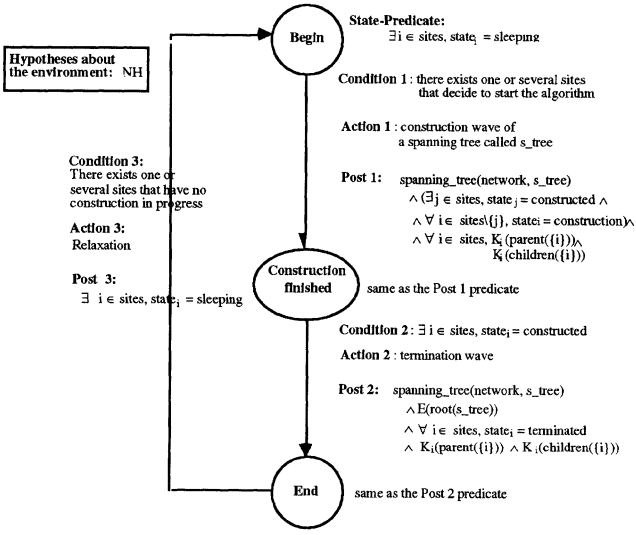

Figure 7.7: Group level specification #2.

state = (.., state$_i$, ..) is a vector that represents the situation of the n sites. In a periodic spanning tree construction, each site can be in four states:

- 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 in the **termination wave**

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. The aim of this new refinement step is to translate the traversal of the **construction wave** throughout the network. Starting with an initiator, the construction of each branch of the tree is computed concurrently. According to our model, several instances of the transition between the *Begin* and *Construction finished* states may be activated simultaneously. The aim of this new refinement step is to initiate the reader with the proof mechanism that is associated with our method.

7.4. Group level specification #3

Let us introduce more details about the distributed behavior. Each branch of the tree is composed of some nodes. These nodes are added in a partially built tree when the construction wave flows down. Several branches perform this work concurrently but, from a group level point of view, we consider that the tree is constructed step by step. So we introduce the *Tree k_constructed* state to represent this mechanism.

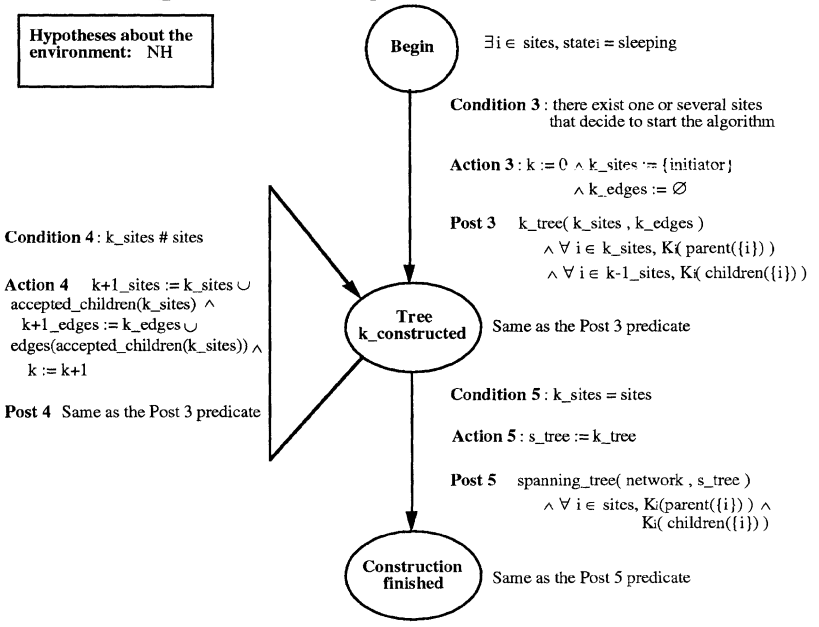

Figure 7.8: Group level specification #3 of the **construction wave.**

In Figure 7.8 the level of knowledge associated with this intermediate state is:
k_tree(k_sites, k_edges) $\wedge \forall$ i $\in$ k_sites, K_i(parent({i})) $\wedge \forall$ i $\in$ k-1_sites, K_i(children({i}))

k_tree(k_sites, k_edges) predicate 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 $\subseteq$ sites and k_edges $\subseteq$ edges). As k_tree is also a tree, the following property is verified: connected(k_tree) $\wedge$ |k_sites| - 1 = |k_edges|If an initiator starts the algorithm then the transition (*Begin*, *Tree k_constructed*) is triggered. 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 added into k_tree (this is the Transition 4). The k+1_sites set is constructed with the union of the sites from the previous stage (k_sites) and the (k_sites) children that accept to be included in the current branch. We denote by accepted_children(k_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 = sites. So Transition 4 is disabled and Transition 5 is enabled. The algorithm reaches a level of knowledge such that the spanning tree is completely built and where each site knows its parent and children.

Sketch of proof

From a theoretical point of view each introduced model should be proved to be constant with the previous refinement level. For example the proof of group level specification #1 (Figure 7.6) 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:

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 spanning tree(network, s_tree) $\wedge$

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

More complex techniques are also needed. As the main purpose is to build a spanning tree after some beginning steps, the problem is to show that actually a spanning tree structure is to be built. In this way, we apply a recurrent scheme for group level specification #3 (Figure 7.8): Transition 3 is the initialization step, Transition 4 is the proof that if the property is verified at step n then it is verified at step n+1 and Transition 5 is the stop condition.

- **Transition 3: initialization step**

Hypothesis: $\exists$ i $\in$ sites $state_i$ = sleeping, Condition 3 is verified and Action 3 is performed.

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

The graph ({initiator},$\varnothing$) 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 $\in$ {initiator}, K_i(parent({i})): because the initiator knows that it is an initiator and it is its own parent.

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

Conclusion: k_tree(k_sites, k_edges) $\wedge$

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

- **Transition 4: recurrence step**

Hypothesis: k_tree(k_sites, k_edges) ∧
∀ i ∈ k_sites, K_i(parent({i})) ∧ ∀ i ∈ k-1_sites, K_i(children({i}))
Condition 4 is verified and Action 4 is performed.

- (k+1_sites = k_sites ∪ accepted_children(k_sites)) ⊆ sites
  (k+1_edges = k_edges ∪ edges(accepted_children(k_edges))) ⊆ edges
  so (k+1_sites, k+1_edges) is a sub-network of the main network.
- By hypothesis | k_sites | = | k_edges | + 1
  k+1_sites = k_sites ∪ accepted_children(k_sites)
  k+1_edges = k_edges ∪ edges(accepted_children(k_sites))
  edges(accepted_children(k_sites)) denotes the edges between the elements of k_sites and the children that accept to be added. As a child accepts the first proposition and rejects the next ones, there is exactly one added edge for each added child
  so | accepted_children(k_sites) | = | edges(accepted_children(k_sites)) |
  so | k+1_sites | = | k+1_edges | + 1
- By the same way we prove connected(k+1_sites, k+1_edges)

So k+1_tree(k+1_sites, k+1_edges) and
∀ i ∈ k+1_sites, K_i(parent({i})) and ∀ i ∈ k_sites, K_i(children({i}))

Conclusion: k+1_tree(k+1_sites, k+1_edges) ∧
∀ i ∈ k+1_sites, K_i(parent({i})) ∧ ∀ i ∈ k_sites, K_i(children({i}))

- **Transition 5: stop condition**

Hypothesis: k_tree(k_sites, k_edges) ∧
∀ i ∈ k_sites, K_i(parent({i})) ∧ ∀ i ∈ k-1_sites, K_i(children({i}))
Condition 5 is verified and Action 5 is performed

k_sites = sites (this is Condition 5)
k = 0: k_tree(k_sites, k_edges) is verified.
Indeed, it has been proved with Transition 3.
k_tree(k_sites, k_edges) ⇒ k+1_tree(k+1_sites, k+1_edges)
This is the proof of Transition 4.
⇒ ∀ k ∈ D, k_tree(k_sites, k_edges)

Conclusion: spanning_tree(network, s_tree) ∧ ∀ i ∈ sites, K_i(parent({i})) ∧ K_i(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.

7.5. Object specifications

Let us give more details about the 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 behavior 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 behavioral constraints defined in previous models.

The previous group specifications design the evolution of a distributed algorithm through a set of sites. The spanning tree construction algorithm is a symmetrical algorithm. Each site runs the same behavior in the network. Then, we have only to describe one object model with as many refinement steps as needed. Objects (or object classes) are made of methods and global variables. Due to the lack of space we only give the first diagram. The first stage is to determine the global variables according with hypotheses and knowledge of the group specifications. Then we define the main methods (**construction** and **termination)** and their interface that correspond to the two recursive waves. Then we can define the first object specification diagram.

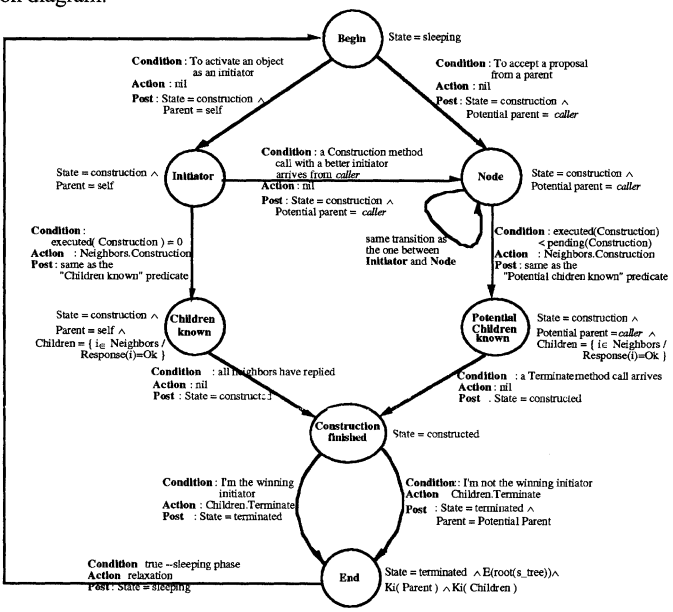

Figure 7.9: Object level specification.

As the reader can witness many refinements are not detailed. They are in [Bonnet 94].

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 behaviors. 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. The behavioral 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 behaviors. This last behavior must be clearly distinguished from the normal behavior. Today our implementations are realized on the Guide operating system that is a distributed object-oriented tool. Another large project is to develop a set of tools that takes into account and implements all the previous concepts.

9 ACKNOWLEDGEMENT

We thank Ivan Lavallée for many fruitful discussions about the spanning tree example and the anonymous referees for their accurate and stimulating comments. This research was partially supported by GDR-PRC of CNRS-MENESR, Parallelism, Networks and System group, Model and Algorithmic for Cooperative Systems sub-group.

10 REFERENCES

[Abrial 95] J.R. Abrial. "The B-Book: Assigning Programs to Meanings", Cambridge University Press, 1995, to appear, and Polycopié de cours, Valeur C, CNAM, 1994.

[Balter 91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, K. Krakowiak, P. Le Dot, M. Meysembourg, H. Nguyen, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandome. "Architecture and implementation of Guide, an Object-Oriented Distributed Systems", Computing Systems, 4(1):31-67, 91.

[Bonnet 94] L. Bonnet. "Spécifications et Preuves d'Algorithmes Distribués en Approche Orientée Objet", mémoire CNAM, Rapport de Recherche CEDRIC 94-22, Dec 94.

[Gries 81] D. Gries. "The Science of Programming", Springer Verlag, 1981.

[Halpern 90] J.Y. Halpern, Y. Moses. "Knowledge and Common Knowledge in a Distributed Environment", Communications of the ACM, 1990.

[Halpern 92] J. Halpern, Y.Moses. "A Guide to Completeness Complexity for Modal Logics of Knowledge and Belief", Artificial Intelligence, 54 (1992) pp 319-379, Elsivier.

[Hoare 69] C.A.R. Hoare. "The Axiomatic Basis of Computer Programming", Communications of the ACM, 69.

[Lamport 78] L. Lamport. "Time, Clock and the Ordering of Events in a Distributed System", Communication of the ACM, Vol. 21, No 8, August 1978, pages 666-677.

[Lamport 94] L. Lamport. "The Temporal Logic of Actions", ACM Transactions on Programming Languages and Systems, 16(3):872-923, May 1994.

[Lavallée 94] I. Lavallée. "Arbre Couvrant et Vague Récursive", Personal communication.

[Owicki 76] S. Owicki, D. Gries. "An Axiomatic Proof Technique for Parallel Programs". Acta Informatica, 1976, Vol. 6, pages 319-336.

11 BIOGRAPHY

Laurent Bonnet is born on December 4th, 1964. He received his engineer diploma in computer science from CNAM (Conservatoire National des Arts et Metiers) in 1994. He has take part in the specification and proof method and he has realized the specification and the proof of the spanning tree algorithm as part of his engineer work thesis. He is now project leader at TELIS, a service company in Paris.

Laurence Duchien has obtained a DEA ("Diplome d'Etudes Approfondies") in 1984 and she received her Ph.D. degree in computer science in 1988 from the Paris VI University (université Pierre et Marie Curie). She worked in the Network team at MASI Laboratory. She joined the Fault-tolerant Distributed Systems team at CEDRIC Laboratory in 1991. She is now assistant professor in computer science at CNAM. Currently her research activities include distributed algorithms for cooperative applications and models of specification and proof for distributed object-oriented applications.

Gérard Florin is gratuated from ENSET Cachan ("Ecole Normale Supérieure de l'Enseignement Technique"). He received his "Doctorat de spécialité" and "Doctorat d'état" degree from the Paris VI university in 1975 and 1985. He is currently professor in computer science in CNAM and he leads the laboratory CEDRIC. His research interest has been in Stochastic Petri nets domain. His research interests now include distributed computing and fault tolerance.

Lionel Seinturier is born on November 7th, 1970. In 1993, he has obtained an engineer diploma in computer science from IIE (Institut d'Informatique d'Entreprise) in Evry, France. In the same year he has obtained a DEA in distributed systems from University of Evry Val d'Essone, France. Since 1994 he is a PhD student in Professor Gérard Florin's team at CEDRIC-CNAM in Paris, France. He is working on the specification of distributed object-oriented algorithms. His current fields of interest are multi-agent systems, computer supported cooperative work and network algorithms.